Moving from quality attribute requirements to architectural decisions[1]

Felix Bachmann, Len Bass, Mark Klein
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pa USA 15213
{fb,ljb,mk}@sei.cmu.edu

Abstract

Quality attribute models are proposed as the linkage between a specification of a quality attribute requirement and a design fragment that is focused on achieving that requirement. Each quality attribute model has a collection of parameters that must be specified in order to determine from the model whether a requirement will be met. These parameters can be bound through design decisions, through values given from a quality requirement, or through knowledge of the designer. Architectural tactics are designed to relate design decisions to control of a quality attribute model parameter in order to achieve particular responses.

In this paper, we present a series of steps that enable moving from a single quality attribute requirement to a design fragment focused on achieving that requirement. We demonstrate these steps through application to an embedded system.

1. Introduction

It is well accepted that the satisfaction of quality attribute requirements for a software system depends heavily on the design of the software architecture for that system. From this a plausible design approach is to use the quality attribute requirements as primary when designing the software architecture. In order for this approach to be successful, four pieces must be in place: precise specification of quality attribute requirements, enumeration of fundamental design approaches to achieve various quality attributes, a linkage between the specification of the requirements and the appropriate design approaches that yields a design fragment focused on achieving the requirement, and a method for composing the design fragments into an actual design.

In this paper, we focus on the third of these pieces: the linkage between a specification of quality attribute requirements and a design fragment focused on achieving that requirement. We build on our prior work on quality attribute scenarios and architectural tactics and propose the use of quality attribute models as the linkage mechanism. We demonstrate the linkage through deriving a design fragment based on a performance requirement. An application of these steps to an additional modifiability scenario is precluded by space limitations but is available in [2].

We begin by briefly summarizing our prior work in quality attribute scenarios and architectural tactics. We then discuss why quality attribute models are the missing link and how they can be exploited to derive design fragments from quality attribute requirements. We illustrate the linkage through an example of a garage door opener.

2. Quality attribute scenarios

Quality attributes as defined in standards such as ISO 9126 [7] are not adequate for design. This is because the definitions do not reflect the context in which they are applied. For example, all systems are modifiable for some set of changes and not modifiable for others. The key for design is characterizing the set of changes that a particular system will be subjected to. Similar comments hold for other attributes.

We characterize quality attributes through quality attribute scenarios and have used this characterization in the ATAM[sm] [5] evaluation method as well as in other methods. Our current definition of a quality attribute scenario has 6 parts – stimulus, source of stimulus, environment, artifact being stimulated, response, and response measure. Quality requirements for a particular system can be cast in terms of these six parts.

In Chapter 4 of [3], we present scenario generation tables for the quality attributes of availability, modifiability, performance, security, testability, and

[1] This work supported by the U.S. Department of Defense

usability. Table 1 gives the scenario generation tables for performance – the attribute we use for our illustration. The scenarios generated by the tables in [3] cover most of the common meanings of these attributes [4].

The scenarios generated by these tables are "general" in that they are system independent. In order to make them act as requirements for a particular system, they must be instantiated for that system and made "concrete".

Portion of scenario	Possible Values
Source	– one of a number of independent sources – possibly from within the system
Stimulus	– periodic events arrive – sporadic events arrive – stochastic events arrive
Environment	– normal conditions – overload conditions
Artifact	– System – Process
Response	– processes stimuli – changes level of service
Response measure	– latency – deadline – throughput – jitter – miss rate – data loss

Table 1: performance scenario generation table

3. Architectural tactics

Experienced architects have a collection of techniques that they use to improve a system response with respect to a particular quality attribute. Some of these techniques are captured in patterns of various sorts but others, such as "reduce computational overhead" or "limit options the system will support", are not.

We have coined the term "architectural tactic" to describe these techniques and define an architectural tactic as a means of controlling a quality attribute measure by manipulating some aspect of a quality attribute model through architectural design decisions. In Chapter 5 of [3], we provide an enumeration of architectural tactics, albeit with a different definition.

Observe that an architectural tactic is concerned with the relationship between design decisions and a quality-attribute response. This response is usually something that would be specified as a requirement (e.g., an average latency requirement). Therefore architectural tactics (by definition) are points of leverage for achieving quality-attribute requirements even though, as yet, no guidance is provided as to how to choose appropriate tactics in particular situations.

Table 2 enumerates the architectural tactics used to achieve performance. See [2] for a description of the meaning of each tactic.

Category of tactic	Architectural tactic name
Manage Demand	– manage event rate – control frequency of sampling external events – reduce computational overhead – bound execution times – bound queue sizes – increase computational efficiency of algorithms
Arbitrate Demand	– increase logical concurrency – determine appropriate scheduling policy – use synchronization protocols
Manage Multiple Resources	– increase physical concurrency – balance resource allocation – increase locality of data

Table 2: performance architectural tactics

4. Quality attribute models

Associated with every quality attribute are one or more "reasoning frameworks" that allow prediction of the response of a system with respect to particular attributes. Performance frameworks such as queuing theory or scheduling theory are the best known and studied and they are very quantitative in nature. Frameworks for modifiability include those based on coupling and cohesion [6] and those based on dependency analysis [1]. These are much more qualitative but still allow prediction of the difficulty of a modification. Other frameworks exist for other attributes. Each framework has uncertainty in terms of the accuracy of its predictions but these frameworks have proven useful in assisting designers.

It is these quality attribute reasoning frameworks and their associated models that we exploit to link quality

attribute requirements (specified as concrete quality scenarios) and architectural design decisions (as embodied in architectural tactics).

Every quality attribute reasoning framework has a collection of types of entities that are included in the framework. Performance models, for example, have units of concurrency such as threads or processes, dependency among these units of concurrency, and resources. There is some collection of inputs (arrival rates, resource requirements) that drives the model. We call all of these "parameters" of the models. These are the items that a designer may potentially control to enable the achievement of a desired response.

5. Linking concrete scenarios to architectural tactics

Our goal in this section is to describe how to derive a set of tactics that are relevant for achieving a particular concrete scenario and then use this to derive candidate design fragments. This carried out using the following set of steps. We assume that input to the set of steps is a concrete scenario and some set of already made design decisions exists.

1. *Identify candidate modeling frameworks*. It may be that some of the information from the concrete scenarios will eliminate possible modeling frameworks. For example, if we know that arrivals are periodic then the queuing modeling framework is eliminated from considerations. Each reasoning framework has a collection of parameters that must be set before the reasoning framework can be applied.
2. *Determine bound and free parameters*. The candidate modeling framework has a number of parameters. Some of these may be given by the concrete scenarios and some may be given by elements of the existing design that are not changeable. For example, a concrete scenario may specify "events arrive periodically". This may require a specific scheduling model. Another element of the existing design might be that a particular operating system is to be used. This determines the execution time associated with processing one event. This is a parameter of the model that is bound. All parameters not bound are considered free.
3. *Enumerate tactics associated with the free parameters.* Because a tactic controls one of the parameters of a model in the reasoning framework, we can list the tactics associated with the free parameters, which we use as candidate tactics for the next steps.
4. *Assign free parameters an initial set of values.* The designer makes an estimate for each free parameter based on intuition or knowledge. If the designer has no intuition or knowledge for a particular parameter then an arbitrary value might be chosen. If this parameter is important to the system, an implementation of a prototype might be appropriate to get an estimate.
5. *Use tactics to develop satisfactory bindings for all free parameters*. This step has two degrees of freedom – the list of candidate tactics and the set of free parameters. We begin our description by considering the situation where there is only one free parameter.
   Each of the candidate tactics for this free parameter controls its value – that is, it allows the adjustment of the free parameter. For each candidate tactic, determine whether it can adjust the value of the free parameter to a new value where the solution of the resulting model satisfies the response measure of the concrete parameter. If it can, then it becomes a relevant tactic. If it cannot, then it is discarded.
   Now consider multiple free parameters. In this situation, we need to consider simultaneously adjusting all free parameters. That is, if tactic one controls parameter 1 and tactic two controls parameter two, we need to determine whether we can move the value for parameter 1 through tactic 1 and the value for parameter 2 through tactic 2 until the dependent variable for a resulting model satisfies the response measure given by the concrete scenario. If we can then we add both tactics to our list of relevant tactics, if we cannot then we discard both tactics. If we have more than one tactic for each parameter, we need to consider all possible combinations of tactics for the parameters.
6. *Allocate responsibilities to architectural elements.* Every tactic enumerated in table 2 has a design fragment assigned, if appropriate. For example one performance tactic suggests using a certain type of scheduler, or a modifiability tactic recommends the use of an intermediary. Applying those fragments to an existing design moves the architecture to a state that supports the scenario, as demonstrated by the modeling framework.

   Design fragments come with their own responsibilities and a set of rules that help to:
   - Create/delete/refine design elements
   - Add responsibilities to existing design elements
   - Reallocate responsibilities of already existing design elements
   - Refine responsibilities and allocate them to design elements

   For example using the tactic *semantic-importance-based scheduling* includes applying the following rules:
   - Create a design element "scheduler"

- Allocate the responsibilities with higher importance to units of concurrency with higher priority

or using the tactic *break the dependency chain* includes applying the following rules:

- Create a design element "intermediary"
- Add responsibilities to the intermediary that translate from the more abstract interface provided to the secondary modules to the concrete interface provided by the primary module
- Refine the responsibilities of the secondary modules to use the services of the intermediary

6. Garage door example

Our sample design problem is that of a garage door opener. The controller for a garage door opener is an embedded real-time system that reacts to open and close commands from several buttons installed in the house and from a remote control unit, usually located in a car. The controller then controls the speed and direction of the motor, which opens and closes the garage door. The controller also reacts to signals from several sensors attached to the garage door. One of the sensors detects resistance to the movement of the door. If the amount of resistance measured by this sensor is above a certain limit, then the controller interprets this as an obstacle between the garage door and the floor. As a reaction, the motor closing the garage door is stopped.

There are many scenarios that specify the requirements for the controller software. In [2] we present both a performance and a modifiability scenario. Here space limits us to just discussing the performance scenario.

If an obstacle (person or object) is detected by the garage door during descent, it must halt within 0.1 seconds.

We now exemplify our steps for this scenario.

1. Identify candidate reasoning frameworks

There are two performance reasoning frameworks that might be applicable to a performance scenario: queuing theory and scheduling theory. We know from looking at our concrete scenario that we have sporadic event arrivals and a hard deadline requirement. The hard deadline requirement suggests that the applicable reasoning framework is scheduling theory. Sporadic arrivals are arrivals that cannot occur arbitrarily often. This is an indicator that there is a bound on the arrival rate variability, again indicating that scheduling theory is relevant. The other relevant parameters are: execution time, number of units of concurrency, and number of processors.

2. Determine bound and free parameters

In this step the scenario is recast in terms of the bound and free parameters of the applicable reasoning frameworks. Scheduling theory is concerned with calculating worst case latency associated with carrying out each scenario, given the execution time, arrival period associated with each unit of concurrency, the number of units of concurrency and how each unit is allocated to one or more processors. Worst-case latency can then be compared with the hard deadline to determine if the requirement is satisfied or not.

For these parameters we first determine which ones our concrete scenario binds. One parameter is the arrival distribution. In this case the arrival distribution describes how often an obstacle is detected. We assume this happens infrequently and there is a bound on how frequently it occurs (known as a sporadic arrival distribution). We assume from the business context of the garage door opener that a single processor will always be adequate.

Since this is the one performance scenario considered in this example we do not yet have any bound parameters in the selected reasoning framework from previously made decisions.

To summarize:

- Bound parameters: arrival distribution and number of processors
- Free parameters: number of units of concurrency and execution time of responsibilities

3. Enumerate tactics associated with the free parameters

This is where we start to employ our "decision procedures", which are really a loosely structured set of rules for which tactics to try (see Table 3). In this step the decisions are based strictly on what parameters are considered fixed and which are considered free.

1) Which parameters are fixed?

- Arrival distribution – arrivals are infrequent.
- Number of processors – we will assume that our platform constrains us to a single processor

From the first rule in Table 3 we conclude that the fixed arrival distribution rules out the following tactics: *Manage event rate* and *Control the frequency of sampling external events*

The architect constrains the solution to a single processor because of the business context and this rules out the following tactics: *Increase physical concurrency*, *balance resource allocation*.

2) Which parameters are free?

- Execution time – The responsibilities will suggest a likely range, but this is not yet fixed.

- Number of units of concurrency – This is free and will be determined later in design

The following tactics are concerned with manipulating execution time: *Reduce computational overhead, Increase computation efficiency, Control the demand for resources* and *Bound execution time*

Some of rules of our performance decision procedure that are applicable for this step are shown in Table 3.

Table3 Example rules of our performance decision procedure

- If the arrival distribution is fixed then *Manage event rate* and *Control the frequency of sampling external events* are not tactics that can be used to control worst-case latency.

- If execution time is a free parameter then consider using the following tactics: *Reduce computational overhead, Increase computation efficiency, Control the demand for resources* and *Bound execution time*

- If the number of processors is bound then eliminate the following tactics as candidates: *Increase physical concurrency* and *balance resource allocation.*

4. Assign free parameters an initial set of values

Two things occur at this step. First, the architect offers his/her best guess for values for the free parameters. The list of applicable tactics suggests factors that impact the setting of these values. Secondly, rules of the decision procedure call attention to possibly problematic situations.

From the previous steps we know that two of the tactics are relevant to estimating execution time: *Reduce computational overhead* and *Bound execution time.* The architect might guess that the sum of the execution time of the 3 responsibilities is about 5 msec. *Bound execution time* calls our attention to the effects of execution time variability, however the architect predicts that these responsibilities have very little variability. *Reduce computational overhead* calls attention to various sources of overhead that represent extra execution time. It is conceivable that each one of the responsibilities involved in obstacle detecting - "detect obstacle", "determine that garage door is descending", and "halt garage door " - incur some OS overhead for some pre-selected real-time operating system. Consequently the architect estimates that the operating system adds an addition 1 msec of overhead. The architect also assumes that all of this scenario's responsibilities are allocated to a single unit of concurrency. This last assumption is possible because this is the sole scenario considered. We discuss some of the issues involved in multiple scenarios in a further section.

While the architect does not yet know all of the details of the other responsibilities in the system, he or she does know that there will be other responsibilities with associated execution times and these other responsibilities hold potential for adversely affecting the ability of this scenario to be realized. The architect is not yet ready to assign values to the execution times associated with these other responsibilities.

The second consideration at this stage is to examine the scenario to determine if it is unreasonable or problematic. For example, if execution times or arrival rates vary considerably, but deadlines can never be missed, this might be problematic. Examples of rules that call attention such potentially problematic situations are in the table below. However, for the current scenario none of these situations apply.

Some of rules of our performance decision procedure that are applicable for this step are shown in Table 4.

Table 4 More example rules of our performance decision procedure

- If the scenario has a hard deadline response requirement that cannot be and if arrivals can occur arbitrarily close to one another then use one of the following tactics to ensure a lower bound for the inter-arrival interval: *Manage event rate* and *Control sampling frequency*.

- If the scenario has a hard deadline response requirement that cannot be relaxed and if execution times vary considerably to the point that they can approach or exceed the hard deadline then consider applying the following tactic*: Bound execution time.*

- If either of the above "unbounded" conditions apply, but arrival rate and execution time are bound parameters then declare the requirement untenable

5. Use tactics to develop satisfactory bindings for all free parameters

At this point all of the parameters have values and there is a candidate list of applicable tactics. The first thing is to look at one or more of the applicable tactics and apply the reasoning framework (in this case scheduling theory) to determine if the current concrete scenario is satisfied without "violating" any of the scenarios that have already been satisfied.

The relevant tactics entering into this step are:

- Controlling resource demand through *Reduce computational overhead, Increase computation efficiency, Control the demand for resources* and *Bound execution time* have a bearing on how execution time affects worst case latency.

- *Increase logical concurrency* and *determine scheduling policy* both have a bearing on understanding how this scenario's responsibilities affect and/or are affected by the other responsibilities in the system

Without considering the effects of other responsibilities, the model is fairly simple. The only contributors to latency are execution time and overhead, 5 msec and 1 msec respectively. Their sum is well under the deadline of 100 msec (that is, .1seconds), leaving 94 msec to spare.

On the other hand it is very conceivable that the other responsibilities in the system take more than 94 msec. Using the last rule in the table below suggests that the design decisions made in the next step be consistent with our simple model, that is, they ensure that the latency associated with this scenario's responsibilities is not affected by any of the other responsibilities.

Some of rules of our performance decision procedure that are applicable so far for this step are shown in Table 5.

Table 5 More example rules of our performance decision procedure

- If the execution time associated with the arrival is close to the deadline consider reducing execution time by using the following tactics: *Reduce overhead, Bound execution time, and/or Increase computation efficiency.*

- If the difference between the worst and best case is significant then review the following tactics and apply their modeling techniques to assess miss rates and average latency respectively: *Bound execution times* and/or *Bound queue sizes*

- If the response requirement for all scenarios can be achieved even with the worst-case delay due to all of the other responsibilities of all of the other scenarios, then use any the following tactics:
  - Allocate responsibilities to one of the existing units of concurrency
    - *Offline scheduling*
  - Or allocate responsibilities to a new unit of concurrency
    - *Increase logical concurrency*
    - *Time-based scheduling*

If the current scenario cannot suffer the worst-case delay due to some or all of the other responsibilities then consider them to be time-sensitive and use the following tactics to create an appropriate scheduling policy: *Offline scheduling, Time-based scheduling (such as deadline monotonic scheduling), and/or Increase logical concurrency.*

Up to now in this step tactics have been used to set and/or adjust model parameters to satisfy the current concrete scenario's response measure. However, it might be the case that either the scenario poses an untenable requirement or the collection of scenarios considered up to this point are untenable in aggregate. If this is the case, tactics should offer some ideas for how to relax requirements or design constraints.

Some of rules of our performance decision procedure that are useful for identify and relaxing requirements and/or design constraints are shown in Table 6.

Table 6 Some of rules for relaxing requirements and/or design constraints

- If the response requirement is specified as a hard but limited misses can actually be tolerated then re-characterize deadlines as follows:
  - Firm deadlines: Completing before the deadline is very important. Missing occasionally can be tolerated. A specific bound on miss rate needs to be specified.
  - Soft deadlines: In this case the term "deadline" is a misnomer. A specification of an average latency requirement is what is needed.

- If the time-sensitive set of responsibilities is not schedulable then incorporate a notion of importance-based scheduling to handle overload situations using *Semantic-importance-based scheduling* or add more resource using *Increase physical concurrency.*

6. Allocate responsibilities to architectural elements

Each tactic will suggest associated design fragments. The tactics of primary concern so far in this example are:

- Controlling resource demand through *Reduce computational overhead, Increase computation efficiency, Control the demand for resources* and *Bound execution time* have a bearing on how execution time affects worst case latency.

- *Increase logical concurrency* and *determine scheduling policy* both have a bearing on

understanding how this scenario's responsibilities affect and/or are affected by the other responsibilities in the system

Reducing computational overhead can map to many design decisions such as:

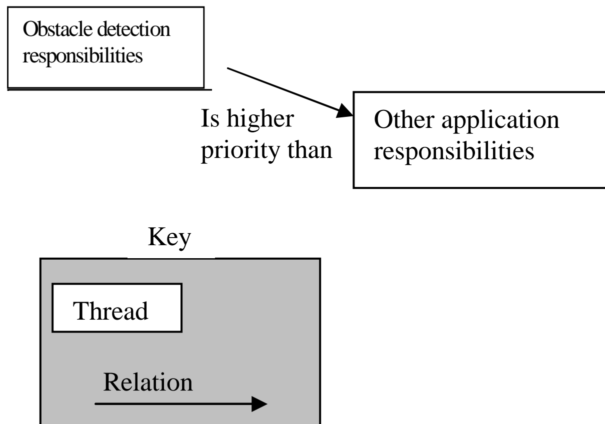
- choice of operating system
- choice of operating system services used in implementing responsibilities
- choice of communication mechanisms, …

We have accounted for OS responsibilities by assuming 1 msec overhead. We have also assumed that all of the scenario's responsibilities have been allocated to a single unit of concurrency that we will assume is a thread.

The responsibilities for this scenario are pretty straightforward;

Tactics of *Increase computation efficiency, Control the demand for resources* and *Bound execution time* are likely not relevant whereas tactics of *Increase logical concurrency* and *determine scheduling policy* are relevant. They suggest allocating the obstacle detection responsibilities to a particular module under one thread of control and assigning this thread a suitably high scheduling priority. This results in an design fragment with two threads: the one containing the obstacle detection responsibilities and the one containing other responsibilities. We do not show the scheduler (which is part of the OS) although that also is a portion of the design fragment. We show a component and connector view of this fragment in Figure 1.

Figure 1: Design fragment



In an ideal world obstacle detection will take only as long as it takes to execute the obstacle detection responsibilities plus a little overhead. However, the possibility exists that properties of the *Other responsibilities*, such as non-preemptability or execution within an interrupt handler might not have been accounted for. Therefore these potentially problematic properties need to be discovered and/or ruled out. We expect that rules in the *time-based scheduling* tactics would cause us to look for and/or ensure against such properties

Observe the relationship between the design fragment and the associated analysis model. The model states that obstacle detection responsibilities must be scheduled with a priority high than other responsibilities. The design fragment captures this by placing these responsibilities into separate threads and showing the priority relationships of those threads.

7. Composition

We have shown how to use the tactics to link one quality attribute performance requirement to a design fragment with active assistance from an architect. The gaping open issue is what happens with multiple scenarios involving multiple quality requirements, especially for other attributes. How to compose the design fragments into a design is the fourth step of moving from quality requirements to design and it must clearly be solved for this approach to be successful.

Some of the problems that must be solved to achieve the composition of design fragments into designs are:

- How to consider the impact of design decisions already made.
- How to choose among the myriad of possibilities of composing design fragments. In [2] we identified a design fragment for modifiability as well as one for performance and there were multiple composition possibilities
- How to maintain view consistency. Each quality attribute framework has a vocabulary that maps into one or more software architecture views. Maintaining consistency between fragments that come from one reasoning framework with those that come from another is a problem that must be solved.

8. Other open issues and conclusions

In addition to the composition problems there are two other problems that must be overcome.

1. What is the availability and utility of the various reasoning frameworks for other quality attributes? Involving the architect, as we did, in the design process allows judgment to be used in application of the reasoning frameworks. We can predict that, over time, reasoning frameworks for various quality attributes will improve.

2. How do the steps we have presented here become embedded into a design method? Once quality requirements become recognized as important to design they will begin to be specified in the 1000s as are functional requirements. This leads to over specification of the requirements. A design method must be sensitive to this over specification.

Regardless of the problems, focusing on quality attribute requirements and using them to drive towards an appropriate architectural design must be a useful approach. The utilization of quality attribute models and tactics in this process is our attempt to move design toward a more scientific basis.

9. References

[1]Bachmann, F., Bass, L., and Klein, M. Illuminating the fundamental contributors to Software Architecture Quality. CMU/SEI-2002-TR-025

[2] Bachmann, F., Bass, L., and Klein, M. Deriving Architectural Tactics – A Step toward Methodical Architectural Design CMU/SEI-2003-TR-004

[3] Bass, L., Clements, P. and Kazman, R. Software Architecture in Practice, 2nd edition. 2003, Addison-Wesley.

[4]Bass, L., Klein, M., Moreno, G., Applicability of General Scenarios to the Architecture Tradeoff Analysis Method. CMU/SEI-2001-TR-014

[5] Clements, P., Kazman, R., and Klein, M., Evaluating Software Architectures. Addison Wesley, 2002

[6]Henry, S. and Kafura, D. Software Metrics Based on Information Flow. IEEE Transactions on Software Engineering. SE-7(5), Sept. 1981

[7] . *International Standard ISO/IEC 9126. Information technology -- Software product evaluation -- Quality characteristics and guidelines for their use*, International Organization for Standardization, International Electrotechnical Commission, Geneva.