Designing software architectures to achieve quality attribute requirements

F. Bachmann, L. Bass, M. Klein and C. Shelton

Abstract: In order to have a software architecture design method that achieves quality attribute requirements several aspects of the method must be in place. First there must be some way to specify quality attribute requirements so that it can be determined whether the designed architecture can achieve them. Secondly, there must be some way for modularising the knowledge associated with quality attributes so that the design method does not need to know how to reason about all of the multiplicity of quality attributes that exist. Finally, there must be some way for managing the interactions among the quality attributes so that either the requirements can be satisfied or the ones that cannot be satisfied are identified. The authors describe a structure called a ‘reasoning framework’ as a modularisation of quality attribute knowledge. The requirements that the architecture must satisfy are specified as concrete quality attribute scenarios. Each reasoning framework provides mechanisms that will transform the architecture with respect to a given quality attribute theory. Within a reasoning framework, the authors distinguish between an architectural model and a quality attribute model and characterise the actions that a reasoning framework undertakes as basic architectural transformations. Finally, the process of identifying interactions among reasoning frameworks is begun so that conflicting requirements can be managed. The use of reasoning frameworks is situated inside an existing architectural design method so that a useful method exists while the open issues of designing to achieve quality attribute requirements are resolved.

1 Introduction

Designing an architecture so that it achieves its quality attribute requirements is one of the most demanding tasks an architect faces. It is demanding for several reasons including a lack of specificity in the requirements, a shortage of documented knowledge of how to design for particular quality attributes, and the trade-offs involved in achieving quality attributes.

It would be desirable to have a method that guides the architect so that any design produced by the method will reliably meet its quality attribute requirements. Any such method must have several features:

- It must have provision for including a wide variety of quality attribute knowledge. ISO 9126 -1 [1] lists six primary quality attributes and 21 secondary quality attributes and this list is not complete, e.g. it omits variability among others. A quality attribute requirement can affect any attribute whether on the list or not and so a method to achieve all quality attribute requirements must embody great breadth in reasoning about quality attributes.
- It must have provision for managing trade-offs that is outside any of the existing quality attribute specific knowledge. Some formal knowledge exists about trade-offs between specific attributes but any method that claims to be able to design to achieve any quality attribute requirement must be able to manage trade-offs among arbitrary quality attributes. Such broad knowledge does not yet exist.
- It must have provision for interpreting quality attribute knowledge in terms of software architectures and for interpreting a software architecture in terms of the quality attribute knowledge. Software architectures are described in terms of software elements and their relations. This can include hardware elements in addition to behavioural descriptions. Quality attribute knowledge is embodied in other terms. The model of modifiability we use[2] considers dependencies among modules, probabilities of change propagating and cost of modification. These concepts can be translated into software architecture concepts such as relations among modules and properties of modules and relations but any method must have a provision for making that translation.

In this paper we propose a modularisation of quality attribute knowledge that encapsulates both quality attribute models and the relationship between these models and software architecture and give examples of this encapsulation. We define what it means to have a trade-off in terms of the modular concept. Finally, we embed these concepts into a pre-existing design method to give some structure to their use.

2 Prior work on satisfying quality attribute requirements

There are three basic approaches to software architecture design with respect to quality attribute requirements.
These are the NFR approach that originated at the University of Toronto, the quality attribute model approach, and the intuitive approach.

2.1 Non-functional requirement (NFR) approach

The basic idea behind the non-functional requirement (NFR) approach, originated by Chung, Nixon, Yu and Mylopoulos [3], is that whether a software architecture will achieve its quality attribute requirements is not precisely determinable. This leads to the idea that the best that can be determined is that various architectural decisions help or hurt various quality attributes, e.g. modularisation helps modifiability and hurts performance. In turn, this leads to a decision tree that an architect can traverse when determining what decisions can be made to help achieve particular quality attributes. There are two problems with this approach:

(i) Because it is not precisely determinable whether a quality attribute requirement will be met by a particular design, the architect must rely on intuition to determine which decisions to consider.
(ii) Whether a particular decision will help or hurt a particular quality attribute is very much a question of context. Modularisation only helps modifiability when the modularisation supports the changes that will occur. Modularisation that cuts across changes will hurt modifiability. Modularisation only hurts performance when it introduces an overhead that is not compensated by other actions. Modularisation that supports deployment onto multiple processors will help performance, not hurt it. In other words, there are a number of assumptions that go into each branch of the decision tree, these assumptions are implicit rather than explicit, and there is no method for knowing in a particular context whether the assumptions are valid.

2.2 The quality attribute model approach

The quality attributes have developed communities that study them and have developed methods for measuring these attributes. ISO9126-2 and 3 [4] list a multiplicity of metrics for various quality attributes. There is a continued search within specific communities for models that enable the prediction of these measures. The community with the most notable success is the performance community with a large variety of models for both scheduling theory and queuing theory. Models for other attributes are less successful but still provide insight into understanding the behaviour of systems.

There are three problems with using quality attribute models as the basis for design:

(i) Not every quality attribute has an existing predictive model. Attributes such as security and usability have proven difficult to develop predictive quantitative models that relate software architectural decisions to quality attribute measures.
(ii) Not every architecture is amenable to analysis by various quality attribute models. For example, rate monotonic analysis (RMA) [5] makes assumptions about the topology of the tasks in the architecture. For architectures for which the topology violates those assumptions, the results of an RMA model would not be valid. For some architectures, there are no appropriate models.
(iii) Quality attribute models may not scale well. Solving a queuing network with forty nodes may be exponentially more difficult than solving one with five nodes. The normal approach is to model only the essential elements of an architecture but this involves judgment as to which elements are the essential ones.

2.3 The intuitive design approach

In recent years there have been several quality attribute centred software architecture design methods including our own attribute driven design (ADD) method [6] and the QASAR method of J. Bosch [7]. These methods are effective at organising requirements but depend to a large extent on the architect to find solutions for the satisfaction of specific quality attribute requirements.

We are embedding this work into the ADD method and so we briefly describe this method. It is a recursive decomposition method with 3 main steps. The key step is step 2 where the next decomposition is derived (see [6] for a more detailed description). Within step 2, we will be focusing in this paper on steps 2b and 2c. The steps of ADD are

1. Choose a part of the system to decompose
2. Refine the chosen part of the system according to the following steps
   a) Choose architectural drivers
   b) Choose architectural patterns and tactics that satisfy the architecture drivers
   c) Allocate remaining functionality
   d) Define interfaces for the instantiated elements
   e) Verify and refine the functional requirements, quality scenarios, and constraints and make them constraints for the instantiated elements
3. Repeat steps 1 through 2e for the next part of the system you wish to decompose

3 Our approach

A key concept in our design philosophy is describing an architecture partially in terms of responsibilities [8]. A responsibility is an obligation to perform a task or to know information to be carried out by a software system or an element of the system. Functional requirements, for example, can be characterised by responsibilities but so can computation in support of the achievement of quality attributes. For example, the checking of a password in support of an authentication requirement is a responsibility of the software.

We draw other elements of our approach from each of the three general approaches enumerated above. We use the concept first articulated in the NFR work of there being an enumerable number of architectural decisions that are made in order to achieve quality attribute goals. We use an enumeration of our own, however [6].

We also use quality attribute models for a number of reasons. They serve to embody the context for choosing among architectural decisions, they provide a rationale for the architectural decisions that we enumerate, and they provide the means to decide whether a proposed architecture design will satisfy particular quality attribute requirements. Although it is the case that existing quality attribute models are neither deep enough nor broad enough to model all quality attributes, the reality is that architects make decisions in order to achieve quality attributes and systems are being constructed that do, in fact, achieve quality attribute requirements. This gives hope that, in time, examination of the architectural decisions will yield a basis for models for those quality attributes where the current models are unsatisfactory. Until that time comes, we incorporate the concept of constraints into our approach. If an architect wishes to include an architectural decision to
support security, for example, then specifying a specific component as a firewall will enable the architect’s decision to be incorporated into the final design without the necessity of having a security model.

Finally, we embed the approach described in this paper inside of ADD. ADD has the property that the next set of design decisions is based on a small number of architectural drivers. This reduces problems of scale in using quality attribute models.

4 Modularising quality attribute knowledge

One of the goals of this work is to understand how to modularise quality attribute knowledge so that it can be used inside a general design method. A quality attribute reasoning framework is our unit of modularity. A reasoning framework encapsulates knowledge relative to one quality attribute theory and its associated models. Figure 1 shows the input into a quality attribute reasoning framework and the output from one.

These inputs and outputs constrain which quality attribute theories can be used as a reasoning framework. The reasoning framework must make a connection between the architecture description and its analysis. The reasoning framework must also be able to frame its output in terms of architecture transformation. This precludes such quality attribute theories as risk-analysis-based security theories. Risk-analysis-based security evaluation ask questions such as ‘have all relevant patches been applied?’ Although the patches themselves may be architecturally based, the risk-based theory does not deal with the underlying architectural causes and, hence, is not suitable for a reasoning framework in our terms.

One of the inputs into a reasoning framework is a description of the current state of the architecture design. Any constraints would be incorporated into the current design. The other input is a list of the relevant quality attribute requirements. The reasoning framework encapsulates knowledge of one quality attribute theory. Consequently, it can only reason about a subset of the quality attribute requirements. These are the relevant requirements in Fig. 1. In the following we discuss quality attribute scenarios as a form for the specification of quality attribute requirements including the response to a new requirement. The scenarios include triggers that are used to identify the relevant requirements for the reasoning framework. Also, the reasoning framework determines whether the current architecture design satisfies the relevant requirements. If all requirements are met, then the reasoning framework has no reason to propose any modifications to the architecture. If there are unmet requirements then the reasoning framework outputs the list of unmet requirements together with a set of proposed transformations to the architecture. The proposed transformations are chosen so that they will improve the architecture with respect to the unmet requirements and not cause any new requirements of the set considered by the reasoning framework to become unmet. It is possible that the set of requirements considered by a single reasoning framework cannot be satisfied simultaneously. The reasoning framework must detect and report such a condition.

Notice that we do not have the reasoning framework make any transformations to the architecture. Multiple reasoning frameworks will each make proposals based on their set of relevant requirements. Any transformations that are chosen will possibly have an impact on multiple quality attributes. Since the reasoning framework knows only about its own theory and nothing about other quality attributes, it cannot make a global choice of which transformation to perform. We will discuss the choice of transformations when we discuss the interaction of quality attributes.

This overview of the actions of a reasoning framework lead to the next topics we will discuss: what are the quality attribute requirements, what does it mean to satisfy one of these requirements, and what are the architectural transformations that are proposed? We will also illustrate the actions of two reasoning frameworks that we have been studying; rate monotonic analysis (RMA) [5] for real time performance, and impact analysis for modifiability [2, 9].

4.1 Quality attribute requirements

Our desire to keep quality attribute knowledge encapsulated has two implications. One that we have discussed is to keep quality attribute knowledge inside the modular structure. This is the purpose of the reasoning framework in this work. Keeping quality attribute knowledge inside the reasoning framework means keeping it out of the general design method. That is, the general design method should not know specifics of any particular quality attribute.

One consequence is that the general design method should have a form for specifying quality attribute requirements that is suitable for specifying any quality attribute requirement, regardless of quality attribute. We use quality attribute general scenarios for this purpose.

A general scenario is a system independent description of a quality attribute requirement. It has six parts:

- Stimulus: a phenomenon that needs to be considered when it arrives at a system (either executing or in development). Examples are an event arriving at an executing system or a change request arriving at a developer.
- Source of the stimulus: the entity (e.g. a human or computer system) that generated the stimulus. For example, a request arriving from a trusted user of the system will in some circumstances be treated differently than a request arriving from an untrusted user.
- Environment: the conditions under which the stimulus occurs. For example, an event that arrives when the system is in an overload condition may be treated differently than an event that arrives when the system is operating normally.
- Artifact: the artifact affected by the stimulus. Examples include the system, the network, or a subsystem.
- Response: the activity undertaken after the arrival of the stimulus. Examples include processing the event for event arrival or making the change without affecting other functionality for a change request.
- Response measure: the attribute-specific constraint that must be satisfied by the response. Examples are that the event must be processed within 100 ms or that the change must be made within two person-days. Response measures can also be Boolean such as ‘The user has the ability to cancel a particular command (yes or no)’. 

Fig. 1 Input into and output from a quality attribute reasoning framework
In [6], we provide a set of general scenario generation tables for the attributes of availability, modifiability, performance, security, testability, and usability. These tables can be used to elicit the specific quality attribute requirements for their particular attributes. A concrete scenario is an instantiation of a general scenario for the particular system under consideration.

A sample concrete scenario is: a remote user initiates a set of transactions to the system aperiodically no more often than one per second under normal operations. The transactions are to be processed with an average latency of two seconds.

Each reasoning framework will be able to reason about a particular set of quality attribute requirements expressed as concrete scenarios such as deadline performance for RMA or developer modification for impact analysis. The determination of which requirements are relevant for which reasoning framework can be done either within the reasoning framework or externally to it.

4.2 **Satisfying a quality attribute requirement**

Next we focus on a single quality attribute requirement and describe the elements we use to determine whether it is satisfied by the current architecture design. Figure 2 shows some of these elements. In the upper right corner, the Figure shows an architectural space (notionally) together with a particular architecture (labelled a) within that space. In the upper left corner is the space of quality attribute requirements together with a single requirement (not labelled). The quality attribute requirement has a response measure associated with it. The Figure also shows the space of possible response measures for that type of quality attribute requirement. The lighter circle in the space of quality attribute measures shows that set of measures that satisfy the particular quality attribute requirement. In the performance scenario example above, this would be all latencies with a value less than two seconds. In the space of quality attribute measures shows that set of measures that satisfy the particular quality attribute requirement. In the performance scenario example above, this would be all latencies with a value less than two seconds. In the space of quality attribute measures is a point that represents the predicted latency of the software architecture a. The Figure also shows a mapping from the architecture to its particular response. We put a question mark over the mapping from software architecture to the actual response measure because we have not yet described how to perform the mapping. Since the latency is dependent on the stimulus to the architecture we must also describe how the stimulus enters into the determination of the response. We do not assume that the response measure space is quantitative. The possible responses could be Boolean or some other non-quantitative measure.

4.3 **Evaluating response measures**

In order to predict the response measure of an architecture to a stimulus we assume the existence of a theory used to analyse a quality attribute. Each reasoning framework is based on a particular quality attribute theory. This theory provides the mechanisms necessary to calculate the response measure. Examples of such theories are rate monotonic analysis (RMA) [5] for real time scheduling and impact analysis [2, 9] for modifiability. These are the two theories we use for our example.

The process for measuring the response of an architecture involves mapping the architecture into a form amenable to being solved by the theory. We now discuss three elements of that process.

4.3.1 **Quality attribute model:** The quality attribute theory defines a vocabulary of concepts. A quality attribute model expresses relationships among the concepts. These relationships can be expressed by a formula, some form of analytic model such as a queuing model, or even a simulation model. What is important is that there is a set of variables in the model that can be derived from the information available (we will expand on this point shortly) and a dependent variable that can be calculated from the independent variables based on the relationships in the model. A quality attribute model instance is a quality attribute model where all of the independent variables have been bound to actual values. The binding is necessary in order to solve the model for the dependent variable.

4.3.2 **Sample quality attribute model:** We will use a sample quality attribute model derived from RMA to exemplify some of the other aspects of a reasoning framework. This example is based on rate monotonic analysis applied to a single processor on which multiple tasks reside, each of which perform computations on their own input data stream. The various tasks compete for the processor with only one task being allocated the processor at a time. The latency (the dependent variable for this model) is the time to completely process each message.
There are three types of time ‘experienced’ by an arbitrary task under these circumstances: pre-emption, execution, and blocking time. Pre-emption time is the contribution to latency attributable to higher priority tasks. Blocking is the contribution to latency due to low priority tasks. Blocking time arises as a consequence of the shared resource topology. Let $C_i$ denote the execution time of task $i$, $T_i$ denote the period of task $i$, and $B_i$ denote the blocking time incurred by task $i$. The worst-case latency for task $i$, assuming that tasks $1$ to $i - 1$ are of higher priority, can be found by iterating the following formula until it converges (that is, the value of $L_n$ remains the same for two consecutive iterations)

$$L_{n+1} = \sum_{j=1}^{i-1} \left( \frac{L_n}{T_j} \right) C_j + C_i + B_i$$

(1)

4.3.3 Solution: Given a quality attribute model instance, the theory tells us how to solve for the dependent variable. We call this the ‘solution’. For the example above, the instance comes from providing values for the execution time, the period, and the priority of each task. The solution comes from an iterative technique that enables the calculation of the latency for each task. Although our example provides a quantitative measure, there is no requirement that the measure be quantitative. It is only necessary to determine whether the predicted response satisfies the required response measure.

4.3.4 Interpretation: The quality attribute model and the solution method are not coupled to any particular architecture. We must have a method for deriving the values of the variables of the quality attribute model. Some of the necessary information comes from the current architectural design and some comes from the stimulus of the concrete scenario for which we are trying to determine satisfaction. We call such a mapping function an ‘interpretation’. Observe that multiple different architectures might map into the same quality attribute model instance. For example, one variable in our sample RMA quality attribute model is execution time of tasks. Multiple different architectures might have the same execution time for their tasks and hence would map to the same quality attribute model instance.

The form of the quality attribute model determines the information from the architectural description that is necessary to provide the values for the model. Using the example above, again, the architectural description must include an identification of the individual tasks, the execution times that contributes to each task, and the period of each task. These may have to be derived from the architectural description. The execution time of a task, for example, may be divided among various computations included within the task and the interpretation must determine which computations are included in each task and the execution time of each computation.

4.4 Calculating the response measure

Figure 3 shows how the quality attribute model, solution, and interpretation act together to enable the calculation of the response measure for an architecture. Figure 3 also shows the stimulus that determines how the architecture will generate its response.

Elements of the architecture contribute values for the variables to partially construct the quality attribute model instance. The stimulus from the quality attribute requirement provides the remainder of the independent variables needed to complete the quality attribute model instance. The instance is solved for a response measure and this is compared to the response measure requirement derived from the concrete scenario to determine whether the requirement expressed in the scenario is satisfied.

4.5 Architectural transformations

Revisiting the sample quality attribute model we see that there are exactly three variables: the period of a task, the execution time of a task, and the priority of a task. If a particular task does not meet its latency requirement, within the model there are only three things that can be manipulated to reduce the latency – the three variables. Furthermore, given a fixed set of scheduling priorities, we know from the quality attribute model that reducing the execution time of a task or increasing the period of a task (not necessarily the task whose latency requirement is unmet) are the only options for reducing the latency of the task whose requirement is unmet.

Given a modification of a variable (say reducing execution time of a task) there are multiple architectural

---

**Fig. 3** A quality attribute requirement supplies both the response measure region and a stimulus for a quality attribute model

The model is an interpretation of an architecture and the stimulus. The model is then solved to determine the predicted response measure.
techniques that can be used to effect this reduction. Any of the computations involved in the task including algorithmic computation or overhead computations can be reduced. Similarly, there are multiple architectural techniques that will have the effect of increasing the period of a task.

This is the motivation for architectural tactics. An architectural tactic is a transformation of a software architecture where the result of the transformation changes a response measure for a quality attribute to be closer to a desired value (this definition differs slightly from that given in [6]).

An architectural tactic can change the structure of an architecture (e.g. generalise modules), can change one of the properties of an architecture (e.g. reduce computational overhead), can add new responsibilities that the architecture must support (e.g. introduce intermediary), or can add a new quality attribute requirement (e.g. introduction of a usability tactic will cause a performance requirement for feedback). Notice that a design rule that does not affect the architecture does not satisfy the definition.

We have developed lists of architectural tactics for the attributes of availability, modifiability, performance, security, testability, and usability – partially through an examination of existing models and partially through interviewing experts in the various quality attributes, see [6] for an explanation. The variables for the existing RMA and impact analysis reasoning frameworks quality attribute models are manipulated by some of the performance and modifiability architectural tactics. We expect that as additional reasoning frameworks are developed for any of the attributes, that the lists of architectural tactics will be refined and improved. The existing tactics for modifiability are given in Fig. 4 and for performance in Fig. 5. Each tactic may or may not be suitable for use in any particular context. We rely on the architect to choose the appropriate tactic depending on its context.

Figure 6 shows (again notionally) how a tactic affects the response measure.

A tactic transforms the architecture a in Fig. 6 (with response measure $L_a$) to the architecture $a'$ (with response measure $L_{a'}$) where it is known that $L_{a'}$ is closer to the acceptable response than $L_a$.

5 Sample problem

Now we introduce the sample problem that we use to explain the working of a reasoning framework. Figure 7 shows a set of responsibilities for our sample problem as interpreted from the manufacturer’s specification. The problem is to receive automotive sensor information and determine whether the sensors suggest either a problem...
with the environment being sensed or with the sensors themselves. If a problem exists, then the rest of the automobile is notified. We have removed any reference to the type of sensors to preserve proprietary information but this is a typical set of responsibilities for sensor based systems: receive input, determine the status of the sensor, examine the value to see if any special conditions can be detected and inform the remainder of the automobile if any such condition is detected or if the sensor is defective.

The original set of responsibilities is organised as a directed graph. As design proceeds, the responsibilities will be modified and divided. Also, new responsibilities may be added and existing ones deleted. A directed graph that represents the current state of the responsibilities must be maintained as a portion of any design process that utilises the reasoning framework.

For this paper, we will use four sample scenarios – one modifiability and three performance. This is, of course, four out of many.

- Modifiability
  1. Replace sensor without change to functionality of the system within 3 person-days.

- Performance
  1. Determine sensor status within 250 ms after receiving sensor input. Sensor input arrives every 500 ms.
  2. Determine differences from past readings within 1250 ms after receiving input. Input arrives every 1250 ms.
  3. Inform the rest of the system of any alerts within 350 ms after the arrival of alert status. Alert status arrives every 350 ms.

Table 1 gives the six parts of these scenarios.

We now make several observations about Fig. 7 with respect to Fig. 1 (our overall description of a reasoning framework).

1. Figure 7 is not an architectural description. Figure 7 represents data flow among a set of responsibilities. An initial architecture must be created either by the reasoning framework or by other means in order for the reasoning framework to have an initial architecture. Subsequent architectures are derived by transformations from the initial architecture so they do not cause a problem.

2. There is insufficient information in Fig. 7 for either of our sample reasoning frameworks. Impact analysis requires the cost of change of individual responsibilities in addition to the...
probability of propagation of a change from one responsibility to other responsibilities. Rate monotonic analysis requires execution time to be assigned to the responsibilities. Necessary but, as yet, undefined properties of responsibilities must be gathered in order for the reasoning framework to operate.

3. Each concrete scenario can be ‘scoped’ [10] with respect to the responsibilities it impacts. Each concrete scenario identifies two responsibilities – ‘recognise the stimulus’ and ‘perform the response’. The scope of a concrete scenario consists of the original responsibilities within which these responsibilities are encompassed plus any intervening responsibilities. Since both the initial responsibilities and the concrete requirements are forms of the requirements, this mapping is something that can be done during requirements gathering. As the responsibilities evolve during the design process, the scope must be evolved as well. The CBSP approach [11] advocates an approach to mapping requirements to its architectural realisation by explicitly identifying components and connectors within which a requirement is met. They do this to maintain traceability between requirements and the architecture. We achieve traceability as a side effect. It does not depend on knowledge of the totality of the emerging architecture but only on knowledge of the responsibilities embedded within the architecture to perform that mapping. Our use of scoping is for efficiency in choosing the appropriate tactics, not solely for traceability.

Now we examine the activities of the two reasoning frameworks we are using for our example, in turn.

5.1 Impact analysis reasoning framework
The impact analysis reasoning framework is described in detail in [2]. For our purposes here, it is sufficient to know that it estimates the cost of a modification to an architecture based on the cost of modification of a responsibility, the probability of propagation of a modification from one responsibility to its children in the directed responsibility graph, and the allocation of responsibilities to modules. The type of concrete scenarios it can reason about are those involving change and the cost of change.

For purposes of demonstration, the impact analysis reasoning framework will take as input an initial architecture in which each of the responsibilities in Fig. 7 is allocated to its own module. Furthermore, the properties that the impact analysis reasoning framework requires are given in Table 2. These properties are the cost of modifying...
a single responsibility and the probability of a modification to one responsibility propagating to another. Finally, the scope of the modifiability scenario is the two responsibilities receive input from sensor and determine sensor status. These are the two responsibilities that are immediately affected by the scenario of changing the sensor. The rows and columns of this table are the responsibilities of the example. The cells represent values that the reasoning framework needs in order to perform its analysis. The values in the cells must be provided externally to any automatic design method.

The cost of a modification is the sum of the cost of modifying the responsibilities directly affected plus the cost of modifying any adjacent responsibilities weighted by the probability of modification. The adjacent responsibilities may, in turn, propagate to their adjacent responsibilities, see [2] for details of the model used.

For the first modifiability scenario, the scope of the scenario includes the two responsibilities: ‘receive input from sensor’ and ‘determine sensor status’. A change to ‘receive input from sensor’ will propagate to ‘correct for environmental factors’ which, in turn, propagates to ‘identify differences’. Further propagation adds little to the cost.

A change to ‘determine sensor status’ will propagate to ‘inform the rest of the system’. We see that the cost of making the modification in the scenario is greater than 5.4 days. The inputs to the cost calculation are given in Table 1. The formula evaluated is $2 + 2 + 1 + 1 + 0.8 + 2 + 0.8 + 0.8 + 1$. This is greater than the three day requirement.

For this unsatisfied scenario, the output of the reasoning framework is a list of modifiability tactics that might be useful in improving the architecture. We demonstrate the action of applying the tactic abstract common services.

Figure 8 shows the transformation defined by ‘abstract common services’. The services common to both Responsibility A and Responsibility B are factored out from these responsibilities and become separate responsibilities in their own module.

Figure 9 shows the result of applying the transformation of Fig. 8 to the architecture derived from Fig. 7. The common service that is abstracted is that of receiving data from the sensor. This responsibility is implicit in the responsibility ‘determine sensor status’ in Fig. 7. The abstraction of the common service means that only one responsibility receives data. We renamed the new responsibility ‘receive data from sensor’, the old responsibility ‘receive data from sensor’ now becomes ‘convert input into internal syntax’. The application of this tactic, therefore, has two aspects. One is the abstracting of the service of receiving input and the other is the use of a single internal syntax for the remainder of the responsibilities.

The application of this tactic has several effects.

* The probability of a change to a sensor propagating throughout the system is reduced because the internal syntax will be the same for all sensors.
* The costs of modifying the responsibilities ‘convert input into internal syntax’ and ‘determine sensor status’ are reduced since the common services have been removed from them.
* The new responsibility ‘receive input from sensor’ must be assigned a cost of modification.

Table 3 shows the probabilities and costs of modifications of the three responsibilities that are different in Fig. 9 from Fig. 7. Table 3 also shows the change in the probability of propagation from ‘convert input into internal syntax’ to ‘correct for environmental factors’ and from ‘correct for environmental factors’ to ‘identify differences from past readings’. These probabilities are sharply reduced because of the use of a common internal syntax for all sensors. The probability of propagation from receive input from

---

**Fig. 8** The transformation of abstract common services

The services common to A and B are factored out of A and B and placed into a separate module.

**Fig. 9** Architecture as a result of applying the transformation ‘abstract common services’
sensor to convert input into internal syntax is still quite high since the conversion into the internal syntax will depend on the syntax of each sensor. In this case, the cost of changing a sensor evaluates to slightly more than 2.5 person-days from evaluating the formula 
\[ \frac{0.2 * 1 + 0.8 * 0.8 * 0.2 * 0.2 * 1}{C_3} \]

The key observations from this example are:
- The quality attribute model together with the scoping information from a concrete scenario point to the responsibilities that are causing the concrete scenario to be unmet.
- Tactics provide the architect with a list of possible transformations involving these responsibilities.
- The architect must choose the tactic to be applied and understand its implications in terms of the properties of the architecture. The interpretation will automatically translate from the properties of the architecture to a new quality model instance.

5.2 Rate monotonic reasoning framework

The RMA reasoning framework can reason about concrete scenarios with hard deadlines and periodic arrivals. We demonstrate the rate monotonic reasoning framework beginning with the set of responsibilities embodied in Fig. 10. That is, we assume that the responsibilities given by the manufacturer have been modified based on the results of applying the transformations from the impact analysis reasoning framework. The necessary property for the RMA reasoning framework is predicted execution time for each responsibility and that is given in Table 4.

As before, the reasoning framework must begin with some architectural assumptions. RMA uses a concurrency view consisting of tasks with responsibilities assigned to tasks. The initial concurrency view is derived by assigning the responsibilities in each of our three performance concrete scenarios to a separate task. That is, each concrete scenario defines two responsibilities which is the scope of the scenario. These two responsibilities are allocated to a single task.

### Table 3: Costs of a modification and probability of propagation after abstracting common services

<table>
<thead>
<tr>
<th>Responsibility</th>
<th>Cost of modification</th>
<th>Receive input from sensor</th>
<th>Convert input into internal syntax</th>
<th>Correct for environmental factors</th>
<th>Determine sensor status</th>
<th>Identify differences from past readings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Receive input from sensor</td>
<td>1 day</td>
<td>N/A</td>
<td>0.8</td>
<td>N/A</td>
<td>0.5</td>
<td></td>
</tr>
<tr>
<td>Convert input into internal syntax</td>
<td>1 day</td>
<td></td>
<td></td>
<td>0.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Correct for environmental factors</td>
<td>1 day</td>
<td></td>
<td></td>
<td>0.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Determine sensor status</td>
<td>1 day</td>
<td>0.5</td>
<td></td>
<td>N/A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Identify differences from past readings</td>
<td>1 day</td>
<td>0.5</td>
<td></td>
<td>0.2</td>
<td></td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Table 4: Execution time for each responsibility

<table>
<thead>
<tr>
<th>Responsibility</th>
<th>Predicted execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Receive input from sensor</td>
<td>50 ms</td>
</tr>
<tr>
<td>Convert to internal syntax</td>
<td>50 ms</td>
</tr>
<tr>
<td>Correct for environmental factors</td>
<td>200 ms</td>
</tr>
<tr>
<td>Identify differences from past readings</td>
<td>50 ms</td>
</tr>
<tr>
<td>Determine sensor status</td>
<td>100 ms</td>
</tr>
<tr>
<td>Determine warnings of type 1</td>
<td>225 ms</td>
</tr>
<tr>
<td>Determine warnings of type 2</td>
<td>225 ms</td>
</tr>
<tr>
<td>Inform rest of system of any alerts</td>
<td>300 ms</td>
</tr>
</tbody>
</table>

### Fig. 10 Concurrency view of architecture with tasks identified and assigned initial priorities

Scheduling is done with lowest priority first.
The initial priority assignment is based on a deadline monotonic strategy. That is, priorities are ordered based on their deadlines. Scheduling is performed with lowest priority first.

This initial architecture results in scenarios 2 and 3 having an infinite latency – that is, they are not satisfied. The tactic ‘reduce execution time’ is applied for the responsibilities ‘correct for environmental factors’, ‘inform the rest of the system of any alerts’, and ‘convert input to internal syntax’. The value to which the execution times are reduced reflects a judgment about what can be achieved with a more careful consideration of the algorithms used. The reasoning framework enabled the architect to focus on the most critical portions of the system. The execution time of the other responsibilities does not, as yet, impact any architectural decisions.

Reducing the execution time did not enable the achievement of the scenarios so the following additional tactics were applied.

- Lengthen the period and deadline for scenario 2.
- Add logical concurrency to the task associated with scenario 1 by dividing the task into two sub-tasks each with individual deadlines.
- Reduce sampling frequency and lengthen deadline for the subtask of ‘determine sensor status’.

Note that applying some of these tactics requires modifying the quality attribute requirements. This has an impact beyond just architecture design and is a matter that is out of the scope of the reasoning framework. The reasoning framework suggests this as an option to satisfy some approximation of the scenarios. The decision as to whether this is reasonable and can be done is a matter for the stakeholders to decide.

After these modifications the new deadline scenarios are:

- **Determine sensor status** with a period of 600 ms and deadlines of 250 ms for the responsibility of ‘convert to internal syntax’ and 600 ms for the responsibility of ‘determine sensor status’.
- **Calculate differences from past readings** with a period of 2500 ms and a deadline of 2500 ms.
- **Inform rest of system of any alerts** with a period of 350 ms and a deadline of 350 ms.

The execution times of the various responsibilities have been adjusted to those in Table 5.

The revised concurrency architecture and the predicted latencies are given in Fig. 11.

### Table 5: Revised execution times

<table>
<thead>
<tr>
<th>Responsibility</th>
<th>Original Predicted execution time</th>
<th>Revised Predicted execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Receive input from sensor</td>
<td>50 ms</td>
<td>50 ms</td>
</tr>
<tr>
<td>Convert to internal syntax</td>
<td>50 ms</td>
<td>30 ms</td>
</tr>
<tr>
<td>Correct for environmental factors</td>
<td>200 ms</td>
<td>100 ms</td>
</tr>
<tr>
<td>Identify differences from past readings</td>
<td>50 ms</td>
<td>50 ms</td>
</tr>
<tr>
<td>Determine sensor status</td>
<td>100 ms</td>
<td>100 ms</td>
</tr>
<tr>
<td>Determine warnings of type 1</td>
<td>225 ms</td>
<td>225 ms</td>
</tr>
<tr>
<td>Determine warnings of type 2</td>
<td>225 ms</td>
<td>225 ms</td>
</tr>
<tr>
<td>Inform rest of system of any alerts</td>
<td>300 ms</td>
<td>225 ms</td>
</tr>
</tbody>
</table>

**Fig. 11**  Revised concurrency view with predicted latencies

6 **Trade-offs**

A trade-off is defined as ‘an exchange of one thing in return for another, especially relinquishment of one benefit or advantage for another regarded as more desirable’ [Note 1].

---

Note 1: From http://www.dictionary.com
In design, trade-off generally refers to an exchange among quality attributes; ‘I traded off performance for security’. Yet in the context of a set of quality attribute requirements as we have formulated them, either the requirements can be met or they cannot. If a quality requirement is met, it does not matter whether it is met easily or with difficulty. That is, if the latency requirement for a particular stimulus is for the response to be less than 2 ms a 1.9 ms response is as good as 0.1 ms response. The impact of a less stringent formulation on the theory we are presenting here is the subject of future work.

A design is a point in a design space and if one quality attribute requirement is not met, the real question is ‘have all of the designs in the design space been examined to determine whether there is a design that meets all of the requirements?” If the answer is ‘yes, all of the designs have been examined’ then one or more of the requirements must be weakened. There is no other alternative. If the answer is ‘no, there are more designs to examine’ then the action is to examine other designs. In neither case does the concept of trade-off apply to the design process as long as the quality attribute requirements are specified precisely. This is a fundamental distinction between our approach and that of [3].

In this context, a tactic defines a transformation that will help meet a requirement that is not currently being met. That is, a tactic is a method for exploring the design space. The concern is that if a tactic is applied then a quality attribute requirement which is currently met may become unmet. For this reason we do not discuss trade-offs among reasoning frameworks but interactions among reasoning frameworks. Understanding these interactions helps us to more efficiently explore the design space.

The NFR framework [3] distinguishes five types of trade-offs: Makes, Helps, Hurts, Breaks, or Unknown. The first four prejudge what impact particular actions have on the quality attributes and the last gives no guidance as to what action to take. We do not prejudge the impact. Instead, we identify interactions and then allow the reasoning framework, depending on context, to determine the impact of a particular action.

There are four different types of interactions possible between reasoning framework A and reasoning framework B.

1. A tactic from reasoning framework A affects responsibilities that have properties used by reasoning framework B. The application of a tactic may create a new responsibility, may divide an existing responsibility, may factor several existing responsibilities, or may delete a responsibility. Responsibilities are used in every reasoning framework and the properties of the responsibilities are the carriers of properties used by the reasoning frameworks’ quality attribute model. If the proposed tactic creates a new responsibility then reasoning framework B is affected. If the proposed tactic deletes a responsibility, then if a property of the deleted responsibility is being used by reasoning framework B, reasoning framework B is affected. If the proposed tactic divides or factors existing responsibilities, then reasoning framework B is affected only if one of the divided or factored responsibilities has a property used by reasoning framework B.

2. A tactic from reasoning framework A may generate a new quality attribute requirement of a type reasoned about by reasoning framework B. For example, one availability tactic is to replicate state. Replication of state adds a new responsibility to synchronise the states and it also generates a new quality attribute requirement that the synchronisation of the states happens with a particular deadline. The addition of the new responsibility we discussed in (1.) but the timing requirement for the synchronisation is a new requirement that must be considered by the appropriate reasoning frameworks.

3. A tactic from reasoning framework A may affect a variable of reasoning framework B’s quality attribute model. In the example given in (2.) of replicating state, not only do the states need to be synchronised but messages must be generated carrying the information necessary to synchronise the state. The rate and size of messages generated is of interest to a performance reasoning framework and is modelled by the variables of the performance reasoning framework. Thus, the application of a tactic from a reasoning framework concerned about availability affects the variables of a reasoning framework concerned about performance.

4. The response measure of reasoning framework A is the same as the response measure of reasoning framework B. Both RMA and a reasoning framework based on having a cyclic executive reason about deadline scenarios. In this case, the two reasoning frameworks are competing and one reasoning framework should be chosen for use on the particular system being designed.

The identification of these interactions can be done without knowing anything about the reasoning framework. Each reasoning framework can identify its variables and its response measure. This allows the design method to manage the interactions without requiring knowledge of the internals of the reasoning framework to be visible to the design method.

7 Use of reasoning frameworks within ADD

There are three problems of scale associated with reasoning frameworks:

1. The number of codified reasoning frameworks is small. Currently we have codified the two reasoning frameworks we have discussed here and one more is in process. There are a large number of quality attribute modelling theories suitable for turning into a reasoning framework but there is work involved in going from a quality attribute theory to a quality attribute reasoning framework. We expect the number of reasoning frameworks to grow but currently we only have two.

2. Each reasoning framework could go deeper. The RMA framework we have codified is based on concurrent pipelines that do not synchronise. The RMA theory supports synchronisation, we just have not yet codified it into a reasoning framework.

3. The solvers for any particular reasoning framework may not scale well with respect to the number of elements in the architecture. We mentioned that solvers for queuing networks may grow exponentially with the number of nodes.

For these reasons, we are experimenting with the use of reasoning frameworks within the context of ADD. In fact, we have applied the reasoning frameworks discussed here to an actual commercial problem (the design of a seat controller) with very good results.

We modify steps 2b and 2c of ADD (Choose architectural patterns and tactics that satisfy the architecture drivers, and Allocate remaining functionality) to be Choose architectural patterns and tactics and allocate functionality to satisfy the architectural drivers with the assistance of codified reasoning frameworks. This has the following advantages:

- There are a limited number of architectural drivers (concrete scenarios), usually around six. The number of elements of an architecture that supports six scenarios is
going to be small and so the problem of scale associated with the size of the architecture will not occur.

- The scale problem of only having codified a limited number of reasoning frameworks is managed through the reasoning framework’s treatment of constraints. For any architectural driver for which there is not a reasoning framework, the desired results are derived by the architect using intuitive means and expressed to the reasoning frameworks as a constraint. The reasoning frameworks will treat constraints as unchangeable and, consequently, the final pattern will reflect the intuition of the architect.

- The managing of interactions among the reasoning frameworks is done by the architect. Since there are a small number of architectural drivers, keeping track of the interactions of the reasoning frameworks and avoiding thrashing among the reasoning frameworks is accomplished by the architect’s intuition and judgment.

- The problem of the limited depth of the reasoning frameworks is managed by choice of problems. Problems that expose the reasoning frameworks to circumstances for which they are not equipped are recognised by the architect and not solved using reasoning frameworks.

None of these solutions is ideal and we are working on techniques to improve them but they are a method for enabling us to gain experience with the use of reasoning frameworks on actual problems without first having to solve all of the problems of scale.

8 Future work and conclusions

We have described a set of concepts intended to support a formal design process. In particular our definition of the elements of a reasoning framework makes explicit the distinction between architectural models and quality attribute models and the requirement for a mapping between them.

Some of the important points of our approach are:

- All quality attributes are treated in the same terms. There is a set of concepts that each reasoning framework must include and the output of each reasoning framework is a transformation. This enables a method that uses quality attribute reasoning frameworks to treat them in a ‘plug and play’ fashion.

- A formal approach does not support designing for non-explicit requirements. Normal practice is to separate and encapsulate most identifiable large blocks of responsibility. Use of a reasoning framework for modifiability only supports separating and encapsulating those blocks of responsibility that are explicitly included in quality attribute requirements. Similarly for performance requirements. This places a burden on the ensuring that the quality attribute requirements are complete.

- We have begun to identify what is meant by interaction among the reasoning frameworks. The use of trade-offs is not appropriate when using a formal approach such as we are advocating here.

- The reasoning frameworks require a level of formality in specification of variables that is not always natural to an architect. This suggests that their use requires tool support.

We have, in fact, produced a prototype system named ArchE (Architecture Expert) to support our extensions to ADD. Space precludes going into any detail about ArchE here although it was ArchE that was used in the commercial design effort we alluded to earlier.

One item that we plan to focus on in the immediate future is the managing of trade-offs and how to more systematically manage the interactions among the reasoning frameworks. Within this context, we also plan to investigate the impact of using a less stringent form of specifying quality attribute requirements that allows for differentiation between different values of the response measure. This will, of course, introduce substantial new functionality to our system.

In conclusion, although the use of reasoning frameworks as an adjunct to a design method is not yet ready for industrial strength use, we believe that we have demonstrated that reliance on quality attribute specific reasoning frameworks provide a means for generating a design and holds promise for the ability to scale to be industrial strength. The uses of reasoning frameworks have the qualities that we asserted in the introduction were necessary for any formal method that supports the achievement of quality attribute requirements.

9 Acknowledgments

The authors would like to thank the referees for this paper whose comments resulted in substantial improvements to it. Also, during the revision of this paper, L. Bass was a Visiting Researcher at the National Institute of Computer Technology, Australia (NICITA). He thanks NICITA for hosting him. This work was supported by the U.S. Department of Defense and Robert Bosch, GmbH

10 References