Research Report 14/97

Software Architecture Design: Evaluation and Transformation

by

Jan Bosch and Peter Molin

Department of
Computer Science and Business Administration
University of Karlskrona/Ronneby
S-372 25 Ronneby
Sweden

ISSN 1103-1581
ISRN HK/R-RES—97/14—SE

Software Architecture Design: Evaluation and Transformation

by Jan Bosch and Peter Molin

ISSN 1103-1581
ISRN HK/R-RES—97/14—SE

Printed by Psilander Grafiska, Karlskrona 1997

Software Architecture Design: Evaluation and Transformation

Jan Bosch & Peter Molin
University of Karlskrona/Ronneby
Department of Computer Science
Soft Center, S-372 25
Ronneby, Sweden
+46-457-78726
[Jan.Bosch|Peter.Molin]@ide.hk-r.se

ABSTRACT
Since the architecture of a software system constrains the non-functional requirements, the decisions taken during architectural design have a large impact in the resulting system. An architectural design method is presented that employs iterative evaluation and transformation of the software architecture in order to satisfy the non-functional requirements (NFRs). Architecture evaluation is performed by using scenarios, simulation, mathematical modelling and reasoning. The architecture can be transformed by imposing an architectural style, imposing an architectural pattern, using a design pattern, converting an NFR to functionality and by distributing NFRs. The method has, in various forms, been applied in several industrial projects.

Keywords
Software architecture design, non-functional requirements, evaluation, scenarios

INTRODUCTION
Perhaps the most complex activity during application development is the transformation of a requirement specification into an application architecture. The other phases also are challenging activities, but the latter are better understood and more methodological and technological support is available to the software engineer. The process of architectural design is less formalised and often is more like intuitive craftsmanship than rational engineering.

The domain of software architecture has received considerable attention during recent years. This is, among others, because especially non-functional requirements (NFRs) are heavily influenced by the architecture of the system. For example, in an architectural design that we evaluated there was a central entity in the system that played a role in most computations. This bottleneck entity caused theoretical and practical upper limits on the performance and reliability of the system. Architectural design is a typical multiple objective design activity where the software engineer has to balance the various requirements during architectural design.

In this paper, an architecture design method is presented that provides support for a rational design process balancing and optimising, especially, the non-functional requirements. The method iteratively assesses the degree up to which the architecture supports each NFR and improves the architecture using transformations until all NFRs are fulfilled. The presented method complements traditional design methods in that it focuses on NFRs whereas traditionally the functionality is prioritised.

The remainder of this paper is organised as follows: In the next section requirements are discussed and in the following section the example architecture is presented that is used throughout this paper. Then, an overview of the method is presented and it is followed by a section describing the assessment of NFRs. Subsequently, types of architecture transformation are presented and the paper is concluded by a discussion of related work and a conclusion.

REQUIREMENTS
Requirement engineering has been studied extensively and it is not our aim to address the process of identifying and specifying requirements. Instead the requirement specification is used as an input for architectural design. However, we establish a terminology for the various requirement concepts.

System requirements are defined as the top level requirement set consisting of software, hardware and mechanical requirements. In this paper, we focus on fulfilling the software requirements and ignore other types of requirements. Software requirements can be defined as consisting of *functional requirements* and *non-functional requirements* (also referred to as quality

attributes or system properties). The functional requirements are related to the domain related functionality of the application. Typically, a functional requirement is implemented by a subsystem or a group of components in the software. Non-functional requirements can be categorised in *development* NFRs and *operational* NFRs. Development NFRs are qualities of the system that are relevant from a software engineering perspective, e.g. maintainability, reusability, flexibility and demonstrability. Operational NFRs are qualities of the system in operation, e.g. performance, reliability, robustness and fault-tolerance. Different from functional requirements, non-functional requirements can generally not be pinpointed to a particular part of the application but are a property of the application as a whole.

EXAMPLE

The examples used in this paper are based on experiences from a fire alarm system [15]. Although a fire alarm system consists of, among others, several types of sensors, communication devices, presentation devices, alarm bells, user interfaces, we start with an identification of the simplest functional behaviour of an abstracted system.

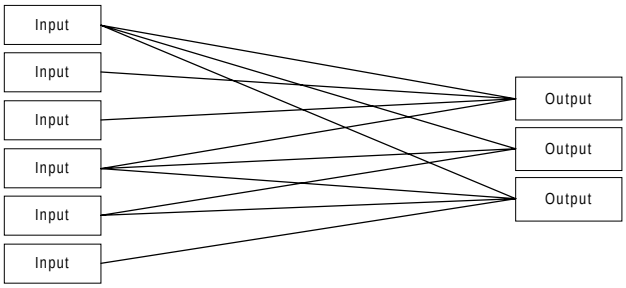

Figure 1. *Functional view of the fire alarm system*

The abstract description consists of a number of inputs and a number of outputs representing sensors and indicators. They are related by some kind of behaviour function possibly implemented as part of the outputs. In this simple view the behaviour of outputs depends on the state of all inputs. Important NFR of the fire alarm system are worst case response time for alarm conditions, high availability and reliability. Efficiency is also important since the systems are price-sensitive.

METHOD OVERVIEW

In software industry, our experience is that non-functional requirements are generally dealt with by a rather informal process during architecture design. Once the system is implemented, tests are performed to determine whether the NFRs are fulfilled. If not, parts of the system are redesigned. Since the system architects often are experienced in building systems in the domain, experience helps them to minimise system redesign. Computer science and software engineering research, on the other hand, has spent considerable effort on several of the non-functional requirements, e.g. the real-time systems research community has put much focus on hard real-time systems and the object-oriented research community has studied software reuse. The problem is, we believe, that each research community only studies a single non-functional system requirement and, consequently, not addresses the composition of its solutions with the solutions proposed by research communities studying different non-functional requirements. However, real systems never have only a single non-functional requirement to fulfil, but generally have to achieve multiple of these requirements. For instance, most real-time systems should be reusable and maintainable to achieve cost-effective operation and usage, whereas fault-tolerant systems also need to fulfil other requirements such as timeliness and maintainability. No pure real-time, fault-tolerant, high-performance or reusable computing systems exist, even though most research literature tends to present systems as being such archetypical entities. All realistic, practical computing systems have to fulfil multiple NFRs. However, constructing such systems is hard because the NFRs tend to be conflicting. Reusability and performance are generally considered to be contradicting, as well as fault-tolerance and real-time computing, to name a few.

Conventional design methods, e.g. [2,9,20], tend to focus on achieving the required system functionality, but do not spend much attention on non-functional requirements. The various NFR-based research communities identified this as unsatisfactory, and have proposed their own design methods for developing real-time [13], high-performance [25] and reusable systems, respectively. However, all these methods focus on a single NFR and treat all other NFRs as having secondary importance, if at all. We consider these approaches unsatisfactory since the software engineer needs to balance the various NFRs for any realistic system. However, lacking a supporting method, software engineers in industry design system architectures in an ad-hoc, intuitive, experience-based manor, with the associated risk of unfulfilled system properties.

Method

The context for the method consists of a requirements specification that is taken as an input to the method and an architectural design generated as output. The design is used for the subsequent phases. In figure 2, the steps in the method are presented graphically. The process starts with an architectural design based on the functionality specified in the requirements specification. Although

software engineers generally will not design a system less reliable or reusable, the non-functional requirements are not explicitly addressed at this stage. The result is a first version of the application architecture design. This design is evaluated with respect to the non-functional requirements. Each NFR is given an estimate in using a qualitative or quantitative assessment technique. The estimated NFR values are compared to the values in the requirements specification. If all estimations are as good or better than required, the architectural design process is finished. Otherwise, the second stage is entered: architecture transformation. During this stage, the architecture is improved by selecting appropriate NFR-optimising transformation. Each set of transformations (one or more) results in a new version of the architectural design that is fed back to the earlier stage in the process. This design is again evaluated and the same process is repeated, if necessary, until all NFR are fulfilled or until the software engineer decides that no feasible solution exists.

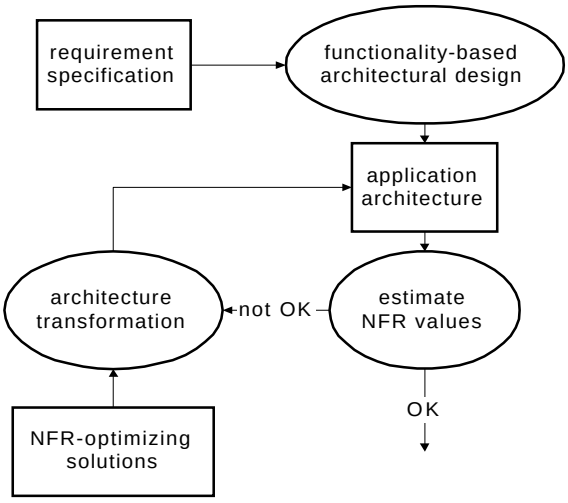

Figure 2. *Outline of an architectural design method*

The fact that the method is iterative is not novel. Some design methods for one-NFR based systems, e.g. real-time or performance engineering, follow a similar iterative process. For instance, [25] defines a similar method for performance engineering but Smith only considers performance.

Experiences

The architectural design method has been applied, in various forms, in three projects, i.e. for fire-alarm systems [15], measurement systems [4] and dialysis systems. Our experiences are that the method does not constrain the creative process during the initial design stages, but, on the other hand, explicitly supports the software engineers.

In the following sections, the individual stages of the method are described in more detail and examples are presented.

FUNCTIONALITY-BASED ARCHITECTURE DESIGN

Based on the requirement specification, the top-level, i.e. architecture, design of the system is performed. The main issue during this phase is to identify the core abstractions based on which the system is structured. Although these abstractions are modelled as objects, our experience is that these objects are not found immediately in the application domain. Instead, they are the result of a creative design process that, after analysing the various domain entities, abstracts the most relevant properties and models them as architecture entities. Once the abstractions are identified, the interactions between them are defined in more detail.

The process of identifying the entities that make up the architecture is different from, for instance, traditional object-oriented design methods [2]. Those methods start by modelling the entities present in the domain and organise these in inheritance hierarchies, i.e. a bottom-up approach. Our experience is that during architectural design it is not feasible to start bottom-up since that would require dealing with the details of the system. Instead one needs to work top-down.

Architecture entity identification is related to domain analysis methods. However, different from these approaches, our experience is that the entities resulting from architecture design are generally not found in the domain. For instance, the architecture of the dialysis system consists of two primary entities, i.e. *devices* and *controllers*. A concrete dialysis machine, however, consists of sensors, such as temperature and concentrate sensors, and actuators, such as heaters and pumps. The identified abstractions are not related in a straightforward way to the concrete entities in a dialysis system. A second difference between architecture design and domain analysis is that the architecture of a system generally covers multiple domains.

ASSESSING NON-FUNCTIONAL REQUIREMENTS

One of the core features of the architectural design method is that the non-functional properties of a system or application architecture are explicitly evaluated during architecture design; thus without having a concrete system available. The traditional approach in software industry is to implement the system and then measure the actual values for the non-functional system properties. The obvious disadvantage is that potentially large

amounts of resources have been put on building a system that does not fulfil its non-functional requirements. In the history of software engineering, several examples of such systems can be found. Being able to estimate the non-functional properties of the system already during early development stages is extremely important to avoid such mishaps.

However, how does one measure system properties based on an abstract specification such as an architectural design? One has to identify that it is not possible to measure the NFRs for the final system based on the architecture design. Instead, the goal is to evaluate the potential of the designed architecture to reach the required levels for the NFRs. For example, some architectural styles, e.g. layered architectures, are less suitable for systems where performance is a major issue, even though the flexibility of this style is high.

Four different approaches for assessing non-functional requirements have been identified, i.e. scenarios, simulation, mathematical modelling and objective reasoning. For each NFR, the engineer can select the most suitable approach for evaluation. In the subsequent sections, each approach is described in more detail.

Scenario-based evaluation

To assess a particular NFR, a set of scenarios is developed that concretises the actual meaning of the non-functional requirement. For instance, the maintainability requirement may be specified by scenarios that capture typical changes in requirements, underlying hardware, etc. The scenarios can then be used to evaluate the number of changes required to adapt the architecture to the new situation. Another example is robustness where the architecture can be evaluated with respect to the effects of invalid input.

The effectiveness of the scenario-based approach is largely dependent on the representativeness of the scenarios. If the scenarios form accurate samples, the evaluation will also provide an accurate result. Object-oriented design methods use scenarios to specify the intended system behaviour, e.g. use-cases [9] and scenarios [27]. For architectural design, however, two sets of scenarios should be developed, i.e. one for design and one for evaluation purposes. Once a version of the architecture is ready for evaluation, the software engineer can 'run' the scenarios for the architecture and evaluate the results. For instance, if most of the change scenarios require considerable reorganisations of the architecture, one can conclude that the maintainability of the architecture is low.

The proposed use of scenarios is analogous to statistical testing of code: based on the ratio between successful and failed scenarios, one can estimate the studied NFR. The main difference is that running a test case on a code module generally leads a clear result, whereas the result of, e.g. maintenance, scenarios often are more difficult to interpret. The quality of the assessment is strongly related to the objective judgement of the engineer 'running' the scenarios.

It should be noted that it is necessary to use a different scenario set for evaluation than for architecture design. The set used for design is generally supported by the architecture. However, while developing the scenarios, it is not necessary to develop two sets. The two sets could be generated later by randomly dividing the developed set of scenarios. In addition, depending on the system, it might be necessary to develop new scenarios for evaluation purposes if the design is iterated a number of times.

In our experience, scenario-based assessment is particularly useful for development NFRs. NFRs such as maintainability can be expressed very naturally through change scenarios.

In the fire alarm system example, several maintenance scenarios have been defined for architecture evaluation. Three, rather simple, examples are:

- The introduction of a new detector. Detector technology is continuously evolving where new detector types are introduced on the market every year. They differ not only with respect to what they measure or how, but also how they interface to a fire alarm system.
- The introduction of a new fire extinguisher system.
- The introduction of a new hardware platform. To fulfil market expectations a family of systems is offered to the customer. The architecture should be easily adopted to new hardware architectures of different sizes and complexity.

The evaluation is based on an estimation of the effort required for an adaptation and the number of affected architectural entities.

Simulation

Simulation of the architecture using an implementation of the application architecture provides a second approach for estimating NFRs. The main components of the architecture are implemented and other components are simulated resulting in an executable system. The context, in which the system is supposed to execute in, could also be simulated at a suitable abstraction level. This implementation can then be used for simulating

application behaviour under various circumstances.

Simulation of the architecture design is, obviously, not only useful for NFR assessment, but also for evaluating the functional aspects of the design. Building a simulation requires the engineer to define the behaviour and interactions of the architecture entities very precise which may uncover inconsistencies in the design.

Once a simulation is available, one can execute execution sequences to assess NFRs. Robustness, for example, can be evaluated by generating or simulating faulty input to the system or by inserting faults in the connections between architecture entities.

Simulation complements the scenario-based approach in that simulation is particularly useful for evaluating operational NFRs, such as performance of fault-tolerance by actually executing the architecture implementation, whereas scenarios are more suited for evaluating development NFRs, such as maintainability and flexibility. Nevertheless, the implementation of the architecture in the simulation can be used to evaluate, for instance, maintainability, by changing the implementation according to change scenarios and measuring the required effort.

How can simulation be effectuated in the fire alarm systems example? The first step is to define an interface to physical sensors and indicators through a layer of communication software as shown in figure 3.

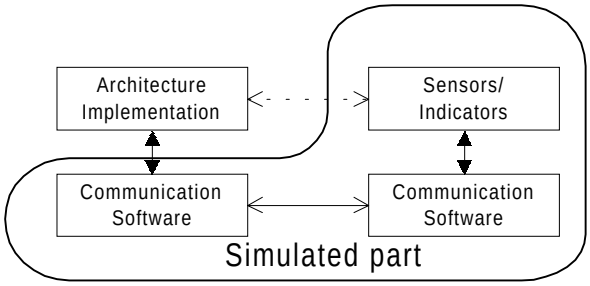

Figure 3. *Simulation of a system*

Based on the interface between the top level architecture implementation and the communication software both communication and sensor behaviour could be simulated. The evaluation of robustness could for example be based on the definition of a stochastic model of unusual conditions. A simple Markov model is shown in figure 4. Scenarios could be derived from this model. The architecture can thus be evaluated with respect to robustness before the communication software is designed or implemented. The accuracy of the evaluation depends of course on how well the model reflects real world conditions, but the results will at least pinpoint possible architectural design flaws.

Mathematical modelling

Various research communities, e.g. high-performance computing [25], reliable systems, real-time systems [13], etc., have developed mathematical models that can be used to evaluate especially operational NFRs. Different from the other approaches, the mathematical models allow for static evaluation of architectural design models. For example, performance modelling is used while engineering high-performance computing systems to evaluate different application structures in order to maximise throughput.

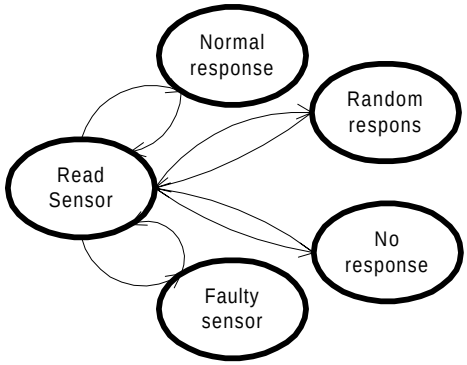

Figure 4. *Markov model of sensors and communication*

Mathematical modelling is an alternative to simulation since both approaches are primarily suitable for assessing operational NFRs. However, the approaches can also be combined. For instance, performance modelling can be used to estimate the computational requirements of the individual components in the architecture. These results can then be used in the simulation to estimate the computational requirements of different execution sequences in the architecture.

An example of mathematical modelling in the fire alarm system is related to one of the fundamental NFR in the fire alarm system, worst case alarm response time should not exceed 3 s. Assume that we have a model as in figure 5 and that a polling system is used where each input is evaluated periodically.

The evaluation of one sensor involves sending a message to the physical sensor using underlying communication software, waiting for the response, receiving the response and evaluate it for possible alarm conditions.

In this very simple performance model we assume that *Send request* requires 2 ms, *Receive answer* requires 2 ms, *Evaluation* requires 5 ms and the actual communication takes 8 ms. If we assume that we have 200 sensors to evaluate and that we use round-robin polling, we will have a worst case response time of

roughly 200*(2+2+5+8) ms = 3,4 s. Even such simple modelling gives valuable insights regarding system performance.

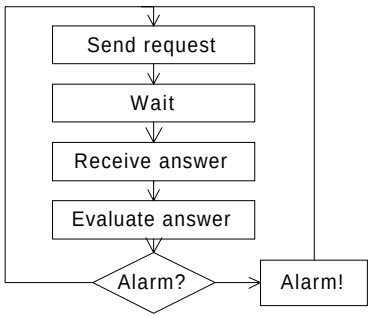

Figure 5. *Simple model of reading sensor values*

Objective Reasoning

A fourth approach to assessing NFRs is through reasoning based on logical arguments. Experienced software engineers often have valuable insights that may prove extremely helpful in avoiding bad design decisions. Although some part of these experiences are based on anecdotal evidence, most can often be justified by a logical line of reasoning.

This approach is different from the other approaches in that the evaluation process is less explicit and more based on subjective factors as intuition and experience. The value of this approach should, however, not be underestimated. Most software architects we have worked with had well-developed intuitions about 'good' and 'bad' designs. Their analysis of problems often started with the 'feeling' that something was wrong. Based on that, an objective argumentation was constructed either based on of the aforementioned approaches or on logical reasoning. In addition, this approach may form the basis for the other evaluation approaches. For example, an experienced software engineer may identify a maintainability problem in the architecture and, to convince others, define a number of scenarios that illustrate this.

Since the fire alarm system is inherently concurrent, it is necessary to choose a concurrency model. One line of argumentation is based on the fact that fine-grain concurrency with a pre-emptive scheduler could be error-prone considering the possibility of race conditions. This line of reasoning may lead to an architectural transformation eliminating the need of such fine-grain concurrency.

ARCHITECTURE TRANSFORMATION

Once the architecture properties have been assessed, the estimated values are compared to the requirements specification. If one or more of the NFRs are not satisfied, the architecture has to be changed to cover these non-functional requirements also. This requires the software engineer to analyse the architecture and to decide due to what cause the property of the architecture is inhibited. Often, the evaluation itself generates hints as to what parts or underlying principles cause low scores.

Assessment of the NFRs is performed assuming a certain context, consisting of, certain subsystems, e.g. databases or GUI systems, an operating system and hardware. Whenever a NFR is not fulfilled, one may decide to either make changes to the presumed context of the system architecture or to make changes to the architecture itself. In the architectural design method discussed in this paper, changes to the architecture are performed as *architecture transformations*. Each transformation leads to a new version of the architecture that has the same functionality, but different values for its properties.

The consequence of architecture transformations is that most transformations affect more than one property of the architecture; generally some properties positively and others in a negative way. For instance, the *Strategy* design pattern increases the flexibility of a class with respect to exchanging one aspect of its behaviour. However, performance is often reduced since instances of the class have to invoke another object (the instance of the Strategy class) for certain parts of their behaviour. However, in this case, the positive effect of increased flexibility considerably outweighs the minor performance impact.

Five categories of architecture transformations have been identified, organised in decreasing impact on the architecture, i.e. imposing an architectural style, imposing an architectural pattern, applying a design pattern, converting NFRs to functionality and distributing requirements. One transformation does not necessarily solve a NFR completely. Two or more transformations might be necessary. In the sections below, each category is discussed in more detail.

Impose architectural style

Shaw and Garlan [23] and Buschmann et al. [5] present several architectural styles (or patterns) that improve the possibilities for certain NFRs for the system the style is imposed upon and are less supportive for other NFRs. Certain styles, e.g. layered architectural style, increase the flexibility of the system by defining several levels of abstraction, but generally decrease the performance of the resulting system. With each architectural style, a fitness for each system property is associated. The most

appropriate style for a system depends primarily on its non-functional requirements. Transforming an architecture by imposing an architectural style results in a complete reorganisation of the architecture.

Architectural styles can generally not be merged, but it is possible to apply a different style in a subsystem than at the system level, provided that the subsystem acts as a correct component at the system level. Thus when during design interation another architectural style is selected for a part of the system, it is necessary to roll-back and to remove the style that initially was imposed.

In our approach, we explicitly distinguish between the components that are used to fulfil the functional requirements and the software architecture of the system that is used to fulfil the non-functional requirements. In practice, the distinction is generally not as explicit, e.g. the implementation of a component influences some NFRs, e.g. reliability and robustness.

An example of an imposed architecture style is illustrated using the simple functional model in figure 1. An evaluation with respect to efficiency and performance will conclude that the proposed solution is inadequate since all *Outputs* need to examine the state of all *Inputs*. *Deviations* are introduced as a way of recording only those *Inputs* that are in a state different from normal. Every *Input* is responsible of creating a *Deviation* and store it in a common area. *Outputs* only need to investigate that area in order to establish its behaviour. The proposed solution, shown in figure 6, is an example of the *Blackboard* architectural style[23].

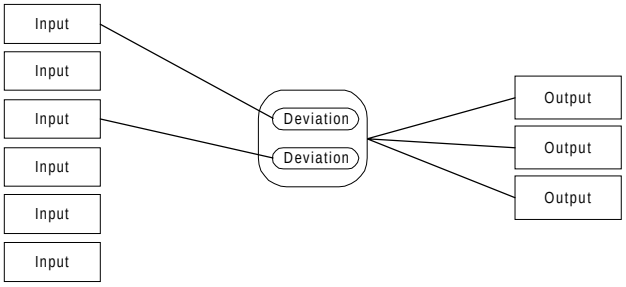

Figure 6. *Fire alarm system architecture based on the blackboard style*

Impose architectural pattern

A second category of transformations is the use of *architectural patterns*. An architectural pattern is different from an architectural style in that it is not pre-dominant. It is also different from a design pattern since it affects the complete architecture, or at least the larger part of it. Architectural patterns generally impose a *rule* [17] on the architecture that specifies how the system will deal with one aspect of its functionality, e.g. concurrency or persistence.

An example from the fire alarm system is related to concurrency. In the functional architecture in figure 1, it is assumed that reading of inputs and potentially generating corresponding outputs take place concurrently. Assuming that light-weight pre-emptive threads are used, this solution can be evaluated with respect to efficiency and reliability. The cost of threads and the fact that pre-emptive threads are error-prone since they could cause racing conditions when accessing shared data, necessitates investigation of other solutions. In this case, the *Periodic Object* pattern [15] is applied since it provides the appropriate granularity of concurrency. A Periodic Object is defined as an abstract object that is regularly activated by a scheduler which calls the objects Tick method. Concrete subclasses implement their own Tick method which defines one slice of the periodic execution of an active object. The degree of concurrency achieved by this solution depends on the “thinness” of the largest slice. The resulting constraint is that all such Tick methods must return within a pre-defined maximum time. This design pattern and design rule is an example of an architectural pattern that influences the complete architecture since all inputs and outputs are affected.

Another example of transforming the architecture could be by redefining the algorithm in figure 5 in order to handle the seemingly long worst case response time.

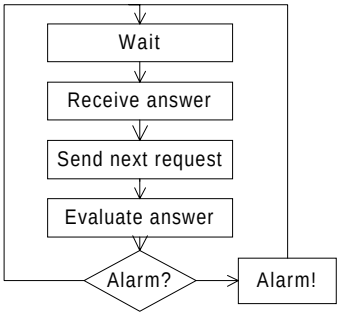

Figure 7. *A modified outline the reading sensor model*

This high-level transformation of the basic algorithm enforces coupling between Inputs, since it requires an interleaved operation where evaluation of data is performed at the same time as another physical sensor is operated upon. Consequently, the entire architecture is affected. However, applying the same performance model as previously the new worst case response time will be 200 * (2+2+8) ms = 2,4 s.

Apply design pattern

A less dramatic transformation is the application of a

design pattern on a part of the architecture. For instance, an abstract factory might be introduced to abstract the instantiation process for its clients. The abstract factory increases maintainability, flexibility and extensibility of the system since it encapsulates the actual class type(s) that are instantiated, but decreases the efficiency of creating new instances due to the additional computation, thereby reducing performance and predictability. Different from imposing an architectural style, causing the complete architecture to be reorganised, the application of a design pattern generally affects only a limited number of classes in the architecture. In addition, a class can be involved in multiple design patterns without creating inconsistencies.

In the fire alarm example, an evaluation of the change scenarios results in the conclusion that the hardware dependent parts of the inputs are susceptible for changes. The behavioural part is defined by product standards and is thus not related to the actual sensor type. Therefore, the *Point* pattern [15] is introduced where these two issues have been separated. The resulting architecture can be seen in Figure 8.

Convert NFRs to functionality

Another type of transformation is the conversion of a NFR into a functional solution that consequently extends the architecture with functionality not related to the problem domain but used to fulfil a NFR. Exception handling is a well-known example that adds functionality to a component to increase the fault-tolerance of the component.

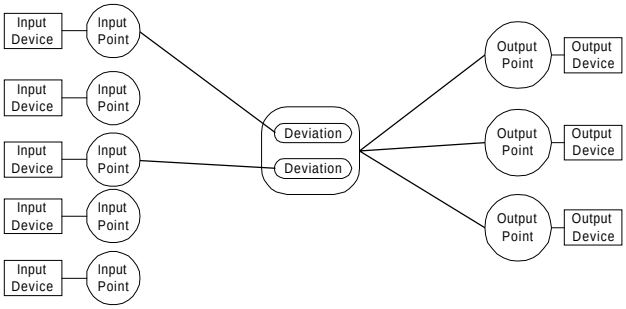

Figure 8. *The Point pattern improves the flexibility of the system*

In the example fire alarm system, there are NFRs related to self-monitoring and availability. In certain cases, detected faults should be handled using hardware redundancy, whereas in other cases problems should be indicated to the fire brigade or the persons responsible for system maintenance. These requirements are, at least, partially fulfilled by transforming them to functional requirements similar to the basic alarm requirements. The corresponding architecture extended with entities dealing with self-monitoring as can be seen in Figure 9.

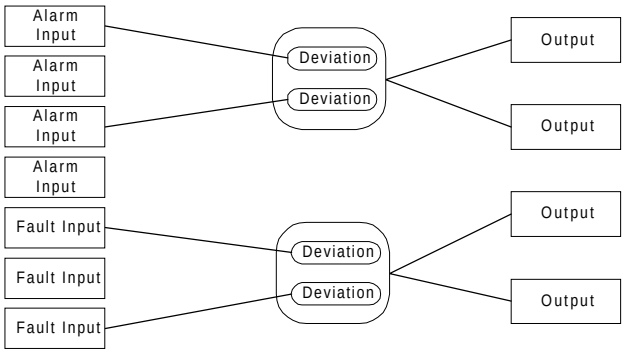

Figure 9. *Self-monitoring transformed to functional behaviour*

Distribute requirements

The final type of transformation deals with NFRs using the *divide-and-conquer* principle: a NFR at the system level is distributed to the subsystems or components that make up the system. Thus, a NFR X is distributed over the n components that make up the system by assigning a NFR x_i to each component c_i such that $X=x_1+ \ldots +x_n$. A second approach to distribute requirements is by dividing the NFR into two or more functionality-related NFRs. For example, in a distributed system, fault-tolerance can be divided into fault-tolerant computation and fault-tolerant communication.

Fire alarm systems are often implemented as a distributed system where one CPU-based system controls one building. Several such systems communicate with each other and the basic requirement is that an alarm detected on one system should be indicated on all other systems. This requirement can be achieved by enforcing a copy of the ”blackboard” to be available on all systems. This distribution can be effectuated by means of communication software operating at a lower layer and assuring that consistent copies of the blackboard are distributed throughout the system. The resulting architecture is shown in figure 10.

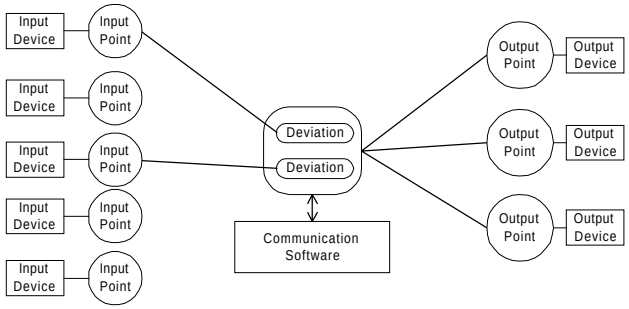

Figure 10. *Communication software implementing the distributional aspects of the fire alarm system*

The NFR stating how well the fire alarm system should cope with communication problems is assigned to the communication software, effectively distributing a system requirement to a system component.

RELATED WORK

The work presented in this paper is related to a number of research activities. Architectural design methods have been defined by Krüger [12] and Shlaer & Mellor [24].

Evaluating architectures has, among others, been studied by Kazman et al. [11] Their SAAM method only uses scenarios for evaluating NFRs, where we use, in addition to scenarios, simulation, mathematical modelling and reasoning. Research on *metrics* [7] is concerned with quantifying various aspects of software. However, most metrics approaches measure on implemented systems rather than on the results of earlier development phases. Simulation and testing at the architectural level is discussed by Richardson and Wolf[19].

Several research communities have developed design methods that focus on the NFR studied by that community. Object-oriented design methods [2,9,20] primarily aim at representing the functionality in the most reusable and maintainable manner. Real-time design methods have, among others, been proposed by [13], whereas designing high-performance computing systems is discussed by [25]. The method proposed in this paper differs from the these approaches in that it addresses the, more realistic, situation where a system has to fulfil a set of NFRs that need to be balanced.

The notion of implementing a system by iterative transformation from a formal specification to an executable specification has been a long-lived goal of formal methods, e.g. [10] and RAISE [18]. RAISE is a result of 20 years of research and experience in systematic software development using the paradigm of `stepwise refinement' [6,28,10]. These differ from the method discussed in this paper in that transformations are used to improve the architecture for certain NFRs rather than converting formally specified functionality. In addition, our transformations are not formally verified.

CONCLUSIONS

An architecture design method has been presented that explicitly addresses the non-functional requirements put on the architecture. It has been identified that the ability of a system to fulfil its non-functional requirements is, up to a considerable extent, restricted by its architecture.

The proposed method starts with a functionality-based design phase in which a software architecture is designed purely based on the functional requirements. The subsequent phases are the NFR evaluation phase and the architecture transformation phase. These phases are performed iteratively until all NFRs are fulfilled. NFRs can be assessed by four techniques, i.e. scenarios, simulation, mathematical modelling and objective reasoning. Five categories of architecture transformations are available, i.e. imposing architectural styles, imposing architectural patterns, using design patterns, converting NFRs and distributing NFRs into functionality.

The architectural design method has been applied, in some form, in the design of three systems, i.e. fire-alarm systems, measurement systems and dialysis systems. Experience shows that the method is provides appreciated support to the software engineers during architectural design.

REFERENCES

1. P. Binns, Matt Englehart, M. Jackson, S. Vestal, 'Domain-Specific Software Architectures for Guidance, Navigation and Control,' *Honeywell Technical Report*, 1994.
2. G. Booch, *Object-Oriented Analysis and Design with Applications* (2nd edition), Benjamin/Cummings Publishing Company, 1994.
3. B. Boehm, 'Aids for Identifying Conflicts Among Quality Requirements,' *International Conference on Requirements Engineering (ICRE96)*, Colorado, April 1996, and IEEE Software, March 1996.
4. J. Bosch, 'Design of an Object-Oriented Measurement System Framework,' *submitted*, 1997.
5. F. Buschmann, C. Jäkel, R. Meunier, H. Rohnert, M.Stahl, *Pattern-Oriented Software Architecture - A System of Patterns*, John Wiley & Sons, 1996.
6. E.W. Dijkstra, *A Discipline of Programming*, Prentice-Hall International, 1976.
7. N.E. Fenton, S.L. Pfleeger, *Software Metrics - A Rigorous & Practical Approach* (2nd edition), International Thomson Computer Press, 1996
8. D. Garlan, R. Allen, J. Ockerbloom, 'Exploiting Style in Architectural Design Environments,' *Proceedings of SIGSOFT '94 Symposium on the Foundations of Software Engineering*, December 1994.
9. I. Jacobson, M. Christerson, P. Jonsson, G. Övergaard, *Object-oriented software engineering. A use case approach*, Addison-Wesley, 1992.
10. C.B. Jones, *Systematic Software Development using VDM*, Prentice-Hall Series in Computer Science. Prentice-Hall International, 1986 .

11. R. Kazman, L. Bass, G. Abowd, M. Webb, 'SAAM: A Method for Analyzing the Properties of Software Architectures,' *Proceedings of the 16th International Conference on Software Engineering*, pp. 81-90, 1994.

12. P.B. Krüger, 'The 4+1 View Model of Architecture,' *IEEE Software*, pp. 42-50, November 1995.

13. J.W.S. Liu, R. Ha, 'Efficient Methods of Validating Timing Constraints,' in *Advanced in Real-Time Systems*, S.H. Son (ed.), Prentice Hall, pp. 199-223, 1995.

14. D.C. Luckham, J.J. Kenney, L.M. Augustin, J. Vera, D. Bryan, W. Mann, 'Specification and Analysis of System Architecture Using Rapide,' *IEEE Transactions on Software Engineering*, Special Issue on Software Architecture, 21(4):336-355, April 1995.

15. P. Molin, L. Ohlsson, 'Points & Deviations - A pattern language for fire alarm systems,' to be published in *Pattern Languages of Program Design 3*, Addison-Wesley.

16. W.F. Ogden, M. Sitaraman, B.W. Weide, S.H. Zweben, 'Part I: The RESOLVE Framework and Discipline - A Research Synopsis,' *Software Engineering Notes* 19, 4, pp. 23-28, October 1994.

17. D.E. Perry, A.L.Wolf, 'Foundations for the Study of Software Architecture,' *Software Engineering Notes*, Vol. 17, No. 4, pp. 40-52, October 1992.

18. *The RAISE Development Method*, The RAISE Method Group, Prentice Hall, 1995.

19. D.J. Richardson, A.L. Wolf, 'Software Testing at the Architectural Level,' *Proceedings of the Second International Software Architecture Workshop*, pp. 68-71, San Francisco, USA, October 1996.

20. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, *Object-oriented modeling and design*, Prentice Hall, 1991.

21. M. Shaw, R. DeLine, D.V. Klein, T.L. Ross, D.M. Young, G. Zelesnik, 'Abstractions for software architecture and tools to support them,' *IEEE Transactions on Software Engineering*, April 1995.

22. M. Shaw, D. Garlan, 'Characteristics of Higher-level Languages for Software Architecture,' *CMU-CS-94-210*, December 1994.

23. M. Shaw, D. Garlan, *Software Architecture - Perspectives on an Emerging Discipline*, Prentice Hall, 1996.

24. S. Shlaer, S.J. Mellor, 'Recursive Design of an Application-Independentt Architecture,' *IEEE Software*, pp. 61-72, January/February 1997.

25. C. U. Smith, *Performance Engineering of Software Systems*, Addison-Wesley, 1990.

26. A. Terry, F. Hayes-Roth, Erman, Coleman, Devito, 'Overview of Teknowledge's DSSA Program,' *ACM SIGSOFT Software Engineering Notes*, October 1994.

27. Wirfs-Brock, B. Wilkerson, L. Wiener, *Designing Object-Oriented Software*, Prentice Hall, 1990.

28. N. Wirth, 'Program Development by Stepwise Refinement,' *Communications of the ACM*, (14):221 - 227, 1971