An Approach to Quality Achievement at the Architectural Level: AQUA

Heeseok Choi[1], Keunhyuk Yeom[2], Youhee Choi[3], Mikyeong Moon[2]

[1] NTIS Organization, Korea Institute of Science and Technology Information
Eoeun-dong 52-11, Yuseong-gu, Daejeon, 305-806, South Korea
choihs@kisti.re.kr
[2] Department of Computer Engineering, Pusan National University
30 Changjeon-dong, Keumjeong-gu, Busan, 609-735, South Korea
{yeom,mkmoon}@pusan.ac.kr
[3] Embedded S/W Research Division, Electronics and Telecommunications
Research Institute, 161 Gajeong-dong, Yuseong-gu, Daejeon, 305-700, South Korea
yhchoi@etri.re.kr

Abstract. Architecture-based software development plays an important role in successfully developing and managing large and complex software systems. Recently, there have been a number of studies for designing, evaluating, or transforming architectures. However, there is not much work being done for closely connecting an architectural evaluation with an architectural transformation in order to achieve quality attributes during the architecture-based software development. For this reason, it is still difficult to achieve consistently quality attributes at the architectural level. This paper presents an approach to quality achievement in architecture-based software development, which is called AQUA. The AQUA involves two distinctive activities, which are architectural evaluation and transformation, but these activities can be seamlessly combined through producing relevant artifacts based on the design decisions that led to the architecture. Due to the proposed approach, we can expect to achieve quality attributes in architecture-based software development.

1 Introduction

Quality attributes of large software systems are principally determined by the system's software architecture, which represents a common high-level abstraction of the system [1,2]. Therefore, architecture-based software development plays an important role in successfully developing and managing large and complex software systems [1,3,4].

Recently, there have been a number of studies for designing, evaluating, or transforming an architecture. Namely, methods for designing software architectures for developing quality softwares[1], methods for evaluating software architectures with respect to software quality attributes (e.g.[2],[3],[4],[5]), or methods for transforming a software architecture in order to improve one or more of its quality attributes (e.g.[3],[6],[7]) have been studied. There is, however, not much work being done for closely connecting architectural evaluation with architectural transformation

in order to achieve quality attributes during the architecture-based software development. For this reason, it is still difficult to achieve consistently quality attributes at the architectural level.

This paper presents an approach to quality achievement in architecture-based software development, which is called AQUA afterwards. The AQUA involves two distinctive activities, which are architectural evaluation and transformation. However, these activities can be seamlessly combined through allowing the evaluation artifacts to be effectively utilized for architectural transformation centering around design decisions acquired from architectural evaluation. Furthermore, activities for architectural evaluation in the AQUA play a significant role in revealing any potential defects or assessing the fulfillment of required quality requirements, and activities for architectural transformation play a significant role in reducing defects in the architecture or making changes to the architecture.

2 Overview of AQUA Process

In this paper, we present an approach to quality achievement in architecture-based software development, which is called AQUA. The AQUA provides software architects with a mean for achieving quality attributes at the architectural level. For the purpose of achieving quality attributes during architecture-based software development, it is necessary to transform architectures based on the evaluation results as well as to evaluate them. Therefore, the AQUA involves two kinds of distinctive activities, which are architectural evaluation and transformation. Namely, the AQUA integrates activities for providing insights of an architecture with respect to its desired qualities with activities for making changes to the architecture within a framework. Due to the AQUA, it can be easily performed to achieve quality attributes at the architectural level without difficulties of bridging heterogeneous approaches. In other words, the information acquired from architectural evaluation can be effectively utilized in making changes to the architecture for quality achievement.

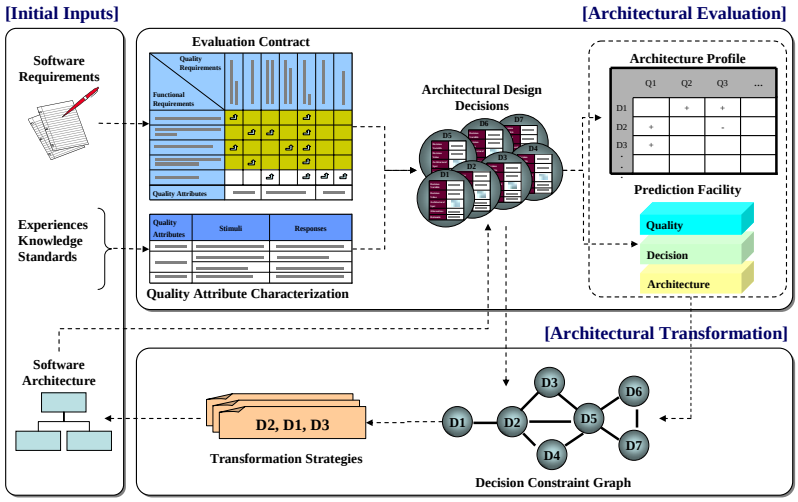

Fig. 1. The AQUA process

Figure 1 presents an overview of the AQUA. The AQUA first needs the generation of an evaluation contract for scoping software requirements and identifying the desired quality attributes of an architecture. Then the AQUA requires characterizing each quality attribute for specializing explicitly the characteristics of quality attributes. Next, the AQUA includes the identification of architectural design decisions having an important impact on the achievement of quality attributes. Such design decisions can be identified by characterizing key designs relevant to quality achievement in the presented architecture with considering the characteristics of quality attributes. Based on the decisions, the AQUA includes the generation of an architecture profile representing the quality achievement of the architecture, and gets to generate a prediction facility helpful in understanding the traceability between quality attributes and architectural designs. Namely, it provides insights concerned with the quality achievement of the architecture with respect to its desired qualities. According to the insights about quality achievement, it is necessary to make changes to the architecture for the purpose of achieving quality attributes. Furthermore, the changes should be able to be planned for avoiding unnecessary changes. For this reason, the impact on other design decisions should be considered before applying the changes to the architecture. Therefore, the AQUA includes the generation of a decision constraint graph for representing explicitly the dependencies among design decisions, then for tracing easily the impacts of a decision change. Through using the decision constraint graph, the AQUA guides the establishment of transformation strategies that lead to a new architecture. Finally, the activities of the AQUA for conducting an evaluation and transformation of an architecture can be repeatedly performed until reaching the desired levels of quality attributes in the architecture. Therefore, the AQUA provides software architects with a mean that supports achieving quality attributes during architecture-based software

development. In the sections below, these artifacts are discussed in more detail.

3 Quality Achievement Activities of the AQUA Process

3.1 Understanding Quality Achievement Goals using Evaluation Contract

The evaluation contract means the consensus between users and software architects about expectations from the evaluation for quality achievement. Namely, expectations from the evaluation can be concluded and negotiated. This contract includes the lists of quality and functional requirements, their relationship, and identifies quality goals of architecture.

Figure 2 represents generating an evaluation contract. To generate an evaluation contract, software architects first document quality requirements and functional requirements of a system separately. In general, functional requirements have relations to one or more quality requirements. Subsequently software architects determine the scope of functional requirements, then software architects determine the scope of quality requirements. Finally, software architects identify the quality attributes representing the goals of an architectural evaluation for quality achievement.

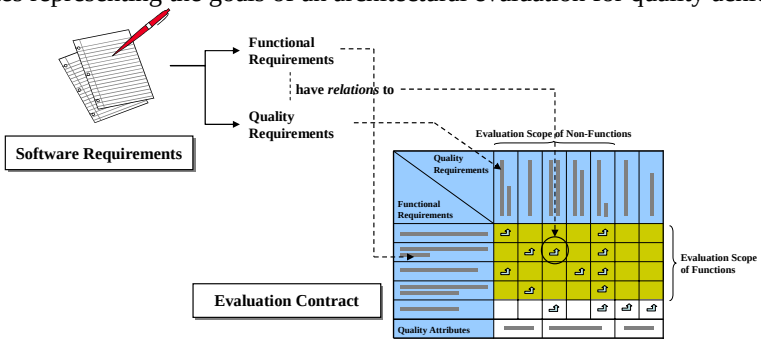

Fig. 2. Generating an evaluation contract

3.2 Finding Architectural Design Decisions

Software architecture is composed of architectural design decisions, which are the aspects of an architecture that have a significant impact on achieving quality attributes, such as components, connectors, and configuration. Namely, architectural decisions are made from an overall system perspective. Essentially, these decisions identify the system's key structural elements, the externally visible properties of these elements, and their relationships, and they define how to achieve the architecturally significant requirements[3]. Since architectural design decisions represent decisions on various design alternatives applicable to design problems during architectural

design, these decisions can be interpreted as pairs of decision variable and decision value. The following are to illustrate the concepts of decision variables and decision values, respectively:

- A ***decision value*** describes a design itself applied to the current architecture as the selected solution out of design alternatives applicable to each design issue. The decision values can be easily conceived from a well presented software architecture. More specifically, parts of designs relevant to functional requirements within the evaluation scope for quality achievement should be first identified. Next, each design is summarized in terms of design elements, relationships, and their properties. Finally, the decision values describing meaningfully key designs are identified in the presented architecture through characterizing such design summaries.
- A ***decision variable*** describes the architectural design issue that each selected solution is addressing, such as "What are the big parts of the system?" and/or "How are they connected?". Such decision issue can be found by analyzing decision values based on architectural knowledge such as design patterns, styles, and architectural views. Namely, the decision variables can be determined by asking questions about why the decision values have resulted from software requirements.

Figure 3 represents the finding of architectural design decisions. As in the above illustrations about a decision variable and a decision value, software architects should first identify design areas relevant to functional requirements within the evaluation scope (① in Figure 3). Subsequently, key designs should be summarized (② in Figure 3). Then decision values can be identified through characterizing design summaries (③ in Figure 3). Finally, decision variables are determined by identifying one or more design issues that each decision value involves (④ in Figure 3).

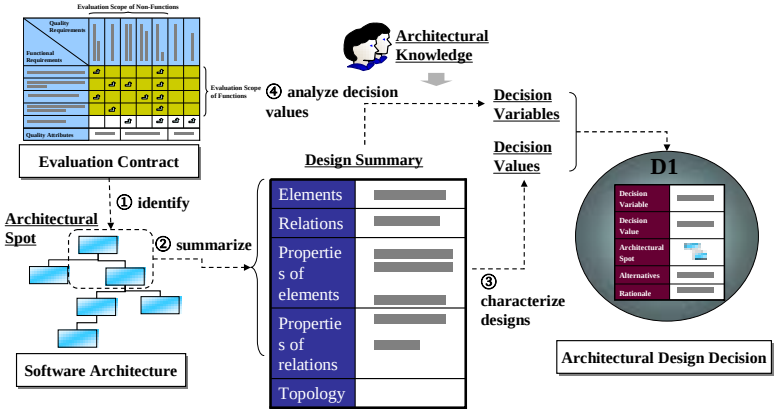

Fig. 3. Finding architectural design decisions

3.3 Generating Decision Constraint Graph

Architectural design decisions also have relations to other decisions in terms of the consistency among designs. For instance, a decision for determining elements of a system should be consistent with a decision for structuring the system. The decision constraint graph is a graph for maintaining the consistency among design decisions. The graph helps in representing explicitly the dependencies among design decisions, and in tracing easily the impacts of a decision change. Here, architectural design decisions introduce two kinds of design constraints, which are unary and binary constraints. The following are to illustrate unary constraints and binary constraints:

- A ***unary constraint*** captures any constraint to the design that the chose alternative (the decision value) might pose, which restricts design alternatives applicable to each design issue (the decision variable). In order to determine unary constraints, software architects should first analyze the characteristics of decision values at various points in the design. For example, if the design elements support the concurrency of system, it can be considered that there is the constraint equal to concurrency support. Next, software architects should determine whether the characteristics are closely related to the requirements specified in previous evaluation contract. Finally, the characteristics irrelevant to requirements should be excluded.
- A ***binary constraint*** captures any constraint for design consistency that two decision values might pose each other, which represents a condition restricting design alternatives applicable to relevant decision variables. In order to determine binary constraints, software architects should analyze only the characteristics causing consistency problems between two decision values.

Figure 4 represents the generation of a decision constraint graph. Each node in the graph represents a decision variable, and each edge in the graph represents constraint relationship between two decision variables. To generate a decision constraint graph, software architects should first document two kinds of constraints according to the above illustration; unary constraints and binary constraints. Based on the identified constraints, the relationships among design decisions are determined with respect to design consistency. As a result, nodes and edges of decision constraint graph are defined.

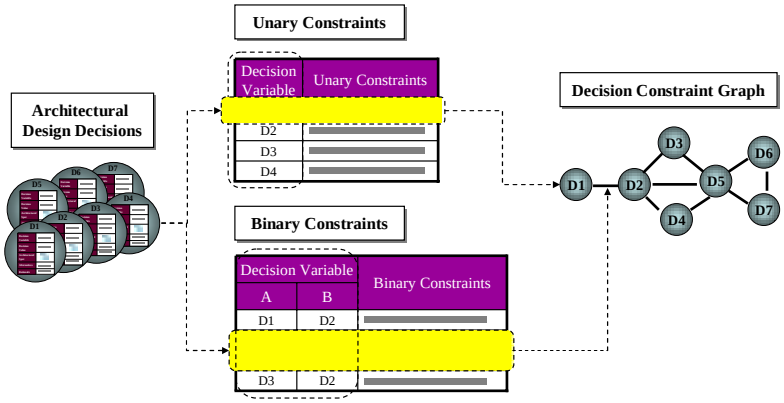

Fig. 4. Generating a decision constraint graph

3.4 Applying Architectural Changes for Quality Achievement

Software architects can apply various changes in order to reduce any potential defects, or to make changes to the architecture. Each change leads to a new version of the architecture that has the same functionality, but different satisfaction for desired quality attributes. However, applying a change to a specific design area may have an impact an adjacent design area or whole architecture. Therefore, it is necessary to analyze how a change to the specific area affected on other areas and to determine applicable design alternatives for the target area needs to be changed.

To do this, software architects first find architectural alternatives. To find architectural alternatives, we can utilize various design theories or refer to other alternatives from candidate architectures or architect's experience. To select appropriate architectural options, we should analyze the architectural options with respect to the scope and impacts of applying them. In particular, architectural options must be consistently selected against other architectural decisions. To achieve this goal, we propose checking the following consistencies based on the decision constraint graph.

- ***Node consistency*** means the satisfaction of unary constraints restricting the decision values applicable to a decision variable. This also means that all architectural alternatives unsatisfying unary constraints on a decision variable would be pruned from the candidate solutions of the variable.
- ***Arc consistency*** is the satisfaction of binary constraints representing a condition between two decision variables. Namely, architectural alternatives applicable to the specific design should satisfy arc consistency between the designs.

4 An Example: House Alarm System

To address the practical applicability and features of our approach, we have

chosen an example that is familiar but rich with interesting design and architectural problems, that of the House Alarm System (discussed in [8] and elsewhere). A house alarm system consists of a main unit to which a number of sensors and alarms are connected. The sensors detect movements in the guarded area, and the alarms generate sounds and/or lights to scare off an intruder. The total area that can be guarded is divided into cells, where a cell contains some sensors and some alarms that guard a specific area. Since the house alarm system is included in real-time systems, there are some special concerns when modeling the system. Naturally, concurrency, communication, and synchronization are the most important factors, but fault tolerance, performance optimization, and distribution must also be dealt with when modeling a house alarm system.

In this example, we focused on illustrating that the AQUA seamlessly supports two distinctive activities centering around architectural design decisions. For this reason, we omitted some artifacts such as quality attribute characterization and architecture profile in this paper. Furthermore, this paper introduced only architectural designs necessary for understanding the proposed approach. As shown in Figure 5 through Figure 7, partial software architecture of a house alarm system was presented with the 4+1 view model in UML. Firstly, Figure 5 represents a set of key abstractions for the system and their logical relationships. Sensors and alarms are connected to a cell handler, which is an active class that handles a specific cell. The cell handler is connected to the system handler, which is an active class that handles the user communication. Next, Figure 6 represents a partially logical organization between sensors and alarms in the system. The house alarm system is structured based on the shared memory style. Finally, Figure 7 shows the interaction when a sensor detects something. It then sends an asynchronous alarm signal to the cell handler. Subsequently, the cell handler sends in parallel synchronous trigger signals to all alarms and an asynchronous alarm signal to the system handler. Inside the system handler, the alarm signal is handled synchronously.

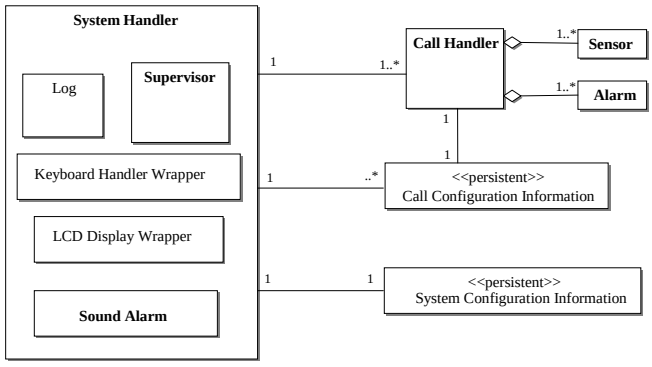

Fig. 5. A set of key abstractions for the system and their logical relationships

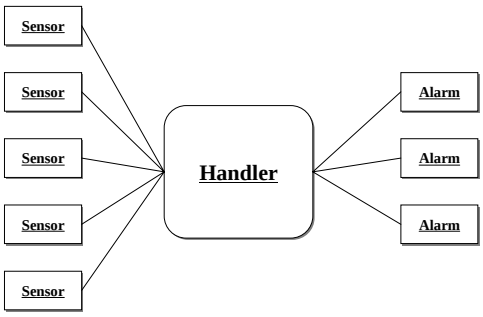

Fig. 6. An architecture of house alarm system based on the shared memory style

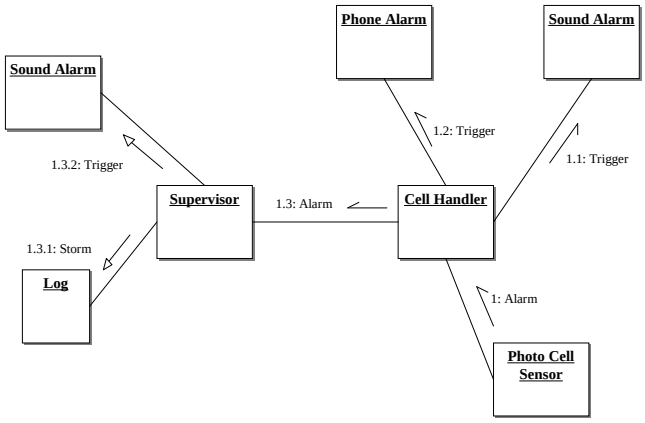

Fig. 7. The interaction when a sensor detects something

4.1 Understanding Quality Achievement Goals using Evaluation Contract

In this example, we first defined the evaluation contract. We found that the house alarm system had the functionality such as activity detection, alarm generation, user communication, and system monitoring. In addition, we found that the system had non-functionality relevant to quality attributes such as performance, concurrency, and fault tolerance. When we defined the evaluation contract, we first placed the functional requirements and non-functional requirements in a row and column of the evaluation contract, respectively. The relationships between functional requirements and non-functional requirements were naturally determined. Here, we restricted the evaluation scope for evaluating the quality achievement of current architecture within the requirements listed in the oblique area of Figure 8. As a result, two kinds of quality attributes (i.e. performance, concurrency) were identified as the things to be dealt with in this example. In this way, we could start quality achievement at architectural level with the generated evaluation contract.

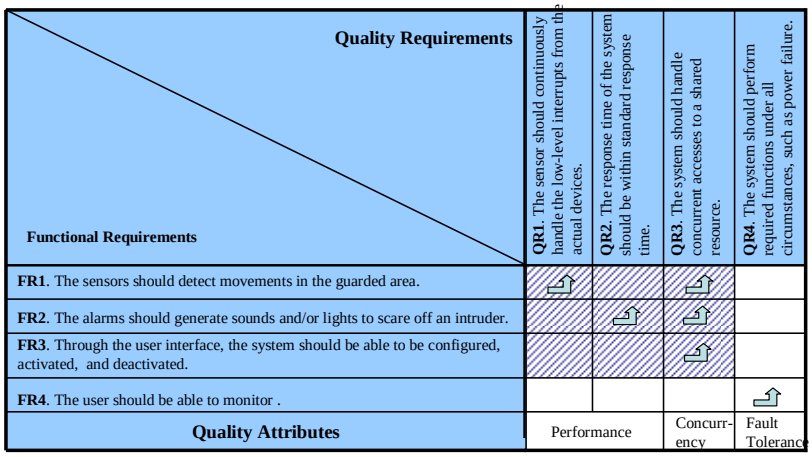

Fig. 8. Evaluation contract

4.2 Finding Architectural Design Decisions

In this example, we identified some design decisions necessary for illustrating our approach. Table 1 summarized architectural design decisions of a house alarm system. There were the decisions that identify the system's key structural elements, their properties, and their relationships. In addition, there were the interesting decisions such as choosing patterns and message types. They became a leverage to help us understand the architecture in practice.

Table 1. Architectural design decisions

Architectural design decision		**Brief Description**
Decision Variable	**Decision Value**	
Design Elements (D1)	Three Kinds of Logical Elements	-The system consists of three kinds of logical elements such as sensors, alarms, and handlers.
Roles of Elements (D2)	-User Communication -Cell Handling -Activity Detection -Sound/Light Effects	-The system handler handles the user communication. -The cell handlers handle a specific cell consisting of sensors and alarms. -The sensors handle the low-level interrupts from the actual device, then detect activity in a specific area. -The alarms handle the low-level communication with the device, then generate sound and light effects.
Properties of Elements (D3)	Active Class	-The main elements are designed as active classes.
Structure of System (D4)	Shared Memory Style	-The system is structured based on the shared memory style.
Task Partition (D5)	Unit of Active Class	-The task is modeled as the unit of active class.

Message Types (D6)	Use of Synchronous & Asynchronous Messages	-Inside the system handler, the synchronous messages are used. But outside the system handler, the asynchronous messages are used.
Task Interaction (D7)	Event-based Communication	-The interaction among tasks is performed via event-based communication
Task Synchronization (D8)	Task Monitoring	-The system monitors concurrently trying to modify or access a shared resource.

4.3 Generating Decision Constraint Graph

For the generation of a decision constraint graph, we identified unary and binary constraints on the design decisions summarized in Table 1. As results, Table 2 summarized unary constraints to each decision, and Table 3 summarized binary constraints between the design decisions. Furthermore, the consistency relationships among design decisions were naturally found by determining binary constraints.

Table 2. Unary constraints

Decision Variable	Unary Constraints
Design Elements	The inputs of system should be different from the outputs of system.
Roles of Elements	Assigned roles should support the concurrency of system.
Properties of Elements	Real-time properties of system should be supported.
Structure of System	Relevant data among elements should be shared.
Task Partition	Real-time properties of system should be supported.
Message Types	Both synchronous and asynchronous properties should be supported.
Task Interaction	The system should be run in terms of external events.
Task Synchronization	The tasks concurrently trying to access a shared resource should be synchronized.

Table 3. Binary constraints

Architectural Design Decisions		Binary Constraints
Decisions	Decisions	
Design Elements	Roles of Elements	Design elements should have the independent roles.
Design Elements	Structure of System	There should be an element for sharing data.
Roles of Elements	Properties of Elements	The concurrency among elements should be satisfied.
Roles of Elements	Structure of System	An element should have a role for sharing data.
Task Partition	Design Elements	The elements having independent roles should be identified as concurrent tasks
Task Partition	Task Synchronization	The tasks concurrently trying to access a shared resource should be synchronized.
Task Interaction	Message Types	The tasks should interact with each other by transmitting any type of message.
Task Synchronization	Structure of System	The tasks concurrently trying to access a shared resource should be synchronized.

Using the decisions and constraints described in Table 1 through Table 3, we generated the decision constraint graph as shown in Figure 9. Though a few decisions

were considered in this example, the relationship among them was more complex in practice. Therefore, we found that the decision constraint graph would be useful for showing complex relationships among the decisions. In other words, the graph showed that the decision had the consistent relationship with one or more of them.

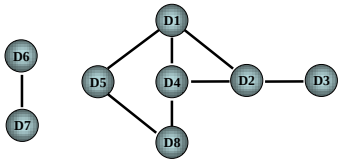

Fig. 9. Decision constraint graph

4.4 Applying Architectural Changes for Quality Achievement

As an example, we tried to transform the current decision for synchronization among the tasks concurrently trying to access a shared resource according to the previous evaluation results. Namely, we considered applying a periodic execution mechanism using a scheduler [3] to the design area of *D8*(Task Synchronization). Prior to applying an alternative to a design area, however, it is necessary to analyze its impact on an architecture. To do this, we gradually performed impact analysis starting from change of *D8*. Figure 10 represents the result of impact analysis for architectural transformation. As shown in Figure 10, some nodes of a graph were traversed during impact analysis by checking both node consistency and arc consistency. Since the change to the *D8* doesn't violate unary and binary constraint for *D4*(Structure of System), the change to the *D8* doesn't have impact on *D4*. However, the adjacent node *D5*(Task Partition) is affected for reasons of consistency violation. Then additional changes for the nodes *D1*(Design Elements) and *D4*(Structure of System) were needed by adding a scheduler to the system. As a result, we easily established a transformation strategy consisting of *D5*, *D1*, and *D4* by the sequence of ① through ③ in Figure 10. Through transforming the presented architecture according to the transformation strategy, we expect to reduce the possibility of errors, but it can reduce performance. Finally, it is necessary to validate the correctness of the transformations performed. It can be achieved by performing iteratively the AQUA process.

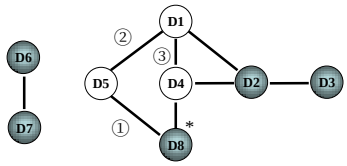

Fig. 10. Establishing a transformation strategy

5 Comparison with Existing Methods

We compared the AQUA with the existing methods in terms of activities necessary for achieving qualities at the architectural level. To do this, we first identified the activities through analyzing existing studies on architectural evaluation and transformation. When we analyzed them, we particularly focused on understanding the full process from quality identification to quality achievement. As a result, we identified six key activities to be handled at the architectural level for quality achievement as described in Table 4. The activities are as follows:

- ***Identifying desired qualities*** is an activity to determine the kinds of quality attributes to be dealt with during the quality achievement of an architecture. Due to this activity, the goal for quality achievement can be clearly defined.
- ***Specializing quality attributes*** contributes in acquiring more informative characteristics of quality attributes. In addition, it contributes in identifying architectural decisions having a significant impact on achieving quality attributes.
- ***Analyzing an architecture*** is an activity to determine quality achievement of current architecture with respect to its desired qualities. It provides insights concerned with the quality achievement of architecture.
- ***Analyzing change impacts*** is an activity to analyze the effects of making changes to the architecture on the other quality attributes or other designs. In particular, it helps make sure the design is consistent with one or more of them.
- ***Modifying an architecture*** leads to a new version of the architecture with the same functionality, but with different values for its desired qualities.
- ***Validating an architecture*** is an activity to validate the correctness of the transformations performed. Through validating the architecture, the quality achievement can be confirmed.

Then we analyzed whether the existing methods and the AQUA effectively supported the activities necessary for quality achievement or not. To do this, we summarized the artifacts closely related to the activities through analyzing some methods including the AQUA as described in Table 4. The artifacts mean that the methods effectively support the described activity. As illustrated in Table 4, the existing methods fail to support the activities for quality achievement consistently. Compared with the methods described in Table 4, however, the AQUA can effectively support quality achievement in architecture-based software development through producing explicitly the artifacts relevant to quality achievement at the architectural level based on the design decisions.

Table 4. Comparison with existing methods

Methods / Activities for Quality Achievement	Architecture Evaluation			Architecture Transformation			Architecture Evaluation & Transformation
	ATAM[4]	SAAM[4]	ARID[4]	Bosch's Approach[3]	Carriere's Approach[6]	Krikhaar's Approach[7]	AQUA
Identifying desired qualities	Scenarios	Scenarios	Scenarios	Activity only	Not defined	Not defined	Evaluation contract
Specializing quality attributes	Utility tree	Scanarios	Not defined	Quality profiles	Not defined	Quality metrics	Quality attribute characterizations
Analyzing an architecture	Sensitivity points &Trade-off points	Sensitivity points	Activity only	Activity only	Features	Not defined	Architectural decisions Architecture profile
Analyzing change impacts	Not defined	Not defined	Not defined	Activity only	Not defined	RPA model	Decision constraint graph
Modifying an architecture	Not defined	Not defined	Not defined	Activity only	Activity only	RPA model	Transformation strategy
Validating an architecture	Not defined	Not defined	Not defined	Activity only	Activity only	Not defined	Architecture profile (by iteration)

6 Conclusions and Future Work

We presented an approach to quality achievement in architecture-based software development, which is called the AQUA. In addition, we applied the proposed approach to the House Alarm system to illuminate the approach. The AQUA involves two distinctive activities, which are architectural evaluation and transformation. Here, architectural evaluation plays a significant role in revealing any potential defects or assessing the fulfillment of required quality requirements, and architectural transformation plays a significant role in reducing defects in the architecture or making changes to the architecture. However, the AQUA effectively integrates the activities for providing insights of an architecture with respect to its desired qualities with the activities for making changes to the architecture within a framework. Furthermore, the AQUA seamlessly supports the activities relevant to quality achievement centering around the architectural design decisions by explicitly documenting them. Through following the AQUA, it can be easily performed to achieve quality attributes at the architectural level without difficulties of bridging heterogeneous approaches.

In the future, we expect that this approach may be more complemented and extended as a result of ongoing researches. Presently, we are interested in several issues in supporting architecture-based software development. In particular, the development of mechanisms for process automation is considered to be important. We believe our approach will be effectively involved at the early stages of a software development lifecycle.

References

1. Bass, L., Clements, P., and Kazman, R., *Software Architecture in Practice, 2/E*, Addison-Wesley, 2003.
2. Kazman, R. et al., "The Architecture Tradeoff Analysis Method", *Proceedings of the 4th IEEE International Conference on Engineering of Complex Computer Systems*, August 1998, pp.68-78.
3. Bosch, J., *Design and Use of Software Architecture*, Addison-Wesley, 2000.
4. Clements, P., Kazman, R., and Klein, M., *Evaluating Software Architectures*, Addison-Wesley, 2002.
5. Dobrica, L. and Niemela, E., “A Survey on Software Architecture Analysis Methods”, *IEEE Transactions on Software Engineering*, IEEE Computer Society, Vol. 28, No. 7, July 2002, pp.638-653.
6. Carriere, S.J., Woods, S., and Kazman, R., "Software Architectural Transformation", *Proceedings of the 6th Working Conference on Reverse Engineering*, October 1999, pp.13-23.
7. Krikhaar, R., et al., "A Two-phase Process for Software Architecture Improvement", *Proceedings of the International Conference on Software Maintenance*, August 1999, pp.371-380.
8. Eriksson, H.E. and Penker, M., UML Toolkit, John Wiley & Sons, 1998.