Quantitative Tradeoff Analysis of Software Architecture using the Architecture Analysis and Design Language

Jihyun Lee
S/W Content Research Laboratory
Electronics and Telecommunication Research Institute
Daejeon, Korea
jihyun@etri.re.kr

Dan Hyung Lee
School of Engineering
Korea Advanced Institute of Science and Technology
Daejeon, Korea
danlee@icu.ac.kr

***Abstract*—In software architecture analysis domain, the problem of satisfying a desired level of quality attributes has been researched. Architecture tradeoff analysis methods have been developed to help architects to examine whether an architecture satisfy the quality attributes. This paper suggests a quantitative tradeoff analysis method of architecture using an architecture analysis and design language. The suggested method is applied to operating system development for automobiles.**

Keywords-software architecture; tradeoff analysis; quality attributes

I. INTRODUCTION

Software architecture is an artifact representing design decisions on functional and non-functional requirements such as quality attribute and an understanding of a high-level design is a way to determine a system in order to satisfy the quality attributes and use the results in analyzing the design decisions. However, existing architecture analysis methods depend on only qualitative experiences from expert architects to derive architectural design decisions. And the decisions affecting more than one quality attribute are difficult to identify and satisfy the desirable level of two or three pairs of quality attributes.

The goal of this paper is to decide what the best architecture is in a situation such as analyzing unbalanced design tradeoffs by providing a quantitative measurement method for handling the seesawing quality goals of a system under construction. The paper is composed as follows. In Chapter 2, related works are reviewed. A novel approach for a quantitative architecture tradeoff analysis is proposed in Chapter 3. Chapter 4 presents validation through a case study for the proposed approach. Chapter 5 concludes the paper.

II. RELATED WORKS

A. Existing Software Architecture Analysis Method

Several architectural analysis methods have been existed. Among of the analysis methods, the Architecture Tradeoff Analysis Method (ATAM) [1] has been developed in the Carnegie Mellon Software Engineering Institute (SEI). The ATAM has purpose of finding the best architecture for quality attribute requirements of a system. Moreover, the ATAM helps architects select more appropriate architecture by describing scenarios, making a utility tree, and analyzing architectural sensitivity points and tradeoffs. However, the ATAM has focused on not providing precise analysis results but finding risks created by architectural decisions.

B. Quantitative Tradeoff Analysis Techniques

For a quantitative architecture measurement, first, a quality scenario depending on the particular issues can be a domain specific technique. Second, a checklist documenting domain knowledge can be another architecture evaluation technique [2]. Third, architectural prototyping is defined to consist of a set of executables created to examine architecture qualities that are requested from customers. The architectural prototyping is composed of the following process: designing, building, and evaluating the architectural prototypes [3].

C. Quality Representation in Architecture

Among diverse architectural description languages (ADLs), the Architecture Description and Analysis Model (AADL) is a standard modeling language for embedded an application system's architecture in components and interactions. AADL allows a software architecture to be described with much liberty; therefore, runtime features such as response time, latency time, and level of flexibility could be defined in AADL. However, AADL was not intended to be used for software architecture tradeoffs. Therefore, a liberal AADL syntax concerning the feature and property will be fully used for specifying quality attributes and tradeoff points.

III. QUANTITATIVE ANALYSIS METHOD

Quantitative tradeoff analysis method has purpose of acquiring quantitative values from an architecture

978-0-7695-3642-2/09 $25.00 © 2009 IEEE
DOI 10.1109/SNPD.2009.65

representation model to examine quantitatively tradeoffs among quality attributes. The method based on measurement can help architects understand architecture precisely and identify tradeoffs among quality attributes using quantitatively measured values.

A. Quality Attributes and Metrics

A quality driven quantitative approach should include designing, building, and evaluating architecture for quality attributes, which should be related to metrics to measure quantitative values from desired quality goals. Quality attributes need to be separated in sub-characteristics. Then, metrics needs to be mapped to the sub-characteristics for measurement. The measured data are used in providing insights in the architectural tradeoff analysis. For example, quality attributes such as performance, flexibility, and availability of automotive applications are shown in table 1.

TABLE I. QUALITY ATTRIBUTE HIERARCHY

QA	Sub-characteristic	Metric
Availability	Fairness	Conformance class
	Stability	Operation elapsed time
Flexibility	Maintainability	Dependency among modules
	Modifiability	Scope of modification
		Component flexibility
Performance	Predictability	Throughput
		Waiting/running/latency time
		Response time

B. Architecture Representation in AADL

A key architectural element in AADL is a component type, a property set, and a component implementation [4]. The AADL elements needed for representing software architecture are summarized in the following table.

TABLE II. ELEMENTS COMPOSING AADL MODEL

Category	AADL Elements
Modeling application software	Component access, subprogram calls, data access connections, data exchange
Structuring alternative architectural configuration	Partitioning runtime configuration
Analysis abstractions	Properties
Modeling execution and concurrency	Runtime execution, timing associated with multiple execution paths
Modeling scheduling elements	Runtime interaction, coordination, timing associated multiple execution paths

The component type is defined through defining elements of a component and its interfaces. A property set is defined with the elements such as property types, property definitions, and constants. A component implementation is declared by specifying internal structural elements such as subcomponents, subcomponent connections, call sequences, properties, and etc.

To represent architecture, a collection of AADL components are linked using connections and subprogram sequences. Logical data connections are represented using flows. Design invariants are represented with constant properties. And design constraints of the architecture are represented by using the properties.

C. Quality Attribute Representation in AADL

AADL components can contain property values, which provide information about the components. The declared properties can support analysis quality attributes by way of representing the inputs to the analysis and showing the analysis results.

TABLE III. ELEMENTS PROPERTY COMPOSITION ATTRIBUTE

Attribute Name	Description
Property association	A value or a list of values are assigned to a named property
Property set	A named collection of property types, names, and constants are defined
Property type	Property type and a set of acceptable values for properties are defined
Property name	A property is defined by declaring a name, identifying a type for the property, and applying it to flows, ports, and etc.
Property constant	A value for a property is defined

The example below represents three kinds of declarations for property type, property constant, and property name that are in a “Property_Example” property set. The name of the property set is “Property_Example”, and the “Units” type declares two elements, inch and foot. The next type, “Width”, is an integer type labeled by the “units” from the “Property_Example”. “One_foot” is declared as a property constant of type “Property_Example::Width” and has a value of 1 foot. The “Thickness” property is declared to have a value of One_foot with the type “Width”.

```
property set Property_Example is
   -- unit type declaration
   Units: type units (inch, foot => inch * 12);
   -- integer type declaration
      Width: type aadlinteger units
   Property_Example::Units;
   One_foot: constant Property_Example::Width => 1
   foot;
   -- property declaration
   Thickness:
      Property_Example::Width => value
         (Property_Example::One_foot)
         applies to (documentation, periodicals);
end Property_Example;
```

Figure 1. Example of Property Declaration.

D. Performance Representation and Analysis from AADL Model

To predict a system performance and runtime characteristics, AADL allows flow related properties. Through abstract application logics in a component, externally observable behavioral system information can be described and measured.

As a representative example, logical flows are represented in a path beginning at a source component and terminating at a sink component. Flow specification is represented using key elements: a source, a sink, and a flow path declaration combined with various ports, connections, and data. A source is a component; a sink is a component; and a flow path is a flow through a component from its incoming ports to its outgoing ports. The flows describe actual flow sequences through components across one or more connections, and they are declared in two types of implementations: flow implementation and end-to-end flow. Flow implementation is for how a flow specification of a component is realized in a component implementation.

TABLE IV. METRICS FOR PERFORMANCE

Metric for Flow Quality	AADL Property
Maximum amount of elapsed time	Latency: Time applies to (flow, connections);
Expected latency	Expected_Latency: Time applies to (flow);
Actual latency	Actual_Latency: Time applies to (flow);
Execution time	Time range of compute_Execution_Time
Propagation_Delay	Queuing or sampling delay
Deadline	Deadline (for periodic, aperiodic, and background threads)

A logical flow representation in AADL is highly related to specifying quality characteristics such as end-to-end latency, end-to-end response time, error propagation, and reliability. Therefore, the end-to-end flow and end-to-end latency are analyzed considering properties that specify its expected timing information and the actual timing information from the flow-related components and ports.

The latency property is related to the maximum amount of elapsed time determined by the connection latency and end-to-end flow latency within the component. Expected_Latency is a property to mark the desired latency.

Actual_Latency is a calculated latency according to the traversing system topology. In AADL, the expected latency can be specified using the "Expected_Latency" property; however, AADL itself does not prohibit the actual latency from exceeding the expected latency. Therefore, architects have a responsibility to verify the satisfaction of this property.

A worst-case latency is determined as the incoming connection is a delayed connection or immediate connection.

E. Flexiblity Representation and Analysis from AADL Model

Flexibility is defined as the quality attribute for how much a software upgrade helps a system to maintain its functionality in an uncertain environment by adding functionality for strategic architecting. Therefore, the flexibility of a software architecture is degenerated when new functions are added to solve a problem.

Frequent changes in hardware, software, and customer's expectations occur through a performance improvement, security enforcement, and bug fixes. However, reliable metrics are absent, and no formal criteria for flexibility have so far been offered.

A quantitative measurement for software flexibility depends on algorithms and change impacts. Finer granularity of software is required from adding activities in order to recognize the types of software addition with high priority. In the definition of flexibility metrics, a design policy is an important criterion. A metric for quantifying the flexibility from design policies is defined as the complexity of executing particular evolution steps.

To measure the flexibility in terms of functionality extension, the flexibility level is divided into 5 levels according to the number of affected components for functionality addition. As the architecture representation increases with structural composing elements and quality attribute constrains, formal syntax for describing the elements and the characteristics of AADL is used. In the AADL syntax, a component is represented in its type definition and realization.

TABLE V. AADL ELEMENTS FOR DESIGN EXTENSION

Design Extension Category	Sub-Elements
Component type definition	Ports, access, subprogram, parameter
Component realization	Property types, property definitions, constraints
Component implementation deletion	Port, access, parameter, modes, mode transitions, refines type, subcomponents, connections, call sequences, flows, properties, all
Component type refinement	Extends, features, flows, properties, all
Component implementation refinement	Refines type, subcomponents, connections, call sequences, flows, properties, all

A flexible architecture should be able to accept new and changed functionality. Based on this concept, flexibility is defined as the quality attribute deciding upon the degree of adaptation for the functionality extension. The degree of adaptation for the functionality extension should be based

on flexibility scenarios. With symptoms frequently occurring in an AADL architecture, architectural symptoms for flexibility may be separated into 5 levels, but they are dependent on quality concerns and scenarios:

- Level 1: In this level, a new component type and implementation is specified in the initial architecture
- Level 2: When the addition of communications are requested from the initial architecture, the changed alternative architecture is assumed at flexibility level 2
- Level 3: Due to a refinement of component type and implementation, if connectors need to be severely decomposed, the architecture is supposed to be at flexibility level 3
- Level 4: When the components are decomposed, including the addition of subcomponents, ports, and connectors, this architecture is supposed to be at flexibility level 4
- Level 5: Flexibility level 5 is approached only after end-to-end flows are specified

The flexibility metric is defined according to the 5 kinds of symptoms in an AADL architecture. The basic concept of a flexibility metric is from the metric of architecture changes based on structural distance [5]; however, the distance metric is focused on only the distance from a starting architectural element to and an end element for measure. Therefore, it does not identify the exact part of the difference from the architecture. But, this drawback is improved in this thesis by considering the symptoms for the changed elements inside and outside of a component. The flexibility metric is defined from level 1 to level 5. As the levels go from level 1 to 5, the architecture design is extended from the purpose of solving internal causes to the facility for handling external causes.

IV. CASE STUDY

OSEK/VDX [6] is a project in the automotive industry, which aims at providing an industry standard for an open architecture for distributed control units in automobiles. OSEK OS is a real-time operating system supporting multitasking for the use in automobiles. In order to validate the suggested approach, task management is selected as a key component composed of OSEK OS because it is highly related to providing the functions dealing with an asynchronous and concurrent execution of tasks. In OSEK OS, two different concepts of task are provided: a basic task and an extended task.

The differences between basic tasks and extended tasks are as follows. When a processor is terminated or the OSEK OS switches to a high-priority task, the basic tasks are terminated. In a waiting state, extended tasks release the processor, and the processor is assigned to a lower priority task without terminating the running.

The development of OSEK OS for automotive control unit application requires adjusting task periods to achieve the desired performance with respect to the response time at which a system task is completed. Also, it considers assisting the appropriate degree of architectural flexibility when introducing new operating system modules such as resource management, execution control, timing, communication, and fault tolerance.

The quality goals of OSEK OS development are summarized as follows:

- Portability: Replaced components should enhance the extended functionality of OSEK OS
- Performance: Response time of system tasks is less than or equals to 85 milliseconds (ms)
- Flexibility: Operational configuration should be set according to conformance class

As the architecture of OSEK OS refines, the number of end-to-end flows are increased, and each flow is expressed including the relevant components and subcomponents with high relevance.

A system component named OSEK_OS is composed of sub-elements such as OS_APIs, TaskMgmt, and EventCtrl. A representative call sequence circulates through OS_APIs, TaskMgmt, and EventCtrl. However, in a detailed perspective, several data structures such as TaskArray, ReadyTaskArray, RunningTaskArray, and EventArray are included in OSEK_OS. An exemplar flow from OS_APIs to TaskMgmt (TaskList) can be selected as a representative function that is related to a quality attribute, performance from a task management section.

For example, a flow sequence of a simple OSEK OS is specified as follows:

```
system implementation OSEK_OS.impl
subcomponents
task_schedule: process taskMgmt;
event_ctrl: process eventCtrl;
connections
C1: data port tID -> task_schedule.refTask;
C2: data port task_schedule.tID -> event_ctrl.assoc;
C3: data port event_ctrl.mask -> maskRef

flows
schedule_flow: flow path tID -> C1 ->
      task_schedule.schedule_flow ->
      C2 -> event_ctrl.control_flow -> C3;
control_flow:
.....
properties
  processing_time => 0 .. 20 ms
end system OSEK  OS.impl;
```

Figure 2. One Example of Flow Sequences in OSEK OS.

Two threads named TaskMgmt and EventCtrl are included in a component with an input port, tID and an output port, maskRef. The connections between two treads are linked through C1, C2, and C3 connections. The flow path in a component is organized to flow C1, a

schedule_flow in a TaskMgmt thread, C2, a control_flow in an EventCtrl, and C3.

Two quality attributes for performance and flexibility could be quantitatively measured, that come from the components involved in an architecture representation such as a flow specification. The architectural measurement is performed based on the specification in the architecture design phase before a system's detailed design is available. However, a broad architecture is not enough to provide information about the stimuli and responses for measuring the system's operations.

For deciding on the adequate level of architecture description, it is necessary to check whether the architecture elements adequate for high-prioritized quality attributes elicited from the utility tree exist or not. Then, whether suitable values for the quality attributes exist from the collection of historical data should be checked. If the values do not exit, quantitative techniques such prototypical implementation should be applied to acquire measured data for quality attributes.

In the architecture design of an automotive application domain, the architecture evaluation and selection considerably affects the manufacturing cost and qualities of automobile applications. Therefore, the available solutions for architecture analysis need to define, fit the evaluation the quality, and select the best appropriate item in terms of cost, quality, and technical constraints.

Regarding the performance, a real-time task and message design is necessary to evaluate a system's timing and representation in the point of an automobile architecture. However, the existing architecture analysis has a lack of modeling support for the analysis and annotation of scheduling characteristics. Also, the architecture description lacks sufficient semantics. The quality attributes at the highest level have been generally analyzed based on the architects' experiences. In this situation, the research of a tradeoff analysis among quality attributes can be quantitatively performed according to the quality attribute's timing values.

The flexibility of an architecture should be able to accept new and changed functionality. Based on this concept, flexibility measurement follows the quality attribute that decides upon the adaptation degrees for a functionality extension. The adaptation degrees for a functionality extension are based on the symptoms frequently occurring in the architectures. And the architectural symptoms for flexibility are separated into 5 levels. The definitions of the flexibility levels are summarized in the following Table 6.

TABLE VI. TECHNICAL METRIC FOR KEY ELEMENT CHANGES

Metric	Level Description
Level 1	Initial level
Level 2	Consistent function access through function APIs
Level 3	Access control of data structure for function adding
Level 4	Selection for operationally different mode
Level 5	Application-specific adaptation

TABLE VII. TRADEOFF ANALYSIS

Flexibility Level	Architecture	Flow	Response Time
1	Architecture 1	TaskMgmt.tID->TaskMgmt.scheduleFlow->TaskList.ref->TaskInf.in	20ms
2	Architecture 2	OS_APIs.itsTM->called->TaskMgmt.tID->TaskMgmt.scheduleFlow->TaskList.ref->TaskInfo.in->TaskInfo.Flow->optionBasic	50ms
3	Architecture 3	OS_APIs.itsTM->called->TaskMgmt.tID->TaskMgmt.scheduleFlow->TaskList.ref->TaskInfo.in->TaskInfo.Flow->optionExtended->ExecuteCtrl.startup->called->TaskMgmt.wait->refWait->EventCtrl.in->EventCtrl.flow->EventCtrl.eMask	60ms
4	Architecture 4	{Configurator.ready->setReadyR|Configurator.suspend->setReadyS}->TaskInfo.id->TaskInfo.flow->TaskList.in->TaskMgmt.tID->TaskInfo.scheduleFlow->TaskList.ref->TaskInfo.in->TaskInfo.Flow->optionExtended->ExtendedCtrl.startup->called	80ms
5	Architecture 5	Configurator.ecc->setECC->AppMode.flow->AppModel.enable->controlEvent->EventHandler.itsEC.OS_API.itsTM->called->TaskMgmt.tID->TaskMgmt.scheduleFlow->TaskList.ref->TaskInfo.in->{Configurator.priority->setPriority… ->EventCtrl.eMask	85ms

To define the flexibility levels, my experiences according to the scope of the changes inside of the component, or to

other components, are used, which represents indirectly how changing effects are distributed when component functions are changed. Based on this, the flexibility metrics may be designed with both flexibility symptoms and flexibility levels. Therefore, they may be different according to the views of quality concerns and scenarios from different situations.

The time measurement for an end-to-end flow is computed based on the processing time. In the case of a static call that is executed right after a call without delay, the response time is the processing time of a component. But, if there is delay for execution, the response time is calculated with the sum of the processing time and delay. The response time of the component should be calculated by summing together all the times consumed for processing, delay, and data transferring.

When analysis is focused on check whether an alternative architecture meets the desired goals, for example, the flexibility level is between 3 to 4, and the response time is deemed good in the case of minimum 20 ms and maximum 75 ms. In this case, architectures 1, 2, and 3 satisfy the response time within the scope from 20 ms to 60 ms, but the alternatives 1 and 2 show an inflexibility in the flexibility level. In the same analysis, alternative architectures 3 and 4 show a suitable flexibility level because they are in flexibility level 3 and 4. However, only architecture 3 shows that it satisfies both the flexibility and performance goals. In this way, it can be analyzed to know which architecture meets the desired goals with the quantitatively measured values for each quality attribute.

V. CONCLUSION

A suggested quality tradeoff analysis method is represented to be used for a quantitative measurement from extended AADL models. The measured data can be useful for an objective decision of architectural quality tradeoffs comparing with the existing an ad hoc architecture tradeoff analysis methods.

For tradeoff analysis, metrics for quality attributes are established and applied to measure each quality attribute with interest. For other quality attributes depending on different characteristics of a system, other metrics are generated and applied for analyzing a target quality attribute, which depends on the domain knowledge, historical data, and prototyping. How to acquire measured data and analyze the quality attributes depends on the domain knowledge, historical data, and prototyping included in quantitative analysis. To know and make a data representation in an architecture design, architects need to selectively develop prototyping within a minimum range.

The paper exemplarily details how to measure quantitatively quality attributes and use the measured data to analyze an architecture. In order to show the method, two kinds of quality attributes of flexibility and performance are chosen for a representative example, as this selection is in the same category with the case handling two or more quality attributes in semantics. The quality of flexibility with that of performance for a tradeoff analysis is connected. For a tradeoff analysis for quality attributes different with each other, metrics need to be devised and measurably analyzed based on a common architectural sub-element like a flow among components, as the case study shows.

A quantitative tradeoff analysis for a software architecture using AADL is generally provided with the concept from a studio project case for developing an automotive operating system.

The contribution of this paper can be mentioned in three aspects. First, the suggested approach is not ad hoc for an architecture tradeoff analysis, but is engineering disciplined. Second, the use of an extended AADL provides a parameter for a quantitative analysis of the quality attributes; it guides a detailed design from an abstract architecture design; and it facilitates the documentation of the quality attributes.

REFERENCES

[1] Rick Kazman, Mark Klein, Paul Clements, "ATAM: Method for Architecture Evaluation," Technical Report, CMU/SEI-2000-TR-004, August 2000.

[2] Len Bass, Paul Clements, Rick KAzman, Linda Northrop, Amy Zaremski, "Recommended Best Industrial Practice for Software Architecture Evaluation," CMU.SEI-96-TR-025, January 1997.

[3] Jakob Eyvind Bardram, Henrik Barbak Christensen, and Klaus Marius Hansen, "Architectural Prototyping: An Approach for Grounding Architectural Design and Learning," Proc. of the 4th Working IEEE/IFIP Conference on Software Architecture, pp. 15-24, June 2004.

[4] Peter H. Feiler, David P. Gluch, John J. Hudak, "The Architecture Analysis & Design Language (AADL): An Introduction," CMU/SEI-2006-TN-011, Feb. 2006.

[5] Taiga Nakamura, Vctor R. Basili, "Metrics of Software Architecture Changes based on Structural Distance," Proc. of the 11th International Symposium of Software Metrics, pp. 8-17, Sep. 2005.

[6] OSEK/VDX Operating System Specification Version 2.2.3, February 2005.