Dynamic Architectural Selection: A Genetic Algorithm Based Approach*

Dongsun Kim and Sooyong Park
Department of Computer Science and Engineering, Sogang University
1 Shinsoo-dong, Mapo-gu, Seoul, 121-742, Republic of Korea
{darkrsw,sypark}@sogang.ac.kr

Abstract

As the software industry is focusing on dealing with various requirements and environments, such as mobile and ubiquitous environments, software systems are increasingly undergoing many situational changes. These changes influence the quality of services that the software provides. Therefore, to maintain the performance of the software, it must be reconfigured. The reconfiguration is a complex problem if an application faces a large number of situations and has a number of software architectural instances. In this paper, we propose a novel approach to autonomous architectural selection in response to the current situation of various environments. This approach enables a software system to determine the best architectural instance for the current situation. To quickly find the best instance, we apply a genetic algorithm to the selection process. Further, we provide a performance evaluation to demonstrate that our approach efficiently find the best instance (or considerably good instance).

1. Introduction

User preferences, which represent the quality attributes that the user desires and their priority of quality attributes, differ for each user and change depending on each instance of usage during execution. Therefore, an application must contain different functions to suit different users' requirements; for example, if user A has more concerns about security, then the application should sacrifice usability and add more security functions to achieve a more secure execution, while user B desires more faster execution and the application must sacrifice usability and durability by removing rich user interfaces and so on.

In addition to the user requirements, the changing environments in which applications perform their tasks complicate the situation. For example, situation variance such as position, noise, light, battery level, and network bandwidth can influence mobile software applications. These factors can change the quality of applications. Therefore, the applications must modify their functionality in response to situational changes.

The goal of this study is to provide a method that autonomously selects an appropriate software architectural instance from the large set of candidates in response to situational and requirement changes at runtime. To describe this problem, we provide a motivating example that illustrates how the software architecture must be changed when situations and requirements change. Then, we precisely formulate the architectural selection problem using softgoal interdependency graphs [2]. To deal with this problem, this paper provides a novel approach to autonomous architectural selection using genetic algorithms [7]. This approach enables a software system to find an architectural instance that satisfies the current situation and user requirements within a short amount of time even if the application have a large number of candidate architectural instances. Further, this selection process attempts to find an optimal architectural instance for the situation and requirements.

The remainder of this paper is organized as follows. The next section provides a motivating example that requires autonomous architectural selection at runtime. Section 3 formulates the architectural selection problem based on softgoal interdependency graphs. In Section 4, we propose a genetic algorithm-based approach to autonomous architectural selection. Section 5 evaluates our approach in terms of its performance. Section 6 compares our approach to related work. Finally, Section 7 concludes the paper.

2. Motivating Examples

2.1. Situations, Quality Attributes and Functional Alternatives

Changing environments and user requirements affect applications. For example, mobile applications are exposed

*This research was performed for the Intelligent Robotics Development Program, one of the 21st Century Frontier R&D Programs funded by the Ministry of Knowledge Economy (MKE).

978-0-7695-3675-0/09 $25.00 © 2009 IEEE
DOI 10.1109/SSBSE.2009.11

to diverse factors such as the location where an application performs its function and the time when it performs its function. These factors lead to changes in application configurations. For example, when the user of an mobile application moves from indoors to outdoors, the noise level of the environment can change (usually, it gets louder). This change may have an impact on the performance of an application (e.g., an sound alert reminding about an appointment may not be effective because of the background noise). To handle these changes, we need to know which aspect from environments and user requirements affects applications and which element in an application is affected by that aspect.

First, a situational change in an environment is an influential aspect on an application. Changes in an environment can be represented by a set of several *situation variables* that represent situational aspects in the environment. For example, in mobile environments, the RSSI (Received Signal Strength Indication) level, battery level, and brightness are representative situational aspects that can affect the quality of service of applications. They can have ordinal, nominal, and numerical values, e.g., RSSI and battery level can have ordinal values while the brightness level has integer values of $[0, 255]$.

A change in situation variables represents a change of environment. For example, if the current value of the situation variable "RSSI level" changes from 1 to 5, we can assume that the environment of an application has changed. To specify the contextual change of the environment, it can have a vector of situation variables that may affect the application's performance. For example, suppose that an application A_1 may concern RSSI level, battery level, and brightness. A vector $< (rssi), (battery), (brightness) >$ can denote the contextual status of the environment (e.g., $< 0, 1, 220 >$ represents RSSI level = 0, Battery Level = 1, and Brightness = 220).

There are two types of user requirements: functional and non-functional requirements. Functional requirements (FRs) include the system behaviors that must be performed in the software system. On the other hand, non-functional requirements (NFRs)[1] address quality issues for software systems[2]. NFRs deal with the degree of satisfaction. For example, using an authentication method in an application may satisfy security requirements *at some level*. This concept is known as "satisficing" - *sufficiently satisfactory*. This term was used by Herbert Simon in the 1950s. In this paper, we deal only with changes of NFRs because applications tend to be required to change their functions at runtime according to the user's changing requirements concerning the quality of service rather than functional aspects.

To adapt to changing situations and quality attributes, a software system can have diverse alternative functions, e.g., "RichGUI," "SimpleGUI," and "NormalDisplay" as shown in Figure 1. These alternatives represent candidate functions that the application can take in situational and quality changes. Alternatives can be grouped by a type. For example, "HighContrastDisplay" and "NormalDisplay" alternatives belong to the same type, "Display" type. In each type, only one alternative can be activated (e.g., "NormalDisplay" and "HighContrastDisplay" cannot coincide).

Each alternative has relationships with quality attributes, as shown in Figure 1. For example, using a rich GUI (Graphical User Interface) may influence the application's usability and durability because the rich GUI can provide a better user experience and consume more battery life. This influence can be quantized. The quantization can describe the relationship more specifically. For example, the high contrast display alternative can have a positive impact on readability (denoted by "$+$") but a worse impact on durability (denoted by "$--$"). These impacts can be aggregated for each quality attribute as shown in Figure 1 (on the top of each quality attributes, responsiveness, usability, durability, and readability). Assume that the plus and minus signs denote "$+1$" and "-1," respectively. These aggregated scores can be used to measure how much the user is *satisficed*.

To simply measure the degree of satisfaction for quality attributes, we can integrate the scores as shown in Figure 1. Suppose that the user has the weight values (i.e., priority) of each quality attribute (0.2, 0.5, 0.1, and 0.2 for responsiveness, usability, durability, and readability, respectively). The weighted sum of the quality attributes is 0.8. This can be interpreted as the value of selected alternatives: Rich GUI, High Contrast Display, and Videotelephony.

The value of the selected alternatives can be a criterion to evaluate the selected alternatives to the current situation and the user's requirement represented by weight values. Therefore, we can identify the best combination of alternative selection by evaluating values of all possible combinations. For example, we can calculate 18 combinations of the alternatives shown in Figure 1 (three of GUI, two of display, and three of messaging alternatives) and we can find one combination that has the maximum value. This combination will provide the best user experience.

2.2. Situational and Quality Changes

The value of alternatives can be changed as the current situation changes. Value 0.8 in Figure 1 is valid in a situation where RSSI Level = 5, Battery Level = 3, and Brightness = 120. When the situation changes, the current value of the combination is not valid and it must be re-evaluated. If the situation changes to RSSI Level = 1, Battery Level = 1, and Brightness = 50, the value of the combination [RichGUI, HighContrastDisplay, and Videotelephony] can change to other values because the impact of alternatives on

[1] In this paper, we will also use **quality attributes** to represent NFRs as described in [2].

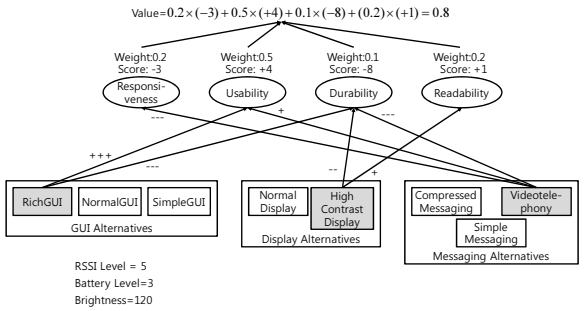

Figure 1. An example of value evaluation of selected alternatives.

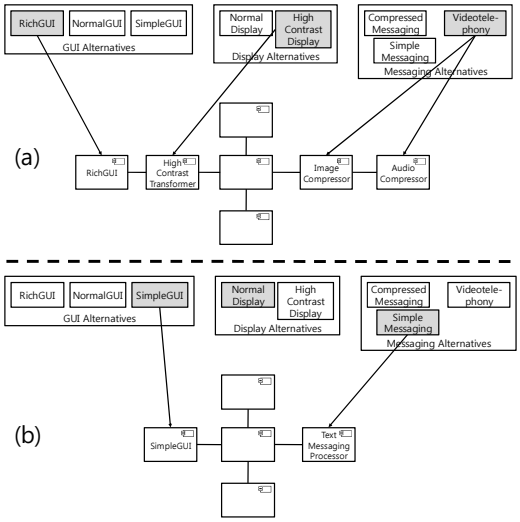

Figure 2. Examples of relationships between alternatives and an architectural configuration. (i.e., possible architectural instances)

quality attributes changes. For example, suppose that the mobile device is exposed to a low RSSI level. Then, the videotelephony function has a negative impact on responsiveness because it consumes more network resources than other alternatives. At this time, the application should select another alternative to provide a better quality of services.

Based on the changed impact values between alternatives and quality attributes, the system can re-evaluate the values of all combinations and select the best one. For every situational change, the system can re-evaluate all combinations to adapt to the current environment at runtime; however, this will be time-consuming if it involves a large number of alternatives. This time-consuming task may lead to a delay and performance degradation. Consequently, this can cause negative user experiences because the system cannot complete adaptation in time that the user can tolerate.

Similar to situational changes, the application must change its configuration when the user requirements related to quality attributes change. Although the application has the same alternatives and monitors the same situation values as those in Figure 1, the value of the selected alternatives can be changed due to the change of weight value for each quality attribute (i.e., the user may change weight values of quality attributes). For example, if the user changes weight values of quality attributes (responsiveness, usability, durability, and readability) shown in Figure 1 into $0.2, 0.2, 05$, and 0.1, respectively, the value of the selected alternatives is changed from 0.8 to -0.1.

The application must re-evaluate all combinations of alternatives to identify whether there are better alternatives that *satisfice* change user requirements. This is also time-consuming. However, it is not possible to calculate all values prior to runtime because the number of combinations of weight values and situation values is not finite (in particular, weight values are usually real numbers in $[0, 1]$). Therefore, the application should dynamically re-evaluate the values of combinations of alternatives to identify the best or better combinations in changing environments (i.e., at runtime).

2.3. Architectural Reconfiguration

After finding the best or better combination, the application must change its architectural configuration according to the selected combination. In other words, when the situation or user requirement changes, the application finds a combination of alternatives that are more appropriate for the current situation values and quality attributes, and then changes its architectural configuration according to the combination. For example, as shown in Figure 2.(a), "RichGUI" and "HighContrastDisplay" alternative can correspond to the "RichGUI" and "High Contrast Transformer" component, respectively. "Videotelephony" alternative can correspond to two components: "Image Compressor" and "Audio Compressor." If the application observes changes from the environment or user, it subsequently changes its configuration according to the selected alternatives, as shown in Figure 2.(b). These possible combinations are called *architectural instances* in this paper.

Deriving an actual software architectural configuration from a combination of architectural decision is also an important issue of software architecture research; however, this issue is not the focus of this paper and also previous studies have already dealt with this issue in terms of interface matching[10] and prescribed reconfiguration strategies[5]. In this paper, we assume that the application that applies our approach is implemented by dynamic architectures that enable the application to reconfigure its configuration.

3. Architectural Selection Problem

3.1. Quality Variables

In this section, we formulate the quality attributes using softgoal interdependency graphs (SIG) that are proposed by NFR (Non-Functional Requirements) framework[2]. A softgoal interdependency graph represents relationships between quality attributes (i.e., NFRs). A softgoal represents a quality attribute, and in an SIG, it is denoted by a cloud shape. Interdependency between two softgoals is denoted by a line connecting the two softgoals. By identifying softgoals and connecting them, an SIG represents the quality attributes of an application.

Representing quality attributes by an SIG gives several benefits to our approach. First, it helps a developer readily elicit quality attributes. The NFR framework proposed by Chung et al.[2] provides a means of identifying and analyzing quality attributes in detail. Second, this is a well-proven method for analyzing and representing quality attributes in several areas including software architectures with NFRs[9]. Third, this tree-like graph-based representation (i.e., SIG) can support the aggregation of impacts between functional alternatives and quality attributes in a bottom-up manner.

The quality attributes are formulated by a set of quality variables. Among softgoals of an SIG, only the highest quality attributes (e.g., readability, performance, durability, and usability in Figure 3) are considered to be quality variables because they will be the target of prioritization and value evaluation in our approach. A quality variable q_i can have a real number that describes how the application *satisfices* the quality attribute that the quality variable represents. A set of quality variables Q represents overall satisfaction of the user (i.e., it represents how much the user is satisficed). These quality variables should be aggregated to represent one integrated measure. A value (or utility) function U that measures the user's overall satisfaction is defined as:

$$\begin{aligned} U(Q,W) &= U(q_1, q_2, \cdots, q_n, w_1, w_2, \cdots, w_n) \\ &= q_1 \cdot w_1 + q_2 \cdot w_2 + \cdots + q_n \cdot w_n \\ &= \sum_i^n q_i \cdot w_i \end{aligned}$$

where Q is a set of quality variables, W is a set of weights, and each w_i is a weight of quality variable q_i, and n is the number of quality variables (i.e., $|Q| = n$). The sum of weights must be equal to 1 (i.e., $\sum_i^n w_i = 1$). The value of a quality variable is determined by the current situation and selected functional alternatives (to be described by the remainder of this section). The weights are defined by the user and they represent the priority of quality attributes. They can be changed by the user at runtime.

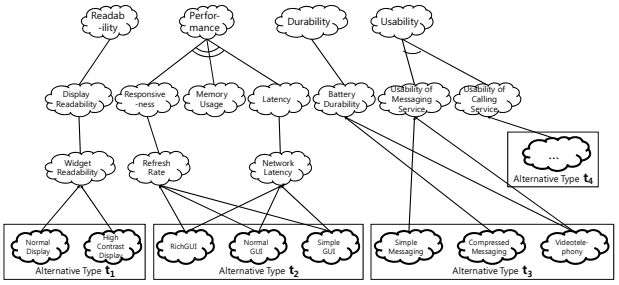

Figure 3. An example of softgoal interdependency graphs with alternative types.

3.2. Alternatives

Functional alternatives described in the previous section are denoted by operationalizing goals in our formulation. An operationalizing goal is introduced by NFR framework[2] to represent a design decision, e.g., simple, compressed, or videotelephony messaging services shown in Figure 2. An operationalizing goal in an SIG is denoted by a cloud shape with bold lines as shown in Figure 3.

An operationalizing goal can have an impact on softgoals. This relationship between an operationalizing goal and a softgoal is represented by *contribution*. An operationalizing goal contributes to one or more than one softgoals with some degrees as shown in Figure 1. In NFR framework, this relationship is quantized into some abstract notations such as "$--$ (= BREAK)" and "$+$ (= HELP)." In our approach, we do not use these notations; instead, we use situation evaluation functions (see Section 3.4) to represent the contributions of operationalizing goals.

In this approach, alternatives that have similar characteristics must be grouped by an alternative type as described in the previous section. Each type t_i can have only one value: one of the alternatives that constitutes the type t_i. In other words, alternatives that cannot coincide should be grouped as an alternative type. Each alternative should belong to only one alternative type. These alternative types are used to define architectural decision variables. An example of alternatives types is shown in Figure 3 (they are grouped by rectangles).

3.3. Architectural Decision Variable

An architectural decision variable determines part of an architectural configuration using an alternative type. In our study, one alternative type corresponds to one architectural decision variable as shown in Figure 4 (an alternative type t_i corresponds to an architectural decision variable a_i). In contrast to alternative types, architectural decision variables represent partial configurations of the ap-

plication. An alternative in an alternative type also corresponds to an architectural decision value ("NormalDisplay" and "HighContrastDisplay" are connected to $a_i = NORM$ and $a_i = HCONST$, respectively). An architectural decision value represents a partial set of components and their relationship between them as shown in Figures 2 and 4. This is formulated as follows. Suppose that a_i is an architectural decision variable. a_i can have an architectural decision value $v_i \in V$ where V is a set of architectural decision values. In this formulation, v_i must correspond to an alternative $i_j \in I$ where I is a set of alternatives. This mapping is one-to-one mapping and the number of architectural decision values (i.e., $|V|$) and the number of alternatives (i.e., $|I|$) must be equal. Also, an alternative type $t_i \in T$ must correspond to an architectural decision variable a_j and their cardinalities must be equal.

A combination of architectural decision variables comprises an architectural instance as shown in Figure 2. Let e_i be an architectural instance and it can be denoted by a vector of architecture decision variables, for example, $e_i =< a_1 = HCONST, a_2 = RichGUI, a_3 = \emptyset, \cdots, a_n = SIMPLEMSG >$. Let E be a set of possible architectural instances. We can formulate E as follows. Let $|a_i|$ be the number of architectural decision values that a_i can take. Let $|E|$ be the number of architectural instances that an application can have. Let n be the number of architectural decision variables, i.e., $|A| = n$. Then, we have the following:

$$\begin{aligned} |E| &= |a_1| \times |a_2| \times \cdots \times |a_n| \\ &= \prod_{i}^{n} |a_i| \end{aligned}$$

The number of possible architectural instances $|E|$ determines the complexity of the architectural selection problem.

3.4. Situation Variables and Functions

A situation variable s_i describes partial information of environmental changes (examples are shown in Figure 1). As described in Section 2, situation variables determine the impacts of architectural decision variables on quality attributes. To formally specify these impacts on quality attributes, we define a situation evaluation function (or situation function) as

$$\begin{aligned} f_{v_1}^{g_1}(S) &= f_{v_1}^{g_1}(s_1, s_2, \cdots, s_n) \\ f_{v_2}^{g_2}(S) &= f_{v_2}^{g_2}(s_1, s_2, \cdots, s_n) \\ &\cdots \\ f_{v_m}^{g_k}(S) &= f_{v_m}^{g_k}(s_1, s_2, \cdots, s_n) \end{aligned}$$

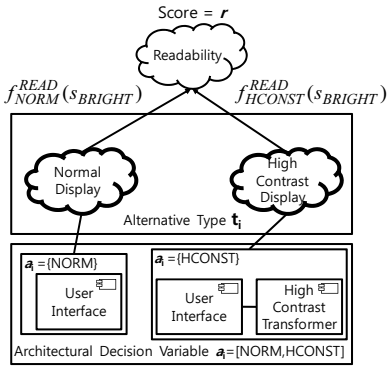

Figure 4. An example that shows interrelationships between architectural decision variables, alternative types, situation variables, and quality attributes.

where S is a set of situation variables, s_i is the *i*-th situation variable, v_j is the *j*-th architectural decision value, and g_k is the *k*-th softgoal. n is the number of situation variables and m is the number of architectural decision values. k is not the number of softgoals, but just an index of softgoals. A situation function is defined for each direct interdependency between an operationalizing softgoal and softgoal as shown in Figure 4. The number of situation functions is determined by the number of direct interdependencies between operationalizing softgoals and softgoals (i.e., the number of impacts) in the application (this is defined by an application developer who constructs the SIG of the application). Every operationalizing softgoal (which represents the related architectural decision value) must have at least one interrelationship with softgoals and the same number of situation functions; therefore, $|F| \geq |V|$ where $|F|$ and $|V|$ are the number of situation functions and the number of architectural alternative values, respectively.

3.5. Architectural Selection Problem

Based on the formulation described in the previous sections, this section describes the architectural selection problem in software systems at runtime. In this problem, quality variables are used to evaluate the user's satisfaction to the selected architectural instance, architectural decision variables are used to represent selected alternatives in this instance, and situation functions with situation variables are used to decide impacts of an architectural instance on quality attributes. An overall description of the architectural selection problem is shown in Figure 5.

The architectural selection problem is a combinatorial optimization problem[3]. In combinatorial optimization, one searches combinations in a problem space to find op-

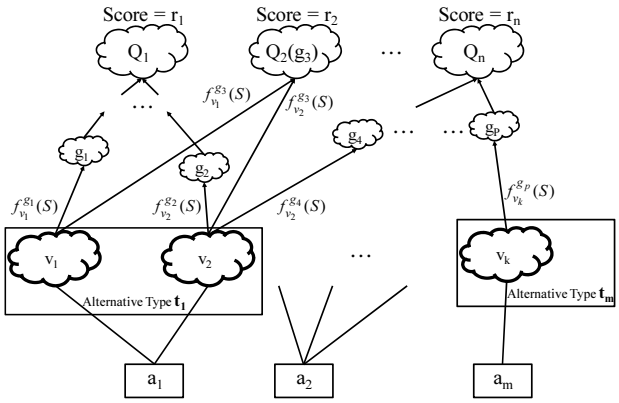

Figure 5. An overview of the architectural selection problem. ADV_i is the *i*-th quality variable, g_i is the *i*-th softgoal, v_i is the *i*-th functional alternative, a_i is the *i*-th architectural decision variable, and $f_{v_i}^{g_j}(S)$ is the situation function of alternative v_i and softgoal g_j.

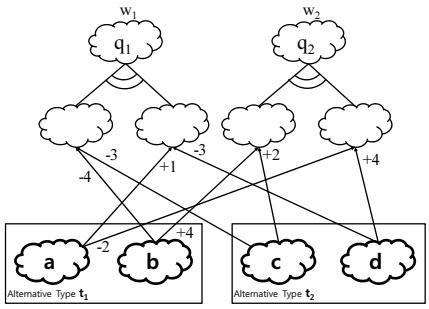

Figure 6. An example in which a greedy algorithm cannot find an optimal solution.

timal solutions according to specific evaluation criteria. In the architectural selection problem, the problem space comprises combinations of architectural decision variables (i.e., architectural instances) described in Section 3.3 and the evaluation criteria is the value (or utility) function described in Section 3.1. To calculate the result of the value function, an SIG and situation functions are required.

The goal of the architectural selection problem is to find an optimal architectural instance based on the current situation and the user's requirement represented by situation variables and quality variables (with weights), respectively. This is formulated as:

$$\begin{aligned} e^* &= \arg\max_{e_i \in E} U(Q, W) \\ &= \arg\max_{e_i \in E} U(Q_{SIG}(A), W) \\ &= \arg\max_{e_i \in E} U(Q_{SIG}(< a_1, a_2, \cdots, a_n >), W) \\ &= \arg\max_{e_i \in E} U(Q_{SIG}(e_i), W) \end{aligned}$$

where Q_{SIG} is an evaluation function based on an SIG and defined as $Q_{SIG} : A \rightarrow \hat{Q}$. $\hat{Q}$ is a set of vectors that comprise every quality variable q_i such as $< q_1, q_2, \cdots, q_m >$ where m is the number of quality variables, (i.e., $|Q| = m$). A is a vector of architectural decision variables, such as $< a_1, a_2, \cdots, a_n >$. This vector represents an architectural instance e_i. e^* is an optimal architectural instance in the current situation under the user requirements described by quality attributes.

To find e^*, value function $U(Q, W)$ must be evaluated through an SIG. In this approach, the evaluation process is conducted in a bottom-up manner from architectural decision variables to quality variables. For each instance e_i (i.e., a vector of architectural decision variables), we can directly derive a corresponding set of operationalizing softgoals (i.e., a set of alternative values). Then, we evaluate every situation function on edges, which are starting from the selected operationalizing softgoals, based on the values of the current situation variables. The results of situation functions are aggregated into related softgoals. This aggregation implies addition as shown in Figure 1.

After evaluating situation functions, the aggregated values are propagated to higher softgoals. There are three types of relationships between sub-softgoals and higher goals: direct, AND, and OR relation as shown in Figure 3. In a direct relation, the value of a subsoftgoal is directly propagated to the parent softgoal. In an AND relation, two or more than two subsoftgoals are related to a higher softgoal. An AND relation assumes the parent softgoal is satisficed if all subsoftgoals are satisficed. Therefore, every value of subsoftgoals is aggregated and the value of the parent softgoal is set to their sum if all subsoftgoals have positive values. If one subsoftgoals has a negative or zero value, the value of the parent softgoal is set to zero. Moreover, if all subsoftgoals have negative or zero values, the value of the parent softgoal is set to their sum. This rule is different from the rule provided by the NFR framework[2] (the framework only marks whether a softgoal is satisficed and unsatisficed) because it is important to measure how much the user is satisficed or unsatisficed in comparable numbers in the autonomous architectural selection problem.

In an OR relation, the parent softgoal is satisficed if, at least, one subsoftgoal is satisficed. The value of parent softgoal is determined as follows. When one more subsoftgoals have positive values, the parent softgoal takes the maximum value among the subsoftgoals; when all subsoftgoals have negative values or zero, the parent takes the minimum value among the subsoftgoals because the parent is definitely un-

satisficed; only one subsoftgoal has a positive value, the parent takes the value. Based on these propagation rules, the values of quality variables are determined.

The problem is that evaluating the value function for all possible architectural instances is time-consuming. The time complexity of this problem is determined by the number of architectural decision variables as discussed in Section 3.3. We can say that the time complexity is $O(|E|) = O(\delta^n)$ where $|E|$ is the number of architectural instances, δ represents the average number of architectural decision values in an architectural decision variable, and n is the number of architectural decision variables. This implies that an exhaustive search is not applicable for this problem (i.e., it may cause the state explosion problem). Therefore, we can apply other approaches such as greedy algorithms and dynamic programming. However, a greedy algorithm for this problem does not guarantee it will converge to an optimal solution. For example, with the SIG shown in Figure 6, the greedy algorithm cannot derive an optimal solution from the SIG. Specifically, we can determine the value of all individual operationalizing softgoals i.e., $a = -0.5, b = 0.0, c = -0.5$, and $d = 0.5$. Now, we know the best choice in each alternative type (b of t_1 and d of t_2). However, the best combination is choosing a and c. Dynamic programming can be applied to solve this problem, but it is also time-consuming (i.e., AND and OR relations in SIG do not allow substructure optimality). Therefore, it is required to provide a method to find an optimal solution for the architectural selection problem in a reasonable time span.

4. Genetic Algorithm-based Architectural Selection

4.1. Genetic Algorithm Procedure

A genetic algorithm[7] is a metaheuristic search method that approximates a solution in the solution space. It is also a well-proven method to deal with combinatorial optimization problems. In a genetic algorithm, the target problem is represented by a string of genes. This string is called a *chromosome*. Using the chromosome representation, a genetic algorithm generates an initial population of chromosomes. Then, it repeats the following procedure until a specific termination condition is met (usually a finite number of generations): (1) select parent chromosomes based on a specific crossover probability and perform crossover; (2) choose chromosomes and mutate the chromosomes based on a specific mutation probability; (3) evaluate fitness of offspring; (4) select the next generation of population from the offspring. In our approach, the above procedure will be adopted to solve the architectural selection problem. To do this, it is required to encode architectural instances into chromosomes, design the fitness function, and determine the crossover and mutation operators. The following sections will describe these issues.

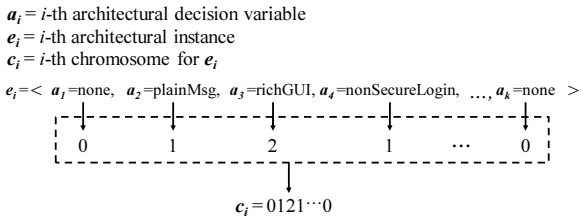

Figure 7. An example of encoding architectural decision variables into chromosomes.

4.2. Representing Architectural Instances in Genes

The main issue in applying a genetic algorithm to a certain application is encoding the problem space into a set of chromosomes. In this approach, we encode architectural instances into chromosomes because our goal is to find an optimal instance from the set of instances. We use architectural decision variables to encode instances as shown in Figure 7. The *i*-th architectural decision variable corresponds to the *i*-th digit of a chromosome. In this encoding method, the meaning of each number $0, 1, 2, \cdots, n$ is just an identifier to distinguish architectural decision values belonging to a specific architectural decision variable. Therefore, any discrete representation that can distinguish elements can be used, e.g., alphabetic representation such as $a, b, c, d, \cdots, z$.

4.3. Crossover and Mutation

Another issue of applying genetic algorithms is designing crossover and mutation operators to produce offspring. Among various crossover and mutation operators, we use two-point crossover and digit-wise probabilistic mutation.

Two-point crossover picks up two chromosomes and chooses two (arbitrary and same) positions for each chromosome. Then, it exchanges digits of the two chromosomes between two positions. We use this technique because it preserves more characteristics of parent chromosomes than other crossover techniques. Further, we assume that similar chromosomes may result in similar values to the value function. Crossover may pick up two parents in the population with crossover probability P_c. In other words, if $P_c = 0.5$, the half of the population is chosen as parents and the crossover operator produces the same number of offspring.

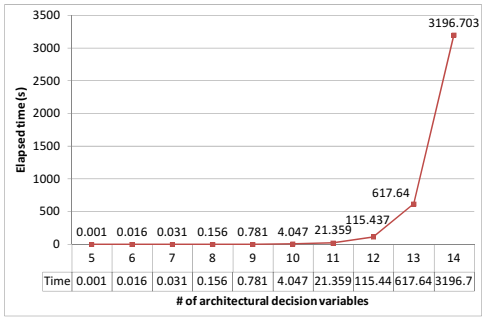

Figure 8. The performance of exhaustive search.

After performing crossover, the algorithm performs mutation. Every digit of offspring produced in the crossover step is changed to arbitrary values with mutation probability P_m. Note that if the mutation probability is too high, it cannot preserve the characteristics of parents. If the probability is too low, the algorithm may fall into local optima. Offspring produced by crossover and mutation are candidates for the next generation population.

4.4. Fitness and Selection

After performing crossover and mutation, the next step is selection. In this step, the algorithm evaluates the fitness values of all offspring and chromosomes that have better values survive. In this approach, the value (utility) function described in Section 3.1 is used as a fitness function to evaluate chromosomes and the tournament selection strategy[8] is used as a selection method. The tournament selection strategy selects the best ranking chromosomes from the new population produced by the previous steps.

The size of population determines the efficiency and effectiveness of genetic algorithms. If the size is too small, it does not allow exploring of the search space effectively, while too large a population may impair the efficiency. Practically, our approach samples at least $\delta \cdot l$ number of chromosomes where δ is the average number of alternative values for each architectural decision variable and l is the length of a chromosome.

By using the procedure described in the above sections, our approach can find the best (or reasonably good) solution from the search space when situation and requirement changes. The next section describes the result of performance evaluation.

5. Evaluation

This section provides the result of performance evaluation of our approach. First, the performance of exhaustive search has been measured to compare with the performance of our approach. Then, we have measured the performance of our approach based on the result of the previous experiment.

Every experiment was performed on SCH-V740 which is a cellular phone produced by Samsung. We implemented our approach based on Java. We designed an arbitrary SIG and gave a set of weight values to quality variables. Then, we produced architectural decision variables that have five architectural decision values on the average. For each architectural decision value, we give an arbitrary number of situation functions (i.e., impacts on softgoals). Those functions evaluate impacts on softgoals. With this setting, we conducted the following experiments.

5.1. Baseline

As a baseline, we have conducted an experiment that measures the performance of exhaustive search. In this experiment, we measured not only the elapsed time to exhaustively search every combination of architectural decision variables but also the best chromosome that will be used to evaluate the performance of our approach. As stated in Section 3.5, the architectural selection problem is a combinatorial optimization problem and has time complexity $O(\delta^n)$. The average number of architectural decision values in each architectural decision variable δ is five, therefore, the size of the search space is increasing as δ^n where n is the number of architectural decision variables. As shown in Figure 8, the device can search the problem space for an optimal architectural configuration in one second when $n < 10$. However, since $n = 10$, the elapsed time to search the problem space is exponentially increasing. The exhaustive search technique is not appropriate for the dynamic architectural selection problem because it is not acceptable for users to wait for the end of search for every moment when the current situation or the set of weights have been changed.

5.2. Performance of Our Approach

Based on the result of the previous experiment, we conducted three performance tests. The first test measures the elapsed time required to find an (near) optimal solution from the search space. It is difficult to anticipate the time required to find an optimal solution because genetic algorithms are randomized algorithms. However, we can consider a near optimal solution that is close to the best solution (such as the Las Vegas algorithm). In this test, we assume that the algorithm terminates when the difference of the elitist chromosome in the population and the best combination found in the previous experiment is smaller than 5% of the best combination (i.e., `if` $Fit(best) - Fit(elitist) < 0.05 \cdot Fit(best)$, `then` terminate the algorithm where $Fit(a)$

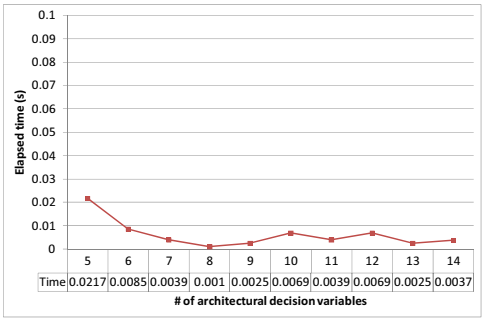

Figure 9. Elapsed time to obtain an (near) optimal solution for each number of architectural decision variables.

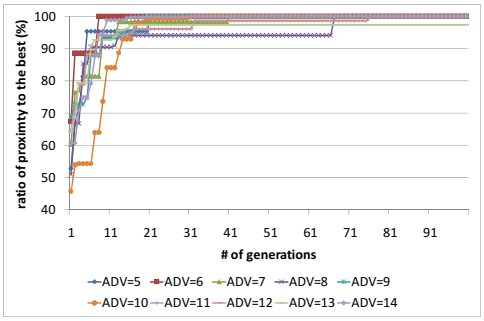

Figure 10. Ratio of proximity to the best (ADV = the number of architectural decision variables).

evaluates the value of chromosome a). As shown in Figure 9, the elapsed time to obtain an (near) optimal solution by our approach is very short compared to the time of exhaustive search (for each number of architectural decision variables, we ran ten tests and the elapsed time shown in the figure is the average value of ten tests). The elapsed time of the genetic algorithm does not increase as the number of architectural decision variables increases because it is basically a randomized algorithm as previously stated.

The next test is conducted to verify how fast the algorithm approaches the best solution. Like the Monte Carlo simulation, we fixed the number of generations and we recorded the value of the elitist chromosome for each generation. The result is shown in Figure 10. Note that y-axis represents the ratio of proximity to the best solution and the fixed number of generation is 100. For every number of architectural decision variable, the elitist chromosome quickly approaches to the best solution. In most cases, the elitist is the same before 40 generations. The approaching speed may vary for each run; however, it cannot influence the result that it finally approaches to the best solution. Further, the elapsed time for 100 generations is less than $0.01s$.

The third test shows the result of a larger number of architectural decision variables. For a larger number of architectural decision variables, it takes a very long time to find the best solution by exhaustive search. However, we can approximate that the elitist is the best or very close to the best by comparing with the elitist of previous generations. In other words, if the elitist has not been changed in a very long time, it may be the best with the high probability. In this test, the required number of generations is set to $10 \cdot l \cdot \delta$. Obviously, a large number of generations requires more time to perform the algorithm as shown in Figure 11; however, it is still much smaller than the time required to perform the exhaustive search. In practice, any time more than five or six seconds is sufficiently long for users to feel bored. Therefore, in this device, the application that uses our approach must control the number of architectural decision variable to be not over 40.

5.3. Analysis of Performance Evaluation

We have conducted a baseline experiment and three tests. The first test, which measured required time and generations to reach the best solution measured by the baseline experiment, indicates that the application can find the best combination of architectural decision variables and serve the new and best architectural instance to the user in response to every situation and requirement changes. However, practically, this type of execution cannot be applicable to real applications because it is impossible for the application to know the best solution for every situation and requirement changes. Therefore, we can fix the number of generations for each change. The second test evaluated this type of execution and shows the algorithm can find an optimal or near-optimal chromosome very fast. A fixed number of generations can be effective for a relatively small number of architectural decision variables; however, it may not be effective for a large number of them. The last test showed the result of the termination condition that finishes the algorithm when the elitist has not been changed for a specified number of generations. This type of execution can be applicable to practical systems because the number of generations proportionally varies as the number of architectural decision variables increases. In addition, even for a large number of architectural decision variables, the algorithm shows good performance compared to the exhaustive search.

6. Related Work

Floch *et al.* [4] proposed a utility-based adaptation scheme. This approach assumes that an adaptable application operates on the adaptation middleware that they have

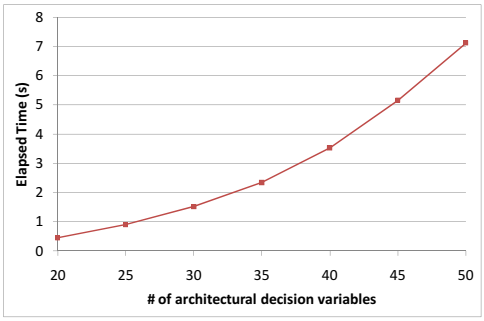

Figure 11. Required time to perform the tests to determine the elitist is the best (or very close to the best).

previously proposed[6], and the middleware monitors the current user context and system context. Based on the monitored context data, the middleware dynamically changes the application's configuration. Possible configurations are component types and their implementations. In this approach, planning is performed by mapping component implementations and properties or utility functions. When a user or system context changes, the middleware evaluates the change and compares it to the utility functions and properties of the current configuration. This adaptive planning effectively reflects contextual changes; however, they does not deal with the priority issue in which the user changes his or her preference about quality attributes.

Capra *et al*. [1] described the conflict problem in mobile applications. To deal with conflicts, they suggested a microeconomic mechanism that performs an (virtual) auction. In this mechanism, the mobile application is aware of the resource status of the device and user profiles. The application resolves intra and interprofile conflict problems using a game theoretic mechanism. Although this approach does not deal with architectural selection or adaptation issues, the idea, which autonomously selects functions of mobile applications at runtime, is related to our approach. The difference is that our approach focuses on the optimal architectural selection problem while their approach focuses on the conflict resolution.

7. Conclusions

As the software market extends its territory to mobile and ubiquitous environments, software systems are exposed to various situations and requirements. This requires the mobile application to change its architectural configuration in response to situation and requirement changes. This issue can be modeled by the architectural selection problem in which a mobile application searches its possible architectural instance and selects an optimal one to the current situation and user requirements.

In this paper, we illustrated a motivating example that requires architectural selection and formulated the architectural selection problem using softgoal interdependency graphs. Then, we proposed a novel approach to the architectural selection problem based on genetic algorithms. This approach enables a software system to autonomously search possible architectural instances (the search space) to find an optimal instance to the current situation and requirements within a short time. The evaluation of this approach showed our approach can accelerate the architectural selection of an application even if the application must search a considerable number of instances[2].

References

[1] L. Capra, W. Emmerich, and C. Mascolo. CARISMA: Context-aware reflective middleware system for mobile applications. *IEEE Trans. Software Eng.*, 29(10):929–945, 2003.

[2] L. Chung, B. A. Nixon, , E. Yu, and J. Mylopoulos. *Non-Functional Requirements in Software Engineering*. Kluwer Academic Publishers, 1999.

[3] W. J. Cook, W. H. Cunningham, W. R. Pulleyblank, and A. Schrijver. *Combinatorial Optimization*. Wiley, 1997.

[4] J. Floch, S. O. Hallsteinsen, E. Stav, F. Eliassen, K. Lund, and E. Gjørven. Using architecture models for runtime adaptability. *IEEE Software*, 23(2):62–70, 2006.

[5] D. Garlan, S.-W. Cheng, A.-C. Huang, B. R. Schmerl, and P. Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *IEEE Computer*, 37(10):46–54, 2004.

[6] S. Hallsteinsen, E. Stav, and J. Floch. Self-adaptation for everyday systems. In *WOSS '04: Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*, pages 69–74, New York, NY, USA, 2004. ACM Press.

[7] J. H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, 1975.

[8] B. L. Miller, B. L. Miller, D. E. Goldberg, and D. E. Goldberg. Genetic algorithms, tournament selection, and the effects of noise. *Complex Systems*, 9:193–212, 1995.

[9] N. Subramanian and L. Chung. An nfr-based framework for aligning software architectures with system architectures. In *Proceedings of the International Conference on Software Engineering Research and Practice & Conference on Programming Languages and Compilers, SERP 2006, Las Vegas, Nevada, USA, June 26-29, 2006, Volume 2*, pages 764–770, 2006.

[10] D. Sykes, W. Heaven, J. Magee, and J. Kramer. From goals to components: a combined approach to self-management. In *SEAMS '08: Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems*, pages 1–8, New York, NY, USA, 2008. ACM.

[2]Due to space limitation, we provide a supplemental material that provides discussion and future work of our approach. Visit this link: http://seapp.sogang.ac.kr:8080/discussion.pdf.