A Genetic Algorithm-based Approach to Dynamic Architectural Deployment

Dongsun Kim and Sooyong Park
Sogang University
Republic of Korea

1. Introduction

Increasing demands for various and complex tasks on contemporary computing systems require the precise deployment of components that perform the tasks. For example, in service robot systems (Hans et al., 2002; Kim et al., 2008) that have several SBCs (single board computers), users may simultaneously request several tasks such as locomotion, speech recognition, human-following, and TTS (text-to-speech). Each task comprises a set of components that are organized by an architectural configuration. These components execute their own functionality to provide services to the user. To execute components, they must be deployed into computing units that have computing power, such as desktops, laptops, and embedded computing units.

The deployment of components into computing units can influence the performance of tasks. If the system has only one computing unit, every component is deployed in the computing unit and there is no option to vary the deployment to improve the performance. On the other hand, if the system has multiple computing units, performance improvement by varying the deployment can be considered. Different instances of component deployment show different performance results because the resources of the computing units are different. Concentrated deployment into a certain computing unit may lead to resource contention and delayed execution problems. Therefore, the system requires an deployment method to improve performance when the user requests multiple tasks of a system that has multiple computing units.

When determining the deployment of components that comprise the architectural configuration for the tasks, it is important to rapidly and precisely make a decision about deployment. Since there are a large number of candidate deployment instances, even for a small number of computing units and components (i.e., their combinations exponentially increase), the deployment instance selection method must efficiently search for the best deployment instance that provides the most effective performance to the user. The exhaustive search method guarantees to search the best instance; however, it requires a long time for performing search. The greedy search method rapidly finds a solution; however, it does not guarantee to search the best instance.

This study proposes a genetic algorithm-based selection method that searches a set of candidate deployment instances for an optimal instance. This method repeatedly produces generations, and the solution found by the method rapidly converges to the best instance. This method more rapidly and precisely searches an optimal instance than the exhaustive search method and the greedy search method, respectively.

Source: Human-Robot Interaction, Book edited by: Daisuke Chugo,
ISBN 978-953-307-051-3, pp. 288, February 2010, INTECH, Croatia, downloaded from SCIYO.COM

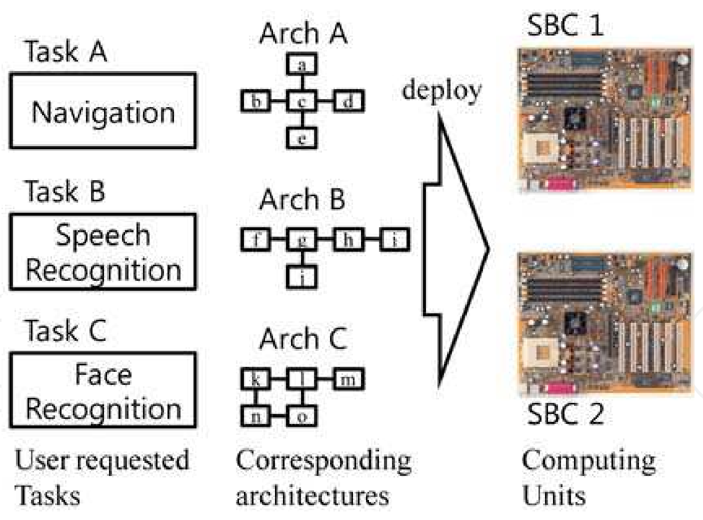


Fig. 1. An example of architectural deployment for required tasks.

This paper is organized as follows: Section 2 illustrates a motivating example that describes the dynamic architectural deployment problem. This problem is formulated as a multiplesack and multidimensional knapsack problem in Section 3. Section 4 describes a genetic algorithm-based approach to the problem. In Section 5, the proposed approach is evaluated in terms of efficiency and accuracy. Section 6 describes a case study conducted to show that our approach can be effectively applied to robot software systems. Section 7 compares our approach to related work. Section 8 provides two issues for discussion. Finally, Section 9 provides conclusion and suggests further studies.

2. Motivating example

Service robot systems such as Care-O-bot (Hans et al., 2002), and home service robot (Kim et al., 2008) have several computing systems termed single board computers (SBCs). An SBC has similar computing power to a desktop or laptop computer. Robot systems (especially service robots) perform their tasks by deploying robot software components into these SBCs, as shown in Figure 1. When a user wants to achieve a specific goal such as navigation, speech recognition, or more complex tasks, the user requests a set of tasks. For each task, the robot system derives its software architecture to handle the requested task. The robot system deploys these architectures to its SBCs to execute the tasks.
Even when the user requests the same tasks, the architectures that perform the tasks can be deployed in different ways. As shown in Figure 2, the consolidated architecture of the architectures shown in Figure 1 can be deployed into two SBCs in different ways. These different instances of deployment can exhibit different results. For example, if components, which consume more CPU time than other components, are deployed into one SBC, then they may lead to resource contention. Resource contention can cause execution delay during task execution. In addition, if two components that require extensive communication between them are deployed into two different SBCs, it may lead to performance degradation.
Performance degradation resulting from inappropriate architectural deployment may lead to more serious problems. For example, the path planning component in the robot navigation architecture and the image analysis component in the face recognition architecture highly consume CPU resources; therefore, if they are deployed into the same SBC, resource contention can occur. This does not allow for the allocation of sufficient CPU

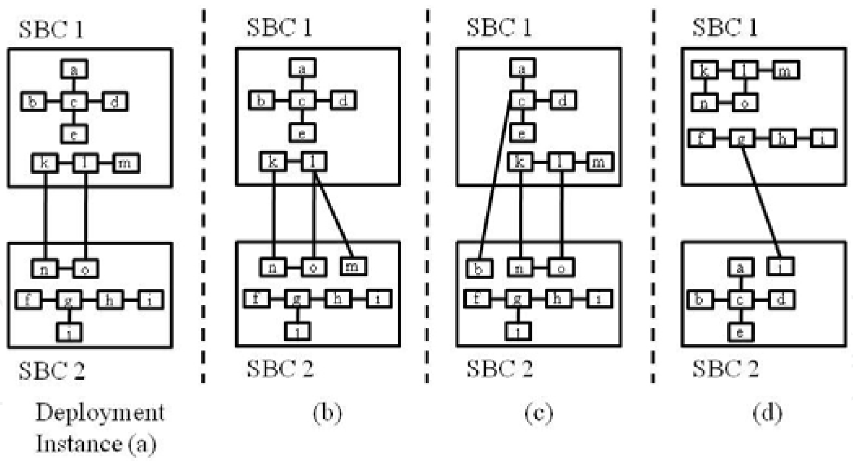


Fig. 2. Examples of architectural deployment instances.

time to the components; the path planning component is not able to respond in the required period of time, and the robot may then collide with the user or a wall.

In the face recognition architecture, the image preprocessing component and the image analysis component frequently interact with each other by passing a large amount of data. It would certainly lead to performance degradation if these two components were deployed into two different SBCs. Suppose that the user simultaneously requests a task that requires the location of the user's face (e.g., human-robot eye contact task). The delay resulting from the inappropriate deployment eventually leads to a malfunction, in which the robot cannot trace the user's face and the required task cannot be performed.

The abovementioned problems can occur when a software system uses multiple computing units (e.g., SBCs) and its elements interact with each other (e.g., software components) with a considerable amount of data. These problems can be serious if the system must deal with real-time tasks that have specific time constraints to achieve the goal of the tasks. This may not only degrade the quality of services but also result in failures of tasks that the user has requested.

To prevent these problems, a software system can search for appropriate deployment instances for every task requested by the user. However, the number of task combinations exponentially increases according to the number of tasks (the set of combinations can be defined by the power set of the set of tasks, i.e., 2^n where n is the number of tasks). Moreover, the number of possible deployment instances is larger than the number of task combinations (when the requested tasks have m components and the system has k computing units, the number of possible deployment instances is k^m where m is strictly larger than the number of tasks). Therefore, it is not efficient to exhaustively search the set of possible deployment instances for an optimal instance at run-time.

It is possible to determine optimal deployment instances for every possible task request prior to system execution, even though the number of possible task requests is large. If a developer has sufficient time to search for optimal deployment instances for every possible task request, then one can find them and record them to exploit them at run-time. However, this method is not applicable if a system has a large number of tasks and a large number of components belonging to the tasks. If the size of the task set is larger than 20 and the average size of the component set in a task request is larger than 30 or 40, it requires a very long time to search an optimal instance, even if it is conducted prior to runtime.

In addition to the size of task and component sets, the set of tasks and their architectural configurations can be dynamically updated at runtime. This implies that a developer should search the sets for optimal deployment again and update the result to the system. This may increase the development time and cost. Moreover, it is impossible to anticipate the configuration of every possible computing unit (e.g., the number of SBCs and their computing power) in operating environments. An operating environment can vary for every user. To deal with this problem, a method is needed to dynamically determine optimal deployment at runtime. This method should decide a deployment instance for every task request in a short time period to prevent the delay in task execution and search for a near-optimal instance.

3. Dynamic architectural deployment

This section formulates the optimal architectural deployment problem. This problem is defined by computing units, their provided resources, architectural configurations, and their required resources. This problem can be modeled by knapsack problems, in which computing units are sacks, components in an architectural configuration are items, and resource consumption efficiency is the value function. The remainder of this section describes the elements of this problem.

3.1 Computing unit

Every software system is executed on a specific computing unit, e.g., desktops, laptops, embedded systems, and other hardware systems that have computing power. When a software system performs its tasks in a computing unit, it uses resources that the computing unit provides, such as CPU time, memory space, and network bandwidth. In particular, software systems that are executed in embedded systems use dedicated resources. For example, in robot systems, the robot software uses sensors (e.g., laser range scanners, ultrasonic sensors, and touch sensors) and actuators (e.g., robot arms, wheels, and speakers). There are two types of resources: sharable and non-sharable.

A sharable resource suggests that a software component consumes a certain amount of the resource. In other words, one component cannot exclusively use the resource and shares the resource with other components. For example, a component consumes a certain amount of main memory space to record its necessary data and another component can simultaneously consume a certain amount of memory to perform its own tasks. However, if components attempt to consume more than the resource can provide, they can experience a resource contention problem, in which the components that request the resource compete for the resource and cannot access it when needed. This implies that the appropriate management of architectural deployment is required.

A non-sharable resource cannot be shared by several components. Only one component exclusively uses a non-sharable resource and other components that require the resource can use the resource only after the currently occupying component releases it. This type of resource is often installed in a specific subset of a computing unit. For example, in general, a robotic system has one wheel actuator and one arm manipulator. They are installed in a specific SBC, respectively (usually, these actuators are installed in separate SBCs). Therefore, components that use these actuators must be deployed in SBCs that have actuators that the components require[1].

[1] Remote invocation schema such as RPC (remote procedure call) and RMI (remote method invocation) can facilitate that the component can remotely exploit devices; however, this may lead to performance degradation due to communication burden.

These resources provided by computing units can be defined by the provided resource space that is the Cartesian product of the individual resource dimension r_j. For each computing unit O_i, its provided resource space P_i is formulated as

$$P_i \triangleq \bigotimes_{j}^{n} r_j$$

where n is the number of resources that the system can provide. The provided resource space represents the resource capability that a computing unit can provide.
A resource dimension r_j, which is a sharable resource, is denoted by r_j^s. It can have a certain integer value ($r_j^s \in [v_l, v_u]$ where v_l and v_u are the lower and upper bounds) that represents the maximum amount that the resource can provide. An instance of the j-th sharable resource dimension r_j belongs to the provided resource space of a computing unit and can have a value. For example, the main memory resource dimension of a system is denoted by r_{MEM}^s and its instance has a value of $[Mem_{MIN}, Mem_{MAX}]$ that represents the size of memory installed in the computing unit.
A resource dimension r_j, which is a non-sharable resource, is denoted by r_j^n and can have a Boolean value ($r_j^n \in \{0,1\}$) that represents whether the computing unit provides the resource. For example, the wheel actuator resource dimension of a system is denoted by r_{wheel}^n, and its instance has a value 0 or 1, where 0 represents that the computing unit does not provide the wheel actuator and 1 represents that it does.
The total provided resource space P_{tot} represents the resource capability of all computing units. This is formulated as

$$\begin{aligned} P_{tot} &\triangleq \bigcup_{i=1}^{m} P_i \\ &= < \sum_{i}^{m} r_1^i, \sum_{k}^{m} r_2^i, \cdots, \sum_{k}^{m} r_n^i > \end{aligned}$$

where m is the number of computing units, n is the number of resources, and $< r_1^i, r_2^i, \cdots, r_n^i >$ is the provided resource space of the computing unit O_i (i.e., P_i). The total provided resource space is used to determine whether the requested tasks are acceptable in terms of resource capability.

3.2 Architectural configuration

Software architectures represent structural information about the elements of a software system and their relationships. The software architecture of a system comprises software components (elements) that represent system functionality and connectors (relationship) that are responsible for providing a communication medium (Shaw & Garlan, 1996). In software architectures, a component provides an executable code that performs specific functions such as transforming a data set and processing user requests. A connector links two or more components and relays messages between the components. The software architecture organizes components and connectors into a software system.
In this formulation, a software architectural configuration denotes an instance of software architecture in a system. During system execution, components and connectors in an architectural configuration consume resources that the system provides. For example, a

component can consume CPU time, memory space, and embedded devices (such as wheel actuators in robotic systems) when it operates in the system. A connector consumes network bandwidth when two components (the connector interconnects) communicate to perform their functionality.
The resource consumption of components and connectors can be defined by the required resource space that is the Cartesian product of the individual required resources. It specifies the amount of resource consumption that a component or connector requires. The required resource space R_i of a component or connector c_i is defined by

$$R_i \triangleq \bigotimes_{j}^{n} r_j$$

where n is the number of resources and r_j represents the j-th resource dimension.
When the j-th resource dimension represents a sharable resource, its instance of the i-th component or connector c_i can have an integer value that represents the required amount of the j-th resource dimension r_j. This formulation assumes that the value represents the average consumption of a component or connector because real-time resource accounting imposes a severe overhead.
The total provided resource space R_{tot} represents the resource requirements of all components and connectors. This is formulated as

$$\begin{aligned} R_{tot} &\triangleq \bigcup_{i=1}^{m} R_i \\ &= < \sum_{i}^{m} r_1^i, \sum_{k}^{m} r_2^i, \cdots, \sum_{k}^{m} r_n^i > \end{aligned}$$

where m is the number of components and connectors, n is the number of resources, and $< r_1^i, r_2^i, \cdots, r_n^i >$ is the required resource space of a component or connector c_i (i.e., R_i). The total required resource space is used to determine whether the requested tasks are acceptable in terms of the resource capability that the system provides.

3.3 Dynamic architectural deployment problem

The dynamic architectural deployment problem can be formulated on the basis of information given in the previous sections. This problem is a combinatorial optimization problem (Cook et al., 1997) in which one searches the problem space for the best combination. Among the various types of combinatorial optimization problems, the dynamic architectural deployment problem can be modeled as a knapsack problem, in which one searches for the best combination of items to be packed in the knapsack. In architectural deployment, components are regarded as the items to be packed and the computing units are regarded as knapsacks that contain the items.
In particular, this problem is a 0−1 knapsack problem, in which items cannot be partially packed into the knapsack because components cannot be decomposed into smaller ones. Further, the dynamic architectural deployment problem has multiple computing units. This implies that the problem should be formulated as a multiple-sack knapsack problem. Additionally, this problem should optimize multidimensional resource constraints. Therefore, this problem is formulated as a multidimensional knapsack problem. Consequently, the

dynamic architectural deployment problem is formulated as a 0−1 multiple-sack multidimensional knapsack problem.
A knapsack problem comprises knapsacks, items, and cost/value functions. In dynamic architectural deployment, computing units in the system are knapsacks and components are items, as described in the previous paragraphs. A cost function decides that the selected items meet a certain constraints. In this problem, the cost function determines whether the selected combination in the knapsacks (i.e., the combination is a set of components in computing units) exceeds provided resources. A value function shows how valuable the selected combination in the knapsacks is. The value function of this problem represents the standard deviation of residual resources in all computing units.
The rationale of the value function formulation is based on the component execution pattern. As described in (Keim & Schwetman, 1975), some tasks can consume computing resources in a bursty manner. For example, the face recognition component of a human face-tracing task may temporarily use a large amount of network bandwidth when it requires the next scene of cameras. On the other hand, some other tasks consume computing resources in a steady manner, as when the localizer component of a navigation task continuously identifies the current position of a robot. The former type of resource consumption (bursty consumption) may particularly lead to performance degradation and execution delay problems mentioned in Section 2.
To prevent these problems, the resources of computing units should be managed to tolerate bursty resource consumption. This can be achieved by maximizing the residual resources of computing units. A computing unit can endure bursty consumption if it has sufficient residual resources. This assumption can be valid for sharable resources; however, for non-sharable resources, it is not useful for maximizing residual resources. To deal with this problem, it is assumed that the cost function determines whether the component can use the specified non-sharable resources. In other words, the cost function returns a positive integer if the computing unit O_k does not provide a non-sharable resource r_j (i.e., $r_j^k = 0$) when component c_i that requires a non-sharable resource r_j (i.e., $r_j^i = 1$); otherwise returns 0.
Another issue is the resource consumption of connectors. In this formulation, connectors are not items to be packed into knapsacks; however, they definitely consume resources such as network bandwidth between computing units. It is assumed that connectors consume computing resources, which are used for connecting components, only when the components are deployed into separate computing units. This type of resource consumption is dependently determined by component deployment.
On the basis of the above assumptions, the value function v can be defined by

$$v : A \to \mathbb{R}$$

$$\begin{aligned} v(a \in A) &= v(< c_1 = O_{c_1}, c_2 = O_{c_2}, \cdots, c_m = O_{c_m} >) \\ &= v(< O_{c_1}, O_{c_2}, O_{c_3}, \cdots, O_{c_m} >) \\ &= v(D_{r_1}(a), D_{r_2}(a), \cdots, D_{r_n}(a)) \\ &= \sum_i^n w_i D_{r_i}(a) \end{aligned}$$

where A is the set of architectural deployment instances, $\mathbb{R}$ is the real number set, c_i is the i-th component, and O_{c_i} is the computing unit, in which c_i is deployed. m is the number of components and *n* is the number of resource dimensions. w_i is the weight of the *i*-th resource dimension where $\sum_i^n w_i = 1$. The value function *v* returns the weighted sum of the residual amount of every resource dimension. Function D_{r_i} represents the residual amount of the resource dimension r_i. This function is defined as

$$D_{r_i} : A \rightarrow \mathbb{R}$$

$$\begin{aligned} D_{r_i}(a) &= D_{r_i}(< O_{c_1}, O_{c_2}, O_{c_3}, \cdots, O_{c_m} >) \\ &= std(d(r_i^1, a), d(r_i^2, a), \cdots, d(r_i^k, a)) \end{aligned}$$

where *std* is the standard deviation function, *k* is the number of computing units, and *d* is the residual resource function that returns the residual amount of resource of the *i*-the resource in computing unit *k* (i.e., r_i^k when the system selects deployment instance *a*).
The cost function *S*, which determines whether the deployment instance violates resource constraints, is defined as

$$S : A \rightarrow \{0,1\}$$

$$\begin{aligned} S(a) &= s(< O_{c_1}, O_{c_2}, O_{c_3}, \cdots, O_{c_m} >) \\ &= \bigcup_j^n s_{r_j}(a) \end{aligned}$$

where s_{r_j} is the function that determines whether architectural deployment instance *a* exceeds the amount of resource dimension r_j. The cost function returns 0 if no resource dimension exceeds, and 1 if at least one resource dimension exceeds. It determines the excess based on function s_{r_j} that is defined by

$$s_{r_j} : A \rightarrow \{0,1\}$$

$$s_{r_j}(a) = \begin{cases} 0 & \text{if } r_j^k \text{ of } O_k \text{ is equal or less than } r_j \text{ of } a, \forall k, \\ 1 & \text{if } \exists k, r_j^k \text{ of } O_k \text{ is larger than } r_j \text{ of } a \end{cases}$$

On the basis of the value and cost function described above, the goal of this problem is defined by

$$\text{find} \quad a^* = \underset{a \in A}{\arg\min}\, v(a)$$

$$\text{subject to} \quad S(a) \neq 1$$

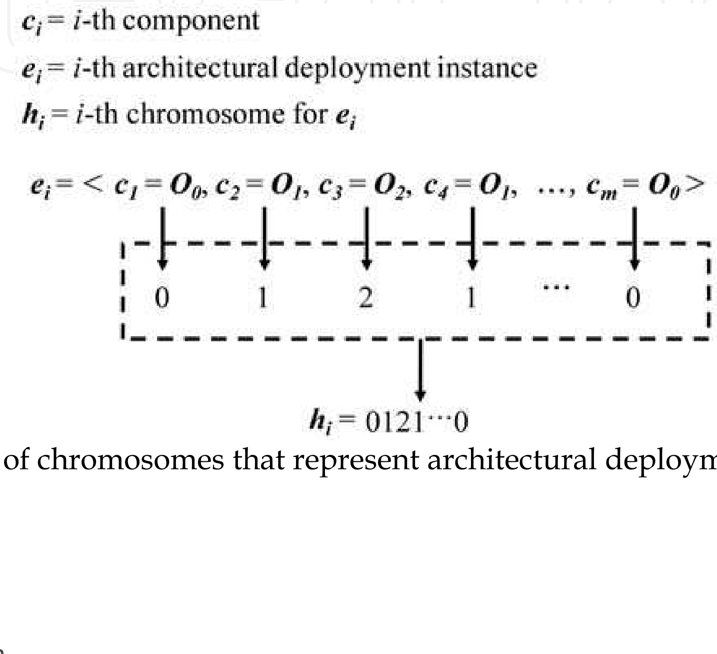
*a** is the best deployment instance based on the value function and *S*(*a*) ≠1 indicates that a deployment instance that violates resource constraints cannot be selected as an actual deployment instance in the system.
As described earlier, this problem is formulated as a 0-1 knapsack problem; however, in general, 0-1 knapsack problems are known as NP-hard problems (Kellerer et al., 2004). In particular, this is a multiple-knapsack, multidimensional, and 0-1 knapsack problem. Moreover, the dynamic architectural deployment problem requires both a better solution and faster search for the solution. To deal with this requirement in the present study, genetic algorithms are applied to the problem. The next section describes our approach in detail.

4. Genetic algorithm based dynamic architectural deployment

A genetic algorithm is applied to the dynamic architectural deployment problem in this study. A genetic algorithm (Holland, 1975) is a metaheuristic search method to approximate an optimal solution in the search space. This method is well known for dealing with combinatorial optimization problems. In a genetic algorithm, the target problem should be represented by a string of genes. This string is called a *chromosome*. By 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 the parent chromosomes on the basis of a specific crossover probability and perform the crossover; (2) choose and mutate the chromosomes on the basis of a specific mutation probability; (3) evaluate fitness of the offspring; and (4) select the next generation of population from the offspring.
In our approach, a genetic algorithmis used to search for an optimal architectural deployment instance. To apply the above procedure to the problem, architectural deployment instances are encoded into the chromosome representation, mutation and crossover operators for the chromosomes are determined, and the fitness function to evaluate the chromosomes is designed.

4.1 Representing architectural deployment instances in genes

It is important to encode the problem space into a set of chromosomes by a string of genes when applying a genetic algorithm to a certain application. In this approach, architectural deployment instances are encoded into chromosomes because our goal is to find an optimal instance from a set of instances. Components are matched to genes; each gene represents that the corresponding component is deployed into a specific computing unit by specifying the number of the computing unit.
For example, a deployment instance can be represented by a chromosome shown in Figure 3. c_i, e_i, and h_i represent the *i*-th component, deployment instance, and chromosome, respectively. In each instance, the *i*-th digit has a value that indicates the computing unit, in which the *i*-th component is deployed (i.e., the *i*-th digit of the chromosome is 1 if the *i*-th component c_i is deployed in computing unit O_1). When the required number of components in the task that the user requested is *m*, the length of the chromosome is *m*. The string of *m* digits is the *i*-th chromosome h_i of the *i*-th architectural deployment instance e_i. On the basis of the representation described above, our approach configures the search space of the dynamic architectural deployment problem.


c_i = i-th component
e_i = i-th architectural deployment instance
h_i = i-th chromosome for e_i
$e_i = < c_1 = O_0, c_2 = O_1, c_3 = O_2, c_4 = O_1, \dots, c_m = O_0 >$
0 1 2 1 ⋯ 0
h_i = 0121⋯0
Fig. 3. An example of chromosomes that represent architectural deployment instances.

4.2 Operators

As described earlier in this section, chromosomes that constitute the population (i.e., the search space) are reproduced by crossover and mutation operators. Therefore, designing these operators is an important issue in applying genetic algorithms. In this approach, two-point crossover and digit-wise probabilistic mutations are used.

The two-point crossover operator picks up two chromosomes as parents with crossover probability P_c and chooses two (arbitrary and same) positions of the parents. The operator exchanges digits between the selected positions of the parents. These new chromosomes are offspring. This technique is used because it preserves more characteristics of parent chromosomes than other crossover techniques (other crossover operators may exchange different digits of parents). Further, it is assumed that similar chromosomes may have similar results to the value function.

After producing offspring by the crossover operator, the algorithm should perform mutation. Every digit of offspring produced by the crossover operator 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 the parent chromosomes. On the other hand, if the probability is too low, the algorithm may fall into local optima. Offspring produced by crossover and mutation are candidates for the population of the next generation.

4.3 Fitness and selection

After performing crossover and mutation, the next step of our approach is selection. In this step, in general, a genetic algorithm evaluates the fitness values of all offspring, and chromosomes that have better values survive. In this study, the value function described in Section 3.3 is used as a fitness function to evaluate chromosomes, and the tournament selection strategy (Miller et al., 1995) is used as a selection method. The tournament selection strategy selects the best ranking chromosomes from the new population produced by crossover and mutation.

The cost function removes chromosomes that violate resource constraints and that the function specifies. In this approach, this filtering process is performed in the selection step. Even if the best ranking chromosomes have higher values than the value function, chromosomes that the cost function indicates as 1 should be removed from the population. On the basis of the value and cost functions, the approach selects the next population.

The size of the population is an important factor that determines the efficiency of genetic algorithms. If the size is too small, it does not allow exploring of the search space effectively. On the other hand, too large a population may impair the efficiency. Practically, this approach samples at least $k \cdot m$ number of chromosomes, where k is the number of computing units and m is the number of components that the task requires.

By using the described procedure, our approach can find the best (or reasonably good) solution from the search space when the user requests a set of tasks and the task requires a set of components. The next section describes the results of the performance evaluation.

5. Evaluation

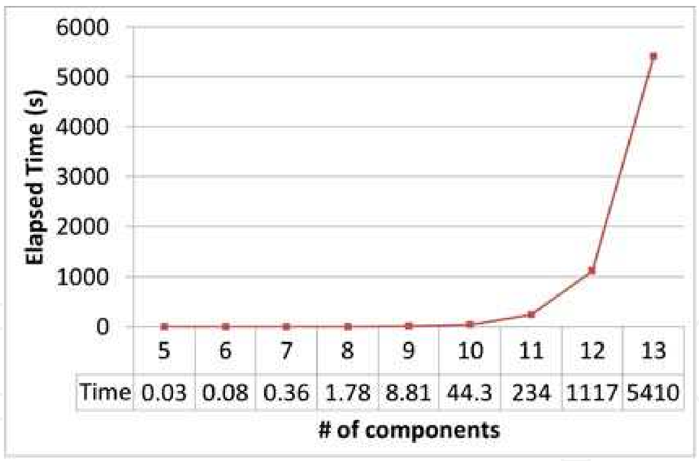
This section provides the results of the performance evaluation. First, the performance of exhaustive search and greedy search was measured. Then, the performance of our approach was measured and compared with the results of the earlier experiments.

Every experiment was performed on a set of desktops equipped with Intel Pentium Core 2 2.4 Ghz CPU and 2 GB RAM. Our approach was implemented using Java. A set of components and a set of five computing units were designed. The required resources of the components and the provided resources of the computing units were arbitrarily specified. The total required resources of the components (i.e., R_{tot}) did not exceed the total provided resources of the computing units (i.e., P_{tot}). Under these conditions, the following experiments were conducted.

5.1 Baseline

As a baseline, an experiment was conducted to measure the performance of exhaustive and greedy searches. In this experiment, the elapsed time of the exhaustive and greedy search was measured, and the best and greedy-best chromosomes (i.e., the best combination found by exhaustive search and the greedy combination found by greedy search, respectively) were determined. These results were used as a baseline to compare with the results of our approach. As stated in Section 3.3, the dynamic architectural deployment problem is a combinatorial optimization problem and has time complexity $O(k_m)$, where k is the number of computing units and m is the number of components requested by a task. In this experiment, there are five computing units k and the number of components n is varied from 5 to 13.

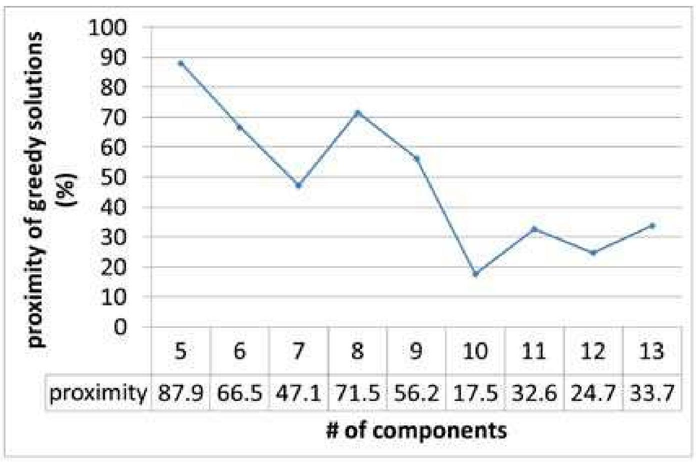
As shown in Figure 4, the exhaustive method searches the problem space in 10 seconds when m < 10. However, since m = 10, the elapsed time to complete searching the problem space exponentially increases. It is not always acceptable for the user to wait for the end of search because it may lead to user intolerance (Schiaffino & Amandi, 2004).



	5	6	7	8	9	10	11	12	13
Time	0.03	0.08	0.36	1.78	8.81	44.3	234	1117	5410

Fig. 4. The performance of the exhaustive search.

Greedy algorithms can be an alternative for reducing the search time to prevent user intolerance. In ubiquitous environments, a greedy algorithm is already adopted to determine better application combinations for a task requested by the user (Sousa et al., 2006). An experiment in which a greedy algorithm searches the same problem space for greedy solutions was also conducted. In every search from m = 5 to m = 13, the greedy search technique could find greedy solutions in 100*ms*. However, as shown in Figure 5, their quality decreases as the number of components increases. Therefore, an optimal solution cannot be expected using a greedy search.



	5	6	7	8	9	10	11	12	13
proximity	87.9	66.5	47.1	71.5	56.2	17.5	32.6	24.7	33.7

Fig. 5. The proximity of greedy solutions compared to the best solutions found in the exhaustive search.

5.2 Performance of GA-based approach

On the basis of the baseline of the previous experiment, three performance tests were conducted. The first test measured the elapsed time required to search an optimal or near-optimal combination of architectural deployment instances. It is difficult to anticipate the time required to find an optimal solution because genetic algorithms are randomized. However, a near-optimal solution that is close to the best solution (such as the Las Vegas algorithm) can be considered. In the first test, it is assumed that our approach terminates when the difference of the elitist chromosome in the population and the best chromosome (i.e., the best deployment instance found in the exhaustive search) is smaller than 5% of the best chromosomes.

In other words, `if` $Fit(elitist) - Fit(best) < 0.05 \cdot Fit(best)$, `then` terminate the search, where $Fit(a)$ evaluates the fitness (or value) of chromosome a.

As shown in Figure 6, the elapsed time to obtain an approximate chromosome by our approach is very short compared to the time for the exhaustive search. For every number of components, the elapsed time does not exceed 2 seconds and does not proportionally increase. This is because of the randomness of genetic algorithms, as previously stated. Even though this test shows that our approach can find a near-optimal solution in a short time, this type of approximation and termination condition is not feasible in practical systems because it is not possible for a system to anticipate the best solution before executing the search. Therefore, another termination condition is required.

The next test provides another termination condition that is similar to the Monte Carlo simulation (i.e., the termination condition specifies the fixed number of generations). This test was conducted to verify how fast our approach converges to the best architectural deployment instance when the fixed number of generations is used as the termination condition. In this test, the elitist chromosome was recorded for every generation and the results are shown in Figure 7. As shown, the number of generations is fixed as 300 and every search is finished in 500*ms*. Except for the case where the number of components is 12 and 13, our approach gradually converges to over 85% of the best solutions in 300 generations when the number of components is from 5 to 11. This implies that our approach can find near-optimal solutions in short generations. However, when the number of components is 12 and 13, the convergence of the solutions is under 65% of the best solutions. This implies that 300 generations are not sufficient to search the larger problem space of a large number of components.

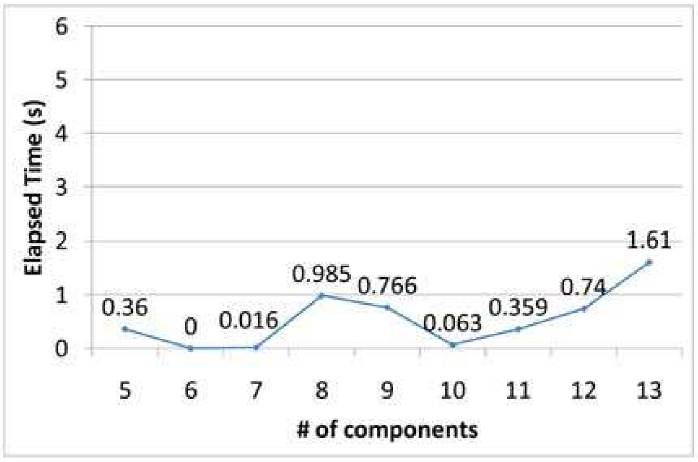


Fig. 6. Elapsed time to obtain an approximate chromosome for each number of components.

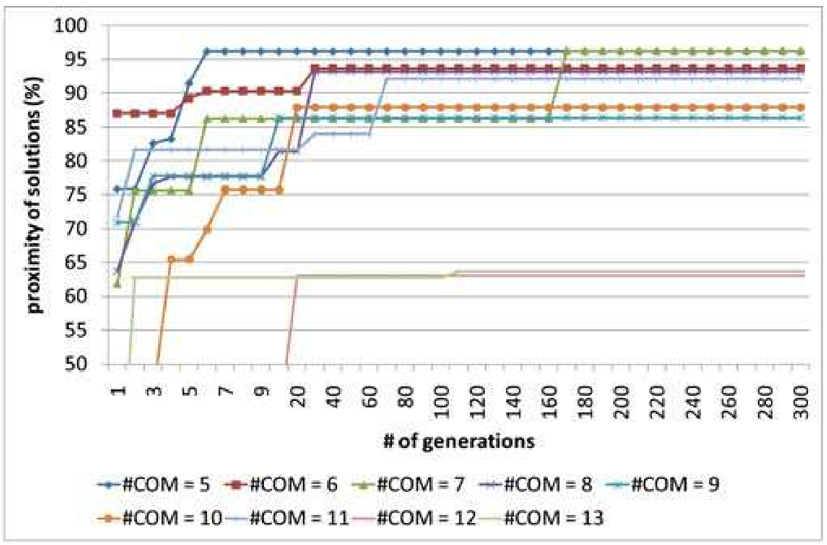


Fig. 7. Ratio of proximity to the best architectural deployment instance (#*COM*= *n* represents the number of components is *n*).

The third test provides an adaptive termination condition. When the number of components is larger than 12, using the fixed number of generation as a termination condition is not feasible; therefore, adaptively increasing the number of generations is applicable. However, simply increasing the number of generations may lead to the problem of the previous test. Hence, it is assumed that the elitist chromosome is the best chromosome with high probability if the elitist has not been changed for a long period. In this test, if the elitist has not been changed in $p \cdot k \cdot m$ generations, the search is terminated, where p is a constant, k is the number of computing units, and m is the number of components. The result of the test is shown in Figure 8 and we set p = 10. Each elapsed time in the figure is the average time of 10 test runs. As shown in Figure 8, when the number of components is 20 and 25, the elapsed time to determine that the elitist is the best chromosome (or very close to the best) is smaller than 5 seconds. When the number of components is from 30 to 40, the elapsed time is around 7 seconds. This implies that the required time to determine that the elitist is the best chromosome does not exponentially increase. Therefore, an optimal or near optimal architectural deployment instance can be found in a reasonable time with higher probability than the fixed number of generations, even if the best solution is not known. The constant *p* should be adaptively controlled, as a wait of more than 5 seconds is probably too long for some users.

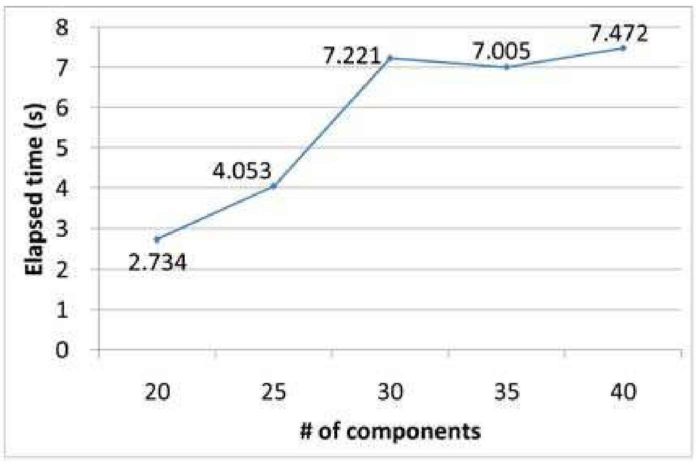


Fig. 8. Required time to perform the search to determine that the elitist is the best.

The evaluation results show that our approach provides optimal, or near-optimal, solutions (which are definitely better than the solutions found by greedy algorithms) in a reasonably short time (which is obviously faster than an exhaustive search). In addition, three different termination conditions are provided and their applicability is discussed. Consequently, the termination condition that adaptively controls the number of generations is practically applicable.

6. Case study

This section describes a case study conducted to show the applicability of our approach. The case study applies our approach to home service robots. In this case study, we assume that the task manager, which manages the robot's operation, requests five tasks to achieve a specific goal requested by the user. *Static deployment*, fixes the location of components that comprise the robot software architecture, obviously has problem. For example, all components in 'Object Recognizer' subsystem architecture must be deployed and executed in the Vision SBC that has cameras. This assumption looked reasonable when the architecture can dominate the SBC's resource. However, this is not appropriate because other tasks can share the resources of the SBC if the tasks *statically* assigned to be deployed into the SBC. Therefore, the system needs dynamic deployment. This case study provides three cases: 1) static deployment to verify the resource contention problem, 2) greedy dynamic deployment, and 3) genetic algorithm-based dynamic deployment.
We conducted the three cases under the following configuration: 1) Two SBCs; one is the Vision SBC that has cameras and pan-tilt gears which controls the robot's head, the other one is called the Main SBC that has wheels, microphones, laser range finders, and speakers. 2) Each SBC is equipped with 1GB of main memory and 2.2 GHz CPU. 3) Two SBCs are connected by 100 Mbps network.

First, we statically deployed autonomous navigator (has five components), TV program recommender (four components), arm manipulator (four components), interaction manager (seven components), and active audition planner (four components) into the Main SBC. At this time the Vision SBC has no resource consumption. Then, we focused on the navigator's behavior. The navigator must receive the robot's current pose (position and direction) and a map around the robot every 200ms to navigate safely. Since all components were deployed in a static manner, the navigator cannot have sufficient computing resource (especially CPU) and cannot retrieve a pose and a map every 200ms. This leads to wrong path planning. Moreover, the robot cannot reach the destination. This situation can be relieved by removing more than one architectures; however, if the goal of the user simultaneously requires all software components, the robot cannot achieve the goal.
To resolve the above problem, we applied dynamic deployment with greedy search. Based on the dynamic architecture reconfiguration framework in our previous research (Kim et al., 2006; Kim & Park, 2006) (Figure 9 shows screen shots of user interfaces in the framework and the robot using the framework), the robot searched for an appropriate deployment instance. The greedy algorithm has $O(nN)$ time complexity where n is the number of components to be deployed and N is the number of SBCs. In this case, we performed deployment 79 times with greedy search. The average time to search a deployment instance was 42ms. This overhead doesn't influence overall performance of the robot software system. However, the found deployment instance was far from the optimal solution. Therefore, we applied our genetic algorithm-based approach described in Section 4.

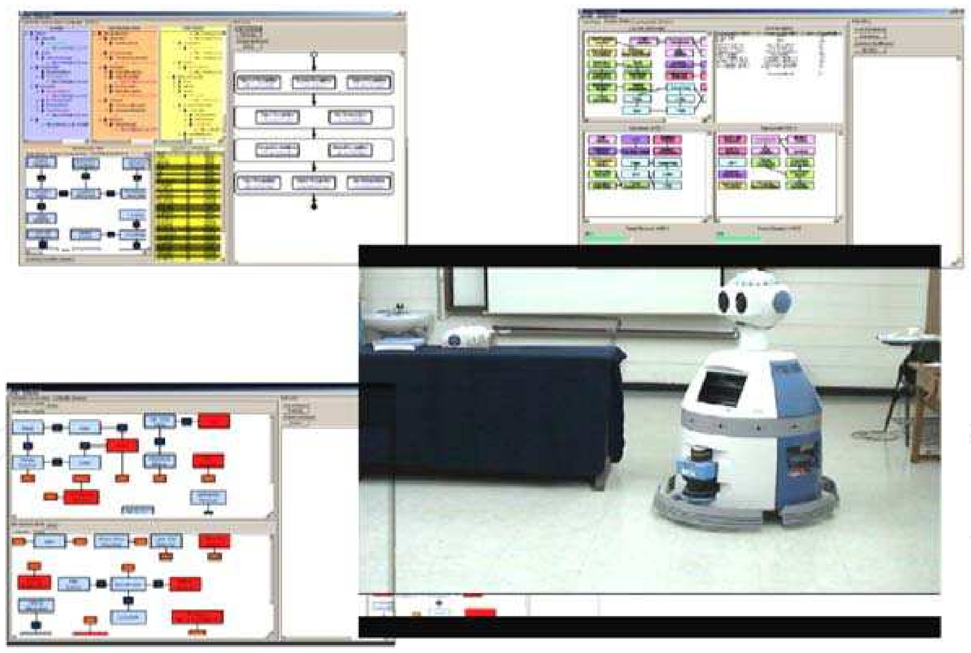


Fig. 9. A capture image of the case study

Using dynamic deployment with our approach, the robot could determine an efficient deployment instance within 3 ~5 seconds. It took more time than the greedy search, but our approach found better (near optimal) deployment instance as shown in Section 5. After deployment, the navigator could have sufficient resources and reliably retrieve the robot's pose and map. Other concurrent tasks were also performed effectively. Consequently, our approach enabled the robot system to perform its services more reliably.

7. Related work

Floch *et al*. (Floch et al., 2006) proposed a utility-based adaptation scheme. This approach assumed that an adaptable application operates on the adaptation middleware that was previously proposed (Hallsteinsen et al., 2004), and the middleware monitors the current user and system context. The system context includes computing resources such as CPU and memory. This approach selects an appropriate component for a specific component type on the basis of the user context (quality attributes) and system context (resource constraints). The differences between this approach and our approach are concerns about resource-efficiency and multiple computing units. This approach evaluates only whether the selected set of components exceeds the provided resources and assumes that the system provides only one computing unit.
Sousa *et al*. (Sousa et al., 2006) described the selection problem in which the system selects an appropriate application for a specific application type. This approach assumes that the user moves around various environments such as home, office, and park. The user needs the same context of his or her task wherever he or she moves. However, the computing power in each environment is different, e.g., a desktop at home or a handheld PC in the street. Therefore, for each environment, the systems must provide a different set of applications with the same context. This approach models the problem as a knapsack problem and solves it by using a greedy algorithm. The difference between Sousa's approach and our approach is the number of computing units and the selection method. This approach assumes that every system has only one computing unit. The greedy algorithm can find a solution in a short time; however, it cannot guarantee that the solution is the best or near-optimal solution.

8. Discussion

The first issue to discuss is the multiobjective (multidimensional) property of the dynamic architectural deployment problem. In general, a multiobjective problem can have a set of solutions that meets the requirements (i.e., Pareto set) (Das & Dennis, 1996). The dynamic architectural deployment problem can also have multiple solutions because it has multidimensional criteria. However, the goal of the problem is to search for only one optimal executable deployment instance rather than a set of possible solutions. In addition, the user or system administrator can specify the priority of dimensions by weight values and searching for an accurate Pareto set is time consuming. Therefore, evaluating the value of instance on the basis of the weighted sum of dimensions is practically applicable to this problem.
In the formulation, it is assumed that the constraint of non-sharable resources (i.e., the required resources of a component) can be acceptable if the computing unit provides the non-sharable resources. This assumption is acceptable only if the computing resource has a scheduling scheme for the non-sharable resource. Fortunately, most computing resources have specific scheduling schema, such as FIFO-like spooling for printer devices and round robin-like access mechanism for disk devices. Therefore, the assumption can be practically applicable to computing systems.

9. Conclusion and future work

The efficient deployment of components into multiple computing units is required to provide effective task execution, as the user simultaneously requests multiple and more complex tasks to the computing system. For example, service robots are requested to simultaneously perform several tasks and have multiple computing units. In such systems, inefficient deployment may lead to the malfunction of the system or the dissatisfaction of the user.

In this paper, a motivating example to deal with this problem was provided that described the problem in detail and formulated it as a multiple-sack multidimensional knapsack problem. To efficiently solve this problem, a genetic algorithm-based approach was proposed and the performance of the approach (efficiency and accuracy) was evaluated. The results of the performance tests demonstrated that our approach produced solutions more rapidly than exhaustive search and more precisely than greedy search methods.

Possible improvements to our approach include the extension of dimensions and parallel execution. Dimension extension implies that the problem formulate can add quality attributes to the problem space. Similar to computing resources, tasks may require a set of quality attributes, and components may provide various levels of quality. Further, some components may provide different quality levels in different computing units.

As stated in the formulation, our approach assumes that the system has multiple computing units. This indicates that our approach can be performed in parallel. The most time-consuming step is the selection process, and an individual chromosome evaluation by the value and cost function can be executed in independent computing units. Future work could focus on how the population of chromosomes can be efficiently divided.

10. References

Cook, W. J., Cunningham, W. H., Pulleyblank, W. R. & Schrijver, A. (1997). *Combinatorial Optimization*, Wiley.

Das, I. & Dennis, J. (1996). Normal-boundary intersection: An alternate method for generating pareto optimal points in multicriteria optimization problems, *Technical report*, Institute for Computer Applications in Science and Engineering.

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

Hallsteinsen, S., Stav, E. & Floch, J. (2004). Self-adaptation for everyday systems, *WOSS '04: Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*, ACM Press, New York, NY, USA, pp. 69–74.

Hans, M., Graf, B. & Schraft, R. D. (2002). Robotic home assistant care-o-bot: Past - present - future, *Proceedings of IEEE Int.Workshop on Robot and Human interactive Communication (ROMAN2002)*, pp. 380–385.

Holland, J. H. (1975). *Adaptation in Natural and Artificial Systems*, University ofMichigan Press, Ann Arbor.

Keim, J.W. & Schwetman, H. D. (1975). Describing programbehavior in a multiprogramming computer system, *ANSS '75: Proceedings of the 3rd symposium on Simulation of computer systems*, IEEE Press, Piscataway, NJ, USA, pp. 21–26.

Kellerer, H., Pferschy, U. & Pisinger, D. (2004). *Knapsack Problems*, Springer.

Kim, D. & Park, S. (2006). Designing dynamic software architecture for home service robot software, *in* E. Sha, S.-K. Han, C.-Z. Xu,M. H. Kim, L. T. Yang & B. Xiao (eds), *IFIP In- ternational Conference on Embedded and Ubiquitous Computing(EUC)*, Vol. 4096, pp. 437– 448.

Kim, D., Park, S., Jin, Y., Chang, H., Park, Y.-S., Ko, I.-Y., Lee, K., Lee, J., Park, Y.-C. & Lee, S. (2006). Shage: A framework for self-managed robot software, *Proceedings of Workshop on Software Engineering for Adaptive and Self-Managing Systems(SEAMS)*.

Kim, J., Choi, M.-T., Kim, M., Kim, S., Kim, M., Park, S., Lee, J. & Kim, B. (2008). Intelligent robot software architecture, *Lecture Notes in Control and Information Sciences* 370: 385– 397.

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

Schiaffino, S. & Amandi, A. (2004). User - interface agent interaction: personalization issues, *International Journal of Human-Computer Studies* 60(1): 129 – 148.

Shaw, M. & Garlan, D. (1996). *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall.

Sousa, J., Poladian, V., Garlan, D., Schmerl, B. & Shaw, M. (2006). Task-based adaptation for ubiquitous computing, *Systems,Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on* 36(3): 328–340.