Optimal component selection using a multiobjective evolutionary algorithm

Andreea Vescan

Abstract: Component selection is a crucial problem in Component-Based Software Engineering (CBSE) that is concerned with the assembly of pre-existing software components.

We are approaching the component selection involving dependencies between components. We formulate the problem as multiobjective, involving two objectives and one constraint. The approach used is an evolutionary computation technique. The experiments and comparisons with the Greedy approach show the effectiveness of the proposed approach.

Key words: Component Selection Problem, Evolutionary computation, Multiobjective.

Received: ??
Revised and accepted: ??

1. Introduction

Component-Based Software Engineering is concerned with designing, selecting and composing components [1]. As the popularity of this approach and hence number of commercially available software components grows, selecting a set of components to satisfy a set of requirements while minimizing a set of various objectives (as cost or the number of used components) is becoming more difficult.

In this paper, we address the problem of automatic component selection. Informally, our problem is to select a set of components from an available set which can satisfy a given set of requirements while minimizing sum of the costs of selected components. To achieve this goal, we should assign each component a set of requirements it satisfies. Each component is assigned a cost which is the overall cost of acquisition and adaptation of that component. In CBSE the construction of cost-optimal component systems is not a trivial task. It requires not only to optimally select components but also to take their interplay into account. In this respect we also use the dependencies between the requirements of the system.

In general, there may be different alternative components that can be selected, each coming at their own set of offered requirements. We aim at a selection approach that guarantees the optimality of the generated component system, an approach that takes into consideration also the dependencies between components.
neural network world 7/08, ??

(restrictions on how the components interact). The compatibility of components is not discussed here, it will be treated in a future development.

The paper is organized as follows: Section 2 starts with the problem formulation. Related work is discussed in Section 3. The proposed approach (that uses an evolutionary algorithm) is presented in Section 4. Greedy approach used for comparisons is described in Section 5. Some experiments and comparisons are performed in Section 6. We conclude our paper and discuss future work in Section 7.

2. Formal statement of the Component Selection Problem

A formal definition of the problem is as follows. Consider $SR$ the set of final system requirements (target requirements) as $SR = \{r_1, r_2, ..., r_n\}$ and $SC$ the set of components available for selection as $SC = \{c_1, c_2, ..., c_m\}$. Each component $c_i$ may satisfy a subset of the requirements from $SR$ denoted $SR_{c_i} = \{r_{i_1}, r_{i_2}, ..., r_{i_k}\}$. In addition $cost(c_i)$ is the cost of component $c_i$.

The goal is to find a set of components $Sol$ in such a way that to every requirement $r_j$ ($j = 1, n$) from the set $SR$ may be assigned a component $c_i$ from $Sol$ where $r_j$ is in $SR_{c_i}$, while minimizing $\sum_{c_i \in Sol} cost(c_i)$ and having a minimum number of used components.

To specify the requirements dependencies we use a dependency matrix. We are only interested in the provided functionalities (that are in the set of requirements $SR$ of the final system) of the components. We take into account only the dependencies between these requirements.

The dependencies specification table must contain dependencies between each requirement in the set of given requirements $SR$.

[htbp]

<table>
<thead>
<tr>
<th>Dependencies</th>
<th>$r_1$</th>
<th>$r_2$</th>
<th>$r_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r_1$</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>$r_2$</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>$r_3$</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

Tab. I Requirements dependencies specification table

In Table I we have specified the dependencies between the requirements $r_1, r_2, r_3$: the second requirement depends on the third requirement, the third requirement depends on the first and the second requirement. Some exceptional cases are required to be checked: no self dependency (the first requirement depends on itself), no reciprocal dependency (the second requirement depends on the third and the third depends on the second requirements), and no circular dependencies (the second requirement depends on the third, the first depends on the second and the third depends on the first). All the above situations are presented in Table I.
3. Related work

Component selection methods are traditionally done in an architecture-centric manner. An approach was proposed in [2]. The authors present a method for simultaneously defining software architecture and selecting off-the-shelf components. They have identified three architectural decisions: object abstraction, object communication and presentation format. Three type of matrix are used when computing feasible implementation approaches. Existing methods include OTSO [3] and BAREMO [4].

Another type of component selection approaches is built around the relationship between requirements and components available for use. A framework for the construction of optimal component systems based on term rewriting strategies is presented in [5]. By taking these techniques from compiler construction, especially optimizing code generation, they have developed an algorithm that finds a cost-optimal component system.

Paper [6] proposes a comparison between a Greedy algorithm and a Genetic Algorithm. The discussed problem considers a realistic case in which cost of components may be different. The selection function from the Greedy approach take into consideration both number of provided (offered requirements) of the components and the cost of the component.

Another type of component selection approaches is built around the relationship between requirements and components available for use PORE [7] and CRE [8]. The goal here is to recognize the mutual influence between requirements and components in order to obtain a set of requirements that is consistent with what the market has to offer.

In relation to existing component selection methods, our approach aims to achieve goals similar to [9], [10]. We are also interested in the relationship between components, their dependencies. The [9] approach considers selecting the component with the maximal number of provided operations. The algorithm in [10] consider all the components to be previously sorted according to their weight value. Then all components with the highest weight are included in the solution until the budget bound has been reached.

4. Proposed approach description

Evolutionary algorithms are a part of evolutionary computing, which is a rapidly growing area of artificial intelligence. Inspired by Darwin’s theory of evolution - Genetic Algorithms (GAs) are computer programs which create an environment where populations of data can compete and only the fittest survive, sort of evolution on a computer. They are well known suitable approaches for optimization problems.

The approach presented in this paper uses principles of evolutionary computation and multiobjective optimization [11]. First, the problem is formulated as a multiple objective optimization problem having two objectives: the total cost of the components used and the number of components used. Both objectives are to be minimized. Besides these, the dependencies are also treated, but are considered as constraints. In Subsection 4.1 more details are given.
There are several ways to deal with a multiobjective optimization problem. In this paper the weighted sum method \cite{12} is used.

Let us consider we have the objective functions \( f_1, f_2, \ldots, f_n \). This method takes each objective function and multiplies it by a fraction of one, the "weighting coefficient" which is represented by \( w_i \). The modified functions are then added together to obtain a single cost function, which can easily be solved using any method which can be applied for single objective optimization.

Mathematically, the new function is written as:

\[
\sum_{i=1}^{n} w_i f_i, \quad \text{where} \quad \leq w_i \leq 1 \quad \text{and} \quad \sum_{i=1}^{n} w_i = 1.
\]

In our case we have two objectives. Furthermore, the new function obtained by aggregating the two objectives can be written as:

\[
F(x) = \alpha \cdot f_1(x) + (1 - \alpha) \cdot f_2(x).
\]

4.1 Requirements execution order

We consider the requirements execution order as constraints: the requirements are given ordered such that the dependencies are satisfied.

Given the requirements dependencies table we have to compute the order of execution of the requirements such that the dependencies are satisfied.

The execution order based on dependencies are modeled as a Constraint Satisfaction Problem. Backtracking search, commonly used for solving Constraint Satisfaction Problems, is used to obtain all possible configurations of execution orders.

4.2 Solution representation

A solution (chromosome) is represented as a string of size equal to the number of requirements from \( SR \). The value of \( i-th \) gene represent the component satisfying the \( i-th \) requirement. The values of these genes are not different from each other (which means, same component can satisfy multiple requirements).

4.3 Genetic operators

The genetic operators used are crossover and mutation. Each of them is presented below.

4.3.1 Crossover operator

We use a simple one point crossover scheme. A crossover point is randomly chosen. All data beyond that point in either parent string is swapped between the two parents.

For example, if the two parents are: \( \text{parent}_1 = [8 0 6 8 0 6] \) and \( \text{parent}_2 = [8 2 6 8 6 9] \) and the cutting point is 3, the two resulting offspring are: \( \text{offspring}_1 = [8 0 6 8 6 9] \) and \( \text{offspring}_2 = [8 2 6 8 0 6] \).
We have stated above that also the dependencies between requirements are considering when constructing the solution but are considered as constraints. All the requirements must be previous sorted according to their dependencies from the dependencies specification table. All the obtain solutions (and also the individuals from the initial population) have the dependencies satisfied.

4.3.2 Mutation operator

Mutation operator used here consist in simply exchanging the value of a gene with another value from the allowed set. In other words, mutation of $i$-th gene consists in allocating a different component in order to satisfy the requirement $i$.

For instance, if we have the chromosome $parent_1 = [8 0 6 8 0 6]$ and we chose to mutate fifth gene, then a possible offspring can be $offspring_1 = [8 0 6 8 6 6]$.

4.4 Algorithm description

In a steady-state evolutionary algorithm one member of the population is changed at a time. The best chromosome (or a few best chromosomes) is copied to the population in the next generation. Elitism can very rapidly increase performance of GA, because it prevents losing the best found solution to date. A variation is to eliminate an equal number of the worst solutions, i.e. for each “best chromosome” carried over a “worst chromosome” is deleted.

The general pseudocode of the evolutionary algorithm used in this paper is given in the Algorithm 1.

<table>
<thead>
<tr>
<th>Algorithm 1</th>
<th>Evolutionary algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Require: SR; {set of requirements}</td>
<td></td>
</tr>
<tr>
<td>SC. { set of components }</td>
<td></td>
</tr>
<tr>
<td>Ensure: Sol. { obtained solution }</td>
<td></td>
</tr>
<tr>
<td>1: Population Initialization;</td>
<td></td>
</tr>
<tr>
<td>2: for ( $t = 1$ to numberOffEvolutions) do</td>
<td></td>
</tr>
<tr>
<td>3: for ( $k=0$ to numberOffIndividualsInPopulation; $k += 2$) do</td>
<td></td>
</tr>
<tr>
<td>4: Randomly choose two individuals, $p_1$ and $p_2$;</td>
<td></td>
</tr>
<tr>
<td>5: OneCutPointCrossover for $p_1$ and $p_2$, resulting child $c_1$ and child $c_2$;</td>
<td></td>
</tr>
<tr>
<td>6: Mutation($c_1$);</td>
<td></td>
</tr>
<tr>
<td>7: Mutation($c_2$);</td>
<td></td>
</tr>
<tr>
<td>8: Memorize in $c_1$ the best individual from $p_1$ and $c_1$ and in $c_2$ the best individual from $p_2$ and $c_2$;</td>
<td></td>
</tr>
<tr>
<td>9: Replace the two worst individuals from population with $c_1$ and $c_2$.</td>
<td></td>
</tr>
<tr>
<td>10: end for</td>
<td></td>
</tr>
<tr>
<td>11: end for</td>
<td></td>
</tr>
</tbody>
</table>
5. Greedy approach

Greedy techniques are used to find optimum components and use some heuristic or common sense knowledge to generate a sequence of sub-optimums that hopefully converge to the optimum value. Once a sub-optimum is picked, it is never changed nor is it re-examined.

A Greedy algorithm proceeds as follows: initially the set of chosen objects is empty; the selection function removes an object from the set of available objects; the new enlarged set is checked to see if the enlarged set is a solution; if the enlarged set is no longer feasible, the object is discarded and never considered again; the discarded object is not put back into the set of available objects; if the enlarged set is feasible it is permanently added to the chosen set. The process repeats itself picking a sequence of sub-optimums until either a solution is found or it is shown that no solution is feasible. The pseudocode of the Greedy algorithm is illustrated in Algorithm 2. We use the problem notation from Section 2.

**Algorithm 2** Greedy algorithm

| Require: | SR; \{set of requirements\} | SC. \{ set of components \} |
| Ensure: | Sol. \{ obtained solution \} |

1: Sol := \emptyset;
2: RSR := SR; \{RSR=Remaining Set of Requirements\}
3: while (RSR <> \emptyset) do
4: Choose $c_i$ from $SC$, not yet processed;
5: @ Mark $c_i$ as processed.
6: if $Sol \cup c_i$ is feasible then
7: Sol := Sol $\cup \{ c_i \}$;
8: RSR := RSR - $SR_{c_i}$;
9: end if
10: end while

The selection function is usually based on the objective function. We consider the proportion of number of requirements satisfied to the cost of the component as a measure to maximize our heuristic decision:

$$|SR_{c_i} \cap RSR|/cost(c_i) \text{ is maximal,}$$

and also the number of dependencies of the component $c_i$ considered \textit{is minimal}. All the dependencies of the selected $c_i$ component must be satisfied, i.e. from the dependencies specification table of $c_i$ the values of the columns that are checked must have been already considered into the solution.

6. Experiments and comparisons

A short and representative example is presented in this section. Starting for a set of six requirements and having a set of ten available components and the dependencies between the requirements of the components, the goal is to find a subset of the given components such that all the requirements are satisfied.
Andreea Vescan: Optimal component selection using a multiobjective evolutionary algorithm

The set of requirements \( SR = \{r_0, r_1, r_3, r_4, r_5\} \) and the set of components \( SC = \{c_0, c_1, c_2, c_3, c_4, c_6, c_7, c_8, c_9\} \) are given.

In Table II the cost of each component from the set of components \( SC \) is presented.

<table>
<thead>
<tr>
<th>Comp</th>
<th>( c_0 )</th>
<th>( c_1 )</th>
<th>( c_2 )</th>
<th>( c_3 )</th>
<th>( c_4 )</th>
<th>( c_5 )</th>
<th>( c_6 )</th>
<th>( c_7 )</th>
<th>( c_8 )</th>
<th>( c_9 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cost</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>9</td>
<td>6</td>
<td>14</td>
<td>15</td>
<td>14</td>
<td>7</td>
<td>14</td>
</tr>
</tbody>
</table>

**Tab. II** Cost values for each component in the \( SC \)

Table III contains for each component the provided services (in terms of requirements of the final system) and Table IV the dependencies between each requirement from the set of requirements.

**Tab. III** Requirements elements of the components in \( SC \)

**Tab. IV** Specification Table of the Requirements Dependencies

The configurations of the requirements execution order for the above problem (computed with the Backtracking search) are stated in Table V.

**Tab. V** Requirements execution order - all possible configurations based on dependencies

### 6.1 Results obtained by the Greedy algorithm

In the current section we discuss the application of the Greedy algorithm presented in Subsection 5. The first step of the selection function is the computation of the
proportion of number of requirements satisfied to the cost of the component. The component with the maximum ratio is chosen to be a part of the solution but only if it has no dependencies.

In the first iteration of the algorithm the component is the only component having no dependencies. So, the first component included into the final solution is component (with all its provided services, i.e. \( r_1 \), required by the final system).

The second choice from the set of remaining set of requirements is the first component \( c_0 \): the computation of the new ratios shows that there are only three components with all the dependencies satisfied: \( c_0 \), \( c_2 \) and \( c_3 \). The \( c_0 \) component has the maximum ratios, i.e. 0.250.

Considering the new ratios and the dependencies, three components have the dependencies satisfied: \( c_6 \), \( c_7 \) and \( c_9 \). Only two (\( c_7 \) and \( c_9 \)) of them have the maximum ratios, i.e. 0.142 and one of them is chosen randomly by the program (\( c_7 \)).

In the set of remaining set of requirements that must be fulfilled there is only one element, \( r_2 \). All the three components remained in the “competition” have the dependencies satisfied but only the component \( c_1 \) and \( c_8 \) have the maximum ratios, i.e. 0.142 (component \( c_5 \) has the value 0.071). Randomly, the \( c_1 \) component is selected.

The set of the remaining set of requirements is empty and we have reached a solution with all the requirements satisfied by the selected components: \( c_4 \), \( c_0 \), \( c_7 \) and \( c_1 \). The cost of the final solution 35 is the sum of the cost of the selected components. A similar solution may be found if at the previous steps the random choice would have selected the \( c_8 \) component, and in this case the cost of the final solution would have been 35. Still, we will see that there is a better solution with only three components and having the final cost 28 (\( c_0 \), \( c_6 \) and \( c_8 \) components).

6.2 Results obtained by the Evolutionary approach

The parameters used by the evolutionary approach are as follows: population size 20, number of iterations 10, mutation probability 0.7, crossover probability 0.7.

The value of \( \alpha \) used while aggregating the objectives was set to 0.5 which gives the same importance to both objectives but we will also show the results obtained for other different values such as: 0.0, 0.3, 0.7.

The proposed approach was tested for more sets of parameters but we are providing in this paper the results for just one set of parameters. The results obtained with different sets of parameters were very similar with the stated set of parameters. We have considered two configurations from all possible obtain configurations of the requirements execution order: the seventh solution and the tenth solution from Table V.

The algorithm was run 100 times and the best, worse and average fitness values were recorded. The evolution of the fitness function for all 100 runs using the value \( \alpha = 0.5 \) is depicted in Figure 1 and Figure 2. Best, worse and average fitness value recorder for each run are presented.

The evolution of fitness function (recorded for the best individual in each run) for \( \alpha = 0.0 \), \( \alpha = 0.3 \) and \( \alpha = 0.7 \) is depicted in Figures 3, 5, 7 (Dependency
Andreea Vescan: Optimal component selection using a multiobjective evolutionary algorithm

number 7) from Table V)) and in Figures 4, 6 and 8 (Dependency number 10) from Table V). We should mention that the best, worse and average values of the fitness function have the same value in almost all the runs except for very few of them.

Running the algorithm we could find more solutions having the total cost 35 than the Greedy approach: [4 0 9 1 0 1], [4 9 9 1 0 9], [8 2 7 8 0 7], [4 0 5 5 0 1], [4 2 6 8 6 1], etc.

![Fig. 1](image1.png)

*Fig. 1* The evolution for independent runs of fitness function (best, worse and average) for $\alpha = 0.5$ for the seventh solution from Tab. V

![Fig. 2](image2.png)

*Fig. 2* The evolution for independent runs of fitness function (best, worse and average) for $\alpha = 0.5$ for the tenth solution from Tab. V

A best solution was also found: [8 2 6 8 6 6] and a worst solution [4 2 7 8 0 9].

How the best solution could be reached is discussed in the following (cutting point is 3): selection: $p_1 = [8 0 6 8 0 6]$ and $p_2 = [8 2 6 8 6 9]$, crossover: $o_1 = [8 0 6 8 6 9]$ and $o_2 = [8 2 6 8 0 6]$ and mutation: new $o_1 = [8 0 6 8 6 6]$ (the last gene is mutated from 9 to 6) and new $o_2 = [8 2 6 8 6 6]$ (the previous to the last gene is mutated to 6).

How the worst solution could be reached is discussed in the following (cutting point is 3): selection: $p_1 = [4 2 7 1 0 1]$ and $p_2 = [8 0 9 8 0 1]$, crossover: $o_1 = [4 2 7 8 0 1]$ and $o_2 = [8 0 9 1 0 1]$ and mutation: new $o_1 = [4 2 7 8 0 9]$ (the last gene is mutated from 1 to 9) and new $o_2 = [4 0 9 1 0 1]$ (the list gene is mutated to 4).
Fig. 3 The evolution of fitness function (best value) for $\alpha = 0.0$ for the seventh solution from Tab. V

Fig. 4 The evolution of fitness function (best value) for $\alpha = 0.0$ for the tenth solution from Tab. V

Fig. 5 The evolution of fitness function (best value) for $\alpha = 0.3$ for the seventh solution from Tab. V
Fig. 6 The evolution of fitness function (best value) for $\alpha = 0.3$ for the tenth solution from Tab. V

Fig. 7 The evolution of fitness function (best value) for $\alpha = 0.7$ for the seventh solution from Tab. V

Fig. 8 The evolution of fitness function (best value) for $\alpha = 0.7$ for the tenth solution from Tab. V
For $\alpha = 0.3$ and $\alpha = 0.7$ we can see that for the two execution order solutions we obtained better fitness function values for the seventh solution (because the dependable requirements are previously satisfied). Only a few “changes” are allowed/possible for the tenth solution because the fourth requirement is satisfied at the “end” of the solution.

6.2.1 Discussion

The two approaches find different solutions with different final cost. Although the same solution could be found (for a proper instance of the given set of requirements, components and component costs and dependencies) the Greedy approach may not find the best solution.

As it can be deduced from the results presented above, the evolutionary approach is performing much better than Greedy algorithm in several aspects. Some of the advantages of using evolutionary algorithms are as follows: it obtain multiple solutions in a single run; it get better results in terms of both number of components used and total cost (while compared to Greedy Algorithm); it is fast; it can be scaled to any number of components and requirements.

7. Conclusion and future work

CBSE is the emerging discipline of the development of systems incorporating components. A challenge is how to assemble components effectively and efficiently.

Component Selection Problem has been investigated in this paper. We have proposed an evolutionary approach. In relation to existing approaches we have also considered the dependencies between the requirements that have to be satisfied by the final system.

References

Andreea Vescan: Optimal component selection using a multiobjective evolutionary algorithm


