Outi Räihä, Erkki Mäkinen and Timo Poranen

Simulated Annealing for Aiding Genetic Algorithm in Software Architecture Synthesis

DEPARTMENT OF COMPUTER SCIENCES
UNIVERSITY OF TAMPERE

D-2010-19

TAMPERE 2010

UNIVERSITY OF TAMPERE
DEPARTMENT OF COMPUTER SCIENCES
SERIES OF PUBLICATIONS D – NET PUBLICATIONS
D-2010-19, DECEMBER 2010

Outi Räihä, Erkki Mäkinen and Timo Poranen

Simulated Annealing for Aiding Genetic Algorithm in Software Architecture Synthesis

DEPARTMENT OF COMPUTER SCIENCES
FIN-33014 UNIVERSITY OF TAMPERE

ISBN 978-951-44-8410-0
ISSN 1795-4274

Simulated Annealing for Aiding Genetic Algorithm in Software Architecture Synthesis

Outi Räihä[1]
[1]Department of Software Systems
Tampere University of Technology
FI-33101 Tampere, Finland
Tel: +358-50-5342813
Fax: +358-3-31152913
outi.raiha@tut.fi

Erkki Mäkinen[2], Timo Poranen[2]
[2]School of Information Sciences,
Computer Science
University of Tampere
Finland
{erkki.makinen, timo.poranen}@uta.fi

Abstract

Automatic synthesis of software architecture has already been shown to be feasible with genetic algorithms. A natural problem is to augment – if not replace – genetic algorithms with some other search methods in the process of searching good architectures. The present paper studies the possibilities of using simulated annealing for synthesizing software architecture. We start from functional requirements which form a null architecture and consider three quality attributes, modifiability, efficiency and complexity. Synthesis is performed by adding design patterns and architecture styles; the end result being a software architecture which corresponds to the quality attributes. It is concluded that simulated annealing as such does not produce good architectures, but it is useful for speeding up the evolution process by quickly fine-tuning a base solution achieved with a genetic algorithm.

Keywords: search-based software engineering, simulated annealing, software design

1. Introduction

The ultimate goal of software engineering is to be able to automatically produce software systems based on their requirements. For the time being, we pass the synthesis of executable programs, and concentrate on the automated derivation of architectural designs of software systems. This is possible because architectural design largely means the application of known standard solutions in a combination that optimizes the quality properties (like modifiability and efficiency) of the software system. These standard solutions are well documented as architectural styles [33] and design patterns [10].

1: corresponding author

Genetic algorithms (GAs) [20] are shown to be a feasible method for producing software architectures from functional requirements [25, 26, 27, 30]. However, experiments with asexual reproduction [28] would suggest that the crossover operator which an essential part of GAs might not be critical for producing good architectures, which would support the idea of using a simpler search method. It is then natural to ask if other search methods are capable of producing equally good architectures alone or in co-operation with genetic algorithms. The purpose of the present paper is to study the possibilities of simulated annealing in the process of searching good architectures when functional requirements are given.

Contrary to GAs, simulated annealing (SA) is a local search method which intensively uses the concept of neighborhood, i.e., the set of possible solutions that are near to the current solution. The neighborhood is defined via transformations that change an element of the search space (here, software architecture) to another. In our application the transformations mean implementing a design pattern or an architectural style.

While GA is already shown to produce reasonable software architectures, it is of great interest to study whether SA is capable to do the same, as it explores the search space in a completely different way than GA. An affirmative answer would, of course, give us a new competitive practical method for producing software architectures. But at the same time, it would be a further confirmation for the thesis, that software engineering in general – and especially software architecture design – is fundamentally a combinatorial search problem.

As with our GA approach, we begin with the functional requirements of a given system. These contain the operations defining the functional entities in the system, the relationships between them, and a null architecture, giving a rough structure for the system. The actual architecture is achieved by the SA algorithm, which gradually transforms the system by adding (and removing) design patterns and applying architecture styles. The resulting architecture is evaluated from three (contradicting) viewpoints: modifiability, efficiency and complexity. The result is given as a UML class diagram, which depicts the produced architecture for the given system, and contains all the patterns and other design choices made by the algorithm.

This paper proceeds as follows. In Section 2 we sketch current research in the field of search algorithms in software design that is relevant for the present paper. In Section 3 we cover the basics of implementing a SA algorithm. In Section 4 we introduce our method by defining the input for the SA algorithm, the transformations and the evaluation function. In Section 5 we present the results from our experiments, as we examine different parameters for the SA and combining SA with our GA implementation. In Section 6 we discuss the findings and in Section 7 we give a conclusion of our results.

2. Related Work

Search-based software engineering (SBSE) considers software related topics as combinatorial search problems. Traditionally, testing has been the clearly most studied area inside SBSE [12]. Other well studied areas include software clustering and refactoring [9, 12, 24]. Using metaheuristic algorithms in the area of software design, and in particular at software architecture design, is quite a novel idea. Only a few studies have been published where the algorithm actually attempts to design something new, rather than re-designing an existing software system. However, approaches dealing with higher level structural units, such as patterns, have recently gained more interest. We will briefly discuss the studies with closest relation to our approach.

Amoui et al. [2] use the GA approach to improve the reusability of software by applying architecture design patterns to a UML model. The authors' goal is to find the best sequence of transformations, i.e., pattern implementations. Used patterns come from the collection presented by Gamma et al. [10]. From the software design perspective, the transformed designs of the best chromosomes are evolved so that abstract packages become more abstract and concrete packages in turn become more concrete. This approach only uses one quality factor (reusability), and also a more refined starting point than what is used in our approach.

Bowman et al. [7] study the use of a multi-objective genetic algorithm (MOGA) in solving the class responsibility assignment problem. The objective is to optimize the class structure of a system through the placement of methods and attributes within given constraints. So far they do not demonstrate assigning methods and attributes "from scratch" (based on, e.g., use cases), but try to find out whether the presented MOGA can fix the structure if it has been modified.

Simons and Parmee [34, 35] take use cases as the starting point for system specification. Data is assigned to attributes and actions to methods, and a set of uses is defined between the two sets. The notion of a class is used to group methods and attributes. Each class must contain at least one attribute and at least one method. Design solutions are encoded directly into an object-oriented programming language.

Räihä et al. [25] have taken the design of software architecture a step further than Simons and Parmee [34, 35] by starting the design from a responsibility dependency graph. The dependency graph can also be achieved from use cases, but the architecture is developed further than the class distribution of actions and data. A GA is used for the automation of design. In this solution, each responsibility is represented by a supergene and a chromosome is a collection of supergenes. The supergene contains information regarding the responsibility, such as dependencies of other responsibilities, and evaluated parameters such as execution time and variability. Mutations are implemented as adding or removing an architectural design pattern [10] or an interface or splitting or joining class(es). Implemented design patterns are Façade and Strategy, as well as the message dispatcher architecture style [33].

Räihä et al. [30] have also applied GAs in model transformations that can be understood as pattern-based refinements. In MDA (Model Driven Architecture), such transformations can be exploited for deriving a Platform Independent Model from a Computationally Independent Model. The approach uses design patterns as the basis of mutations and exploits various quality metrics for deriving a fitness function. They give a genetic representation of models and propose transformations for them. The results suggest that GAs provide a feasible vehicle for model transformations, leading to convergent and reasonably fast transformation process. In their recent studies, Räihä et al. [26] have added scenarios, which are common in real world architecture evaluations, to evaluate the fitness of their synthesized architectures.

Jensen and Cheng [14] present an approach based on genetic programming for generating refactoring strategies that introduce design patterns. The authors have implemented a tool, REMODEL, which takes as input a UML class diagram representing the system under design. The system is refactored by applying "mini-transformations". The encoding is made in tree form (suitable for GP), where each node is a transformation. A sequence of mini-transformations can produce a design pattern; a subset of the patterns specified by Gamma et al. [10] is used to identify desirable mini-transformation sequences. Mutations are applied by simply changing one node (transformation), and crossover is applied as exchanging sub-trees. The QMOOD [4] metrics suite is used for fitness calculations. In addition to the QMOOD metrics, the authors also give a penalty based on the number of used mini-transformations and reward the existence of (any) design patterns. The output consists of a refactored software design as well as the set of steps to transform the original design into the refactored design. This way the refactoring can be done either automatically or manually; this decision is left for the software engineer. This approach is very close to those of Räihä et al. and the approach used here, the difference being that Jensen and Cheng have clearly a refactoring point of view, while we attempt upstream synthesis.

A higher level approach is studied by Aleti et al. [1], who use AADL models as a basis, and attempt to optimize the architecture with respect to Data Transfer Reliability and Communication Overhead. They use a GA and a Pareto optimal fitness function in their ArcheOptrix tool, but they concentrate on the optimal deployment of software components to a given hardware platform rather than how the components are actually constructed and how they communicate with one another.

Research has also been made on identifying concept boundaries and thus automating software comprehension [11] and re-packaging software [5], which can be seen as finding working subsets of an existing architecture. These approaches are, however, already pushing the boundaries of the concept "software architecture design". As for different aspects on GAs, the role of crossover operations in genetic synthesis of software architectures is studied by Räihä et al. [27, 28].

SA has been used in the field of search-based software engineering for software refactoring [21, 22, 23] and quality prediction [6]. O'Keeffe and Ó Cinnéide [21, 22, 23] work on the class level and use SA to refactor the class hierarchy and move methods in order to increase the quality of software. Their goal is to minimize unused, duplicated and rejected methods and unused classes, and to maximize abstract classes. The algorithm operates with pure source code, and the outcome is given as refactored code as well as a design improvement report. This approach is the closest to the one presented here, but it operates on a lower level and backwards (re-engineering), while our approach operates on a higher level architecture and goes forwards in the design process. Similar studies (class level refactoring) have also been made by Seng et al. [31, 32], who use GA as their search algorithm and Harman and Tratt [13], who use hill climbing.

In the area of quality prediction, Bouktif et al. [6] attempt to reuse and adapt quality predictive models, each of which is viewed as a set of expertise parts. The search then aims to find the best subset of expertise parts, which forms a model with an optimal predictive accuracy. The studies using SA are few, and none use this algorithm for such a high-level design problem as designing software architecture from requirements.

3. Simulated Annealing

Simulated annealing is a widely used optimization method for hard combinatorial problems. Principles behind the method were originally proposed by Metropolis et al. [18] and later Kirkpatrick et al. [16] generalized the idea for combinatorial optimization.

The SA algorithm starts from an initial solution which is enhanced during the annealing process by searching and selecting other solutions from the neighborhood of the current solution. There are several parameters that guide the annealing. The search begins with initial temperature t_0 and ends when the temperature t is decreased to the frozen temperature t_f, where $0 \leq t_f \leq t_0$. The temperature gives the probability of choosing solutions that are worse than the current solution. The result of a transformation that worsens the current solution by δ, is accepted to be the new current solution if a randomly generated real i is less than or equal to a limit which depends on the current temperature *t*. If a transformation improves the current solution, it is accepted directly without a test.

Algorithm 1 simulatedAnnealing

```
Input:
Responsibility dependency graph G, null architecture M, initial temperature $t_0$,
frozen temperature $t_f$, cooling ratio $\alpha$ , and temperature constant $r$
Output: UML class diagram D;
 $initialSolution \leftarrow$ encode(G ,M)
$initialQuality \leftarrow$ evaluate($initialSolution$)
    $S_1 \leftarrow initialSolution$
    $Q_1 \leftarrow initialQuality$
    $t \leftarrow t_0$
while $t > t_f$ do
    $r_i \leftarrow 0$
    while $r_i < r$ do
        $S_i \leftarrow$ transform($S_1$)
        $Q_i \leftarrow$ evaluate($S_i$)
        if $Q_i > Q_1$
            $S_1 \leftarrow S_i$
            $Q_1 \leftarrow Q_i$
        else
            $\delta \leftarrow Q_1 - Q_i$
            $p \leftarrow$ UnifomProbability
            if $p < e^{-\delta /t}$
                $S_1 \leftarrow S_i$
                $Q_1 \leftarrow Q_i$
            end if
        end if
        $r_i \leftarrow r_i + 1$
    end while
    $t \leftarrow \alpha * t$
end while
$D \leftarrow$ generateUML($S_1$)
return $D$
```

An important parameter of SA is the cooling schedule, i.e., how the temperature is decreased. We use the geometric cooling schedule, in which a constant *r* is used to determine when the temperature is decreased, and the next temperature is obtained simply by multiplying the current temperature by cooling ratio α ($0 < \alpha < 1$). This is the most frequently used schedule [36]. It was chosen because of its simplicity, and because of the fact that all the classical cooling schedules can be tuned so that they give the same practical temperatures [36].

The SA has been successfully applied for numerous combinatorial optimization problems, for an instructive introduction to the use of SA as a tool for experimental algorithmics, see [3, 15]. In order to determine good parameters for a problem, experimental analysis is often needed. There are also adaptive techniques for detecting the parameters [17].

The SA implementation used in our tests is shown in Algorithm 1. In Section 5 we compare the present SA and our previous GA implementation [25]. In order to be able to fairly compare the implementations, the solutions produced by the two methods should be evaluated by the same quality functions and the initial solutions should be of the same quality. Hence, we use the same method for producing the initial solutions for SA as we have done with GA in [25, 26, 30]. The initial solution is achieved by encoding functional requirements and thus building a base architecture. The base class structure is derived from the null architecture, and the base architecture is achieved by randomly applying a transformation. The same approach for creating several solutions for an initial population is used in our GA implementation [25, 26, 30], and thus the initial quality is the same for both SA and GA, as they both use the same evaluation function.

4. Method

We begin by creating use cases to define the basic functional requirements. The use cases can be refined into sequence diagrams, where in turn operations and classes can be elicited. This results in a null architecture, giving a structural view of the functional requirements of the system at hand but not dealing with the quality requirements. The null architecture is encoded to a form that can be processed by the search algorithm in question. The algorithm produces software architecture for the given quality requirements by implementing selected architecture styles and design patterns, and produces a UML class diagram as the result.

4.1. Requirements

We will use two example systems: the control system for a computerized home, called hereafter ehome, and a robot war game simulator, called robo. We will demonstrate building input for the search algorithm in the case of ehome; building input is similar in the case of robo.

Use cases for the ehome system are assumed to consist of, e.g., logging in, changing the room temperature, changing the unit of temperature, making coffee, moving drapes, and playing music. In Figure 1, the coffee making use case has been refined into a sequence diagram.

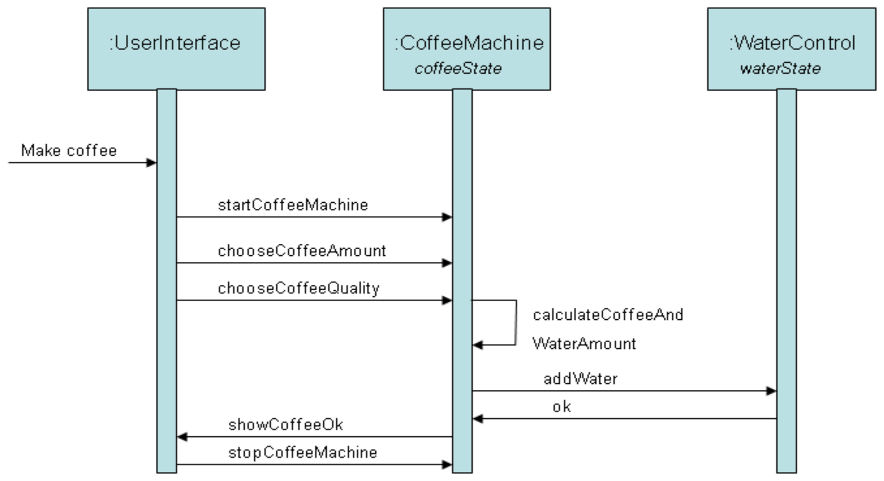

Fig. 1 Make coffee use case refined

The use case begins with a message from the user to begin the process of making coffee. We will not consider the technical details of the user interface component here, but simply assume that it receives a command which may be composed of a series of subcommands (such as here, making coffee requires the user to define a series of properties regarding the amount and quality of coffee). As all functionalities related to the process of making coffee cannot be included in the user interface component, we realize that a CoffeeMachine object (component) is needed. The UI thus relays a message to the CoffeeMachine that the coffee machine should be turned on, and then goes on to giving other relevant information. After all data is gathered, we notice that the coffee machine is not capable of producing coffee on its own: it needs to be connected to a water source. Thus, the WaterManager component is created, and the CoffeeMachine sends it a message to add water. Other use cases can be refined in a similar fashion.

The null architecture in Figure 2 for the ehome system can be mechanically derived from the sequence diagrams. The messages in the sequence diagram now become the operations and the objects/components become the classes. Also, if the need for a data source is detected, such as in this case the WaterState and CoffeeState, they will become attributes in the classes. The null architecture only contains use relationships, as no more detail is given for the algorithm at this point. The null architecture represents the basic functional decomposition of the system. A null architecture for robo (which can be achieved by performing the same steps as did with ehome) is given in Figure 3.

After the operations are derived from the use cases, some properties of the operations can be estimated to support the genetic synthesis, regarding the amount of data an operation needs, frequency of calls, and sensitiveness for variation. For example, it is likely that the coffee machine status can be shown in several different ways, and thus it is more sensitive to variation than ringing the buzzer when the coffee is done. Measuring the position of drapes requires more information than running the drape motor, and playing music quite likely has a higher frequency than changing the password for the system. Relative values for the chosen properties can similarly be estimated for all operations. This optional information, together with operation call dependencies, is included in the information subjected to encoding.

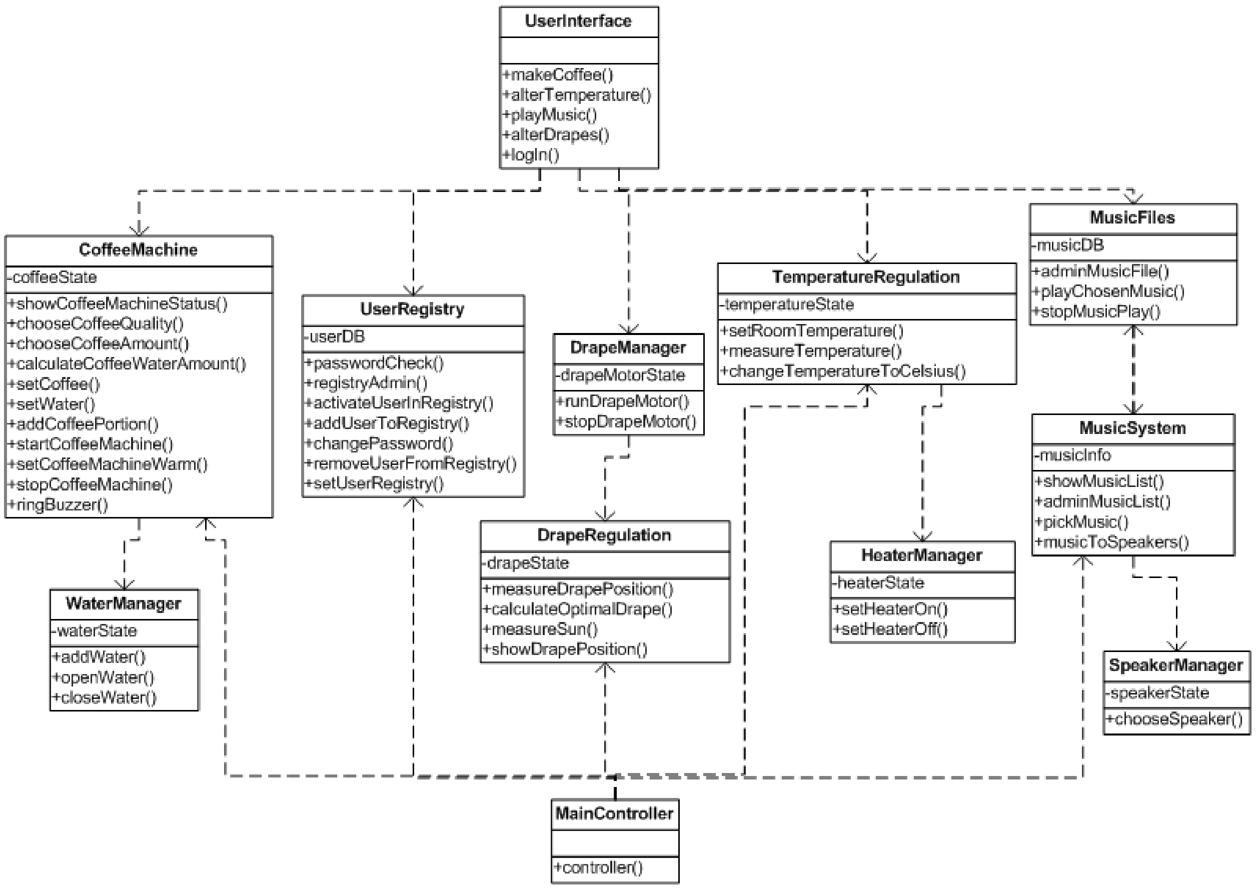

Fig. 2 Null architecture for ehome

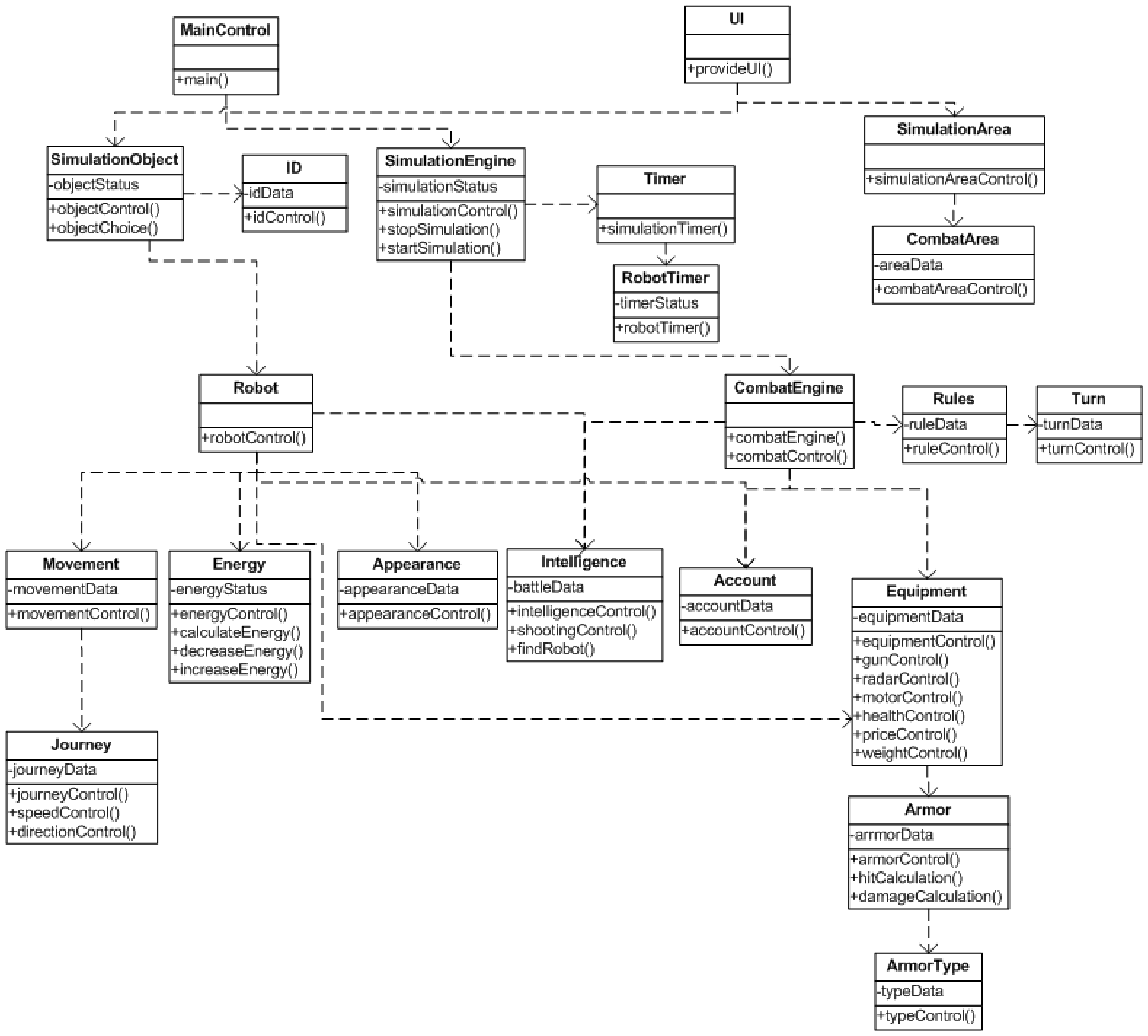

Fig. 3 Null architecture for robo

4.2. Encoding

Ultimately, there are two kinds of data regarding each operation o_i. Firstly, there is the basic information given as input. This contains the operations $O_i = \{o_{i1}, o_{i2}, \ldots, o_{ik}\}$ depending on o_i, its name n_i, type d_i, frequency f_i, parameter size p_i and variability v_i. Secondly, there is the information regarding the operation o_i's place in the architecture: the class(es) $C_i = \{C_{i1}, C_{i2}, \ldots, C_{iv}\}$ it belongs to, the interface I_i it implements, the dispatcher D_i it uses, the operations $OD_i \subset O_i$ that call it through the dispatcher, the design patterns $P_i = \{P_{i1}, P_{i2}, \ldots, P_{im}\}$ it is a part of, and the predetermined null architecture class MC_i. The dispatcher is given a separate field as opposed to other patterns for efficiency reasons.

The null architecture is encoded for the algorithm as a vector $V <ov_1, ov_2, \ldots, ov_n>$ of vectors $ov_1, ov_2, \ldots, ov_n$. Each vector ov_k, in turn, contains all data for a single operation. Thus, n is the number of operations of a system, and the collection of these operation defining vectors depicts the entire system when collected into one vector *V*. Figure 4 depicts an operation vector ov_k.

O_i	n_i	d_i	f_i	p_i	v_i	C_i	I_i	D_i	OD_i	MC_i	P_i

Fig. 4. Operation vector ov_k

4.3. Transformations

An architecture is transformed (i.e., one of its neighbors is found) by implementing architecture styles and design patterns to a given solution. The patterns we have chosen include very high-level architectural styles [33] (message dispatcher and client-server), medium-level design patterns [10] (Façade and Mediator), and low-level design patterns [10] (Strategy, Adapter and Template Method). The transformations are implemented in pairs of introducing a pattern or removing a pattern. This ensures a wider traverse through the search space, as while implementing a pattern might improve the quality of architecture at one point, it might become redundant over the course of development. The dispatcher architecture style makes a small exception to this rule: the actual dispatcher must first be introduced to the system, after which the responsibilities can communicate through it. The transformations are the following, and each of them has a certain probability with which it is selected:

- introduce/remove message dispatcher
- communicate/remove communication through dispatcher
- introduce/remove server
- introduce/remove Façade
- introduce/remove Mediator
- introduce/remove Strategy
- introduce/remove Adapter
- introduce/remove Template Method.

The legality of applying a pattern is always checked before transformations by giving preconditions. For example, the structure of the Template Method demands that depending operations are in the same class. In addition, a corrective function is added to check that the solution conforms to certain architectural laws, and that no anomalies are brought to the architecture. These laws demand uniform calls between two classes (e.g., through an interface or a dispatcher but not both), and state some basic rules regarding architectures (e.g., no operation can implement more than one interface). The corrective function, for example, discards interfaces that are not used by any class, and adds dispatcher connections between operations in two classes, if such a connection already exists between some operations in those classes.

For example, if the “add Strategy” transformation is chosen, it is checked that the operation o_i is called by some other operation in the same class *c* and that it is not a part of another pattern already (pattern field is empty). Then, a Strategy pattern instance sp_i is created. It contains information of the new class(es) sc_i where the different versions of the operation are placed, and the common interface si_i they implement. It also contains information of all the classes and operations that are dependent on o_i, and thus use the Strategy interface. Then, the value in the class field in the vector ov_i (representing o_i) would be changed from *c* to sc_i, the interface field would be given value si_i and the pattern field the value sp_i. Adding other patterns is done similarly. Removing a pattern is done in reverse: the operation placed in a "pattern class" would be returned to its original null architecture class, and the pattern found in the supergene's pattern field would be deleted, as well as any classes and interfaces related to it.

4.4. Quality Function

In the case of software architecture design, selecting an appropriate evaluation function is particularly difficult, as there is no clear value to measure in the solutions. In real world, evaluation of software architecture is almost always done manually by human designers, and metric calculations are only used as guidelines. Also, two architects rarely agree on a unique quality for certain architecture, as evaluation is bound to be subjective, and different values and backgrounds will influence the outcome of any evaluation process. However, for a search algorithm to be able to evaluate the architecture, a purely numerical quality value must be calculated.

In a fully automated approach, no human interception is allowed, and thus the evaluation function needs to be based on metrics. The selection of metrics may be as arguable as the evaluations of two architects on a single software architecture. The rationale behind the selected metrics in this approach is that they have been widely used and recognized to accurately measure some quality aspects of software architecture. Hence, the metrics are chosen so that they measure quality aspects that can be seen as "most agreed upon" in the real world, and singular values can be seen as accurate as possible. However, the combination of metrics and multiple optimization is another problem entirely. For many metrics, it may be arguable what quality attribute they measure, and may be seen as measurements for several different quality attributes. Many of these quality attributes, however, are controversial. A perfect example is the selected quality attribute pair: modifiability and efficiency. The problem of multiple optimization is a direct result of the contradictive aims of the two quality attributes: when attempting to optimize one, the quality will decrease in view of the other. In our GA approach we have implemented Pareto optimality [29] to conquer this problem. However, when evaluating the applicability of simulated annealing, we found it more practical to use a single weighted fitness.

The chosen quality function is based on well-known software metrics [8]. These metrics, especially coupling and cohesion, have been used as a starting point for the quality function, and have been further developed and grouped to achieve clear "sub-functions" for modifiability and efficiency, both of which are measured with a positive and negative metric. The biggest modifications to the basic metrics include taking into account the positive effect of interfaces and the dispatcher and client-server architecture styles in terms of modifiability, as well as the negative effect of the dispatcher and server in terms of efficiency. Choosing and grouping the metrics this way makes sure that all architectural decisions are always considered from all viewpoints. Adding a pattern always adds a classes or an interface (or both), and is thus considered by complexity. As the calls to an operation are also affected, the change is always also considered positive or negative by both modifiability and efficiency.

Dividing the evaluation function into sub-functions also answers the demands of the real world. Hardly any architecture can be optimized from all quality viewpoints, but some viewpoints are ranked higher than others, depending on the demands regarding the architecture. By separating efficiency and modifiability, which are especially difficult to optimize simultaneously, we can assign a bigger weight to the more desired quality aspect. When w_i is the weight for the respective sub-function sf_i, the evaluation function $f_c(x)$ for solution x can be expressed as

$f_c(x) = w_1*sf_1 - w_2*sf_2 + w_3* sf_3 - w_4* sf_4 - w_5* sf_5$.

Here, sf_1 measures positive modifiability, sf_2 negative modifiability, sf_3 positive efficiency, sf_4 negative efficiency and finally sf_5 measures complexity. All the sub-functions are normalized so that they have the same range. The sub-functions are defined as follows (|X| denotes the cardinality of X):

sf_1 = (|calls to interfaces| * ∑ (variabilities of operations called through interface)) + (|calls through dispatcher|) ∗ ∑ (variabilities of operations called through dispatcher)) – |unused operations in interfaces| ∗ β ,

sf_2 = |calls between operations in different classes, that do not happen through a pattern|* ∑ (variabilities of called operations) + |calls between operations same class|* ∑ (variabilities of called operations) *2,

sf_3 = ∑ (|operations dependent of each other within same class| ∗ parameterSize) + ∑ (|usedOperations in same class| ∗ parameterSize + |dependingOperations in same class| ∗ parameterSize),

sf_4 = ∑ ClassInstabilities + (2*|dispatcherCalls| + |serverCalls|)∗ ∑ frequencies + |calls between operations in different classes|,

sf_5 = |classes| + |interfaces|.

The multiplier β (β > 1) in sf_1 means that having unused operations in an interface is almost like breaking an architecture law, and should be more heavily penalized. It should also be noted, that in sf_1, most patterns also contain an interface. In sf_3, “usedOperations in same class” means a set of operations in class C, which are all used by the same operation from class D. Similarly, “dependingOperations in same class” mean a set of operations in class K, which all use the same operations in class L.

5. Experiments

In this section we present the results from the preliminary experiments done with our approach. Tests were made using the ehome and robo example systems (introduced before). Most of the parameters used in our tests originate from the previous tests reported in [25, 26, 30], and give promising results with the GA approach. The implementation was made with Java 1.5. The tests were run on a DELL laptop with 2,95 GB of RAM and 2,26 GHz processor, running with Windows XP.

All tests were made with the constant r set to 20, and frozen (final) temperature t_f set to 1. The weights for all sub-functions of the quality evaluation function were set to the same, i.e., all weights w_i were set to 1. The calculated quality value for each curve is the average value from 20 test runs.

The GA used in the combination experiments is based on our previous implementations [25, 26, 30]. The GA uses the same encoding, transformations (mutations) and quality function as defined here for the SA. The crossover operator is a single-point random crossover. Selection is made with a rank-based roulette wheel method. As this paper concentrates on simulated annealing, the particularities of the GA implementation are not discussed further here; details can be found in [25, 26, 30] and a general introduction to GAs is given by, e.g., Michalewicz [19].

5.1 Using SA First

The “standard” tests were made with 7500 as starting temperature and 0.05 as cooling ratio. A longer annealing was also experimented with by setting the starting temperature to 10 000 (cooling ratio 0.05), and a faster annealing was tested by setting the cooling ratio to 0.15 (starting temperature 7500). However, the results were unsatisfactory for both systems, and there were no significant differences between the results achieved with different SA parameters. The trend of the quality curve for the SA was descending, and the end quality value was worse than the initial value. The standard and high temperature tests for both systems took approximately 10 seconds/ run and the fast annealing tests less than 5 seconds/ run. We then tried to build a base solution with a short and fast annealing (starting temperature 2500 and cooling ratio 0.15), and then continue the search with a genetic algorithm, which ran for 250 generations and had a population of 100 (combination SAGA). This approach did not produce much better results: the SA curves were quite similar than with longer and slower runs, and while the quality curve for the GA portion did increase for a short while, it began to quickly descend drastically. Again, the end quality value was worse than the value for the initial solution. The SAGA test runs took a little less than one minute per run for both systems.

5.2 Using a Combination of GA and SA

As using the SA first did not produce good results, we finally tried using GA for creating a good base solution (again, with 250 generations and a population of 100), and then applying SA (starting temperature 2500, cooling ratio 0.15) for further tuning the solution (combination GASA). In this case, the results were much better. The GA does a good basic work, and the SA is able to further improve the solution very quickly. The runtime for these tests were approximately a bit more than one minute per run for ehome, while the runs for robo were slightly faster.

Figure 5 presents the GA portion and Figure 6 the SA portion of the GASA quality curve for ehome. Figures 7 and 8 present the respective curves for robo. Note, that the SA algorithm starts where the GA ends: the difference in the GA end value and SA start value is due to the fact that quality values are not recorded until one round of transformations has already been completed. As can be seen in Figure 5, the GA begins with a short plummet, after which the quality (fitness) begins to develop steadily. After about 100 generations the fitness appears to stabilize, i.e., the curve is not increasing, and it does not seem likely to further develop. In Figure 6, the SA begins to develop the solution from where the GA left off, and the curve develops rapidly until quite near the end of the SA process.

In Figure 7, depicting the GA portion for the robo system, the GA first plummets similarly as in the curve for ehome, but after it starts ascending, the development seems more rapid and steady than for the ehome, and it appears as if the quality could still increase after the GA finishes. The SA portion of the GASA curve for robo, in Figure 8, seems quite similar to the GA curve at first, but looking at the actual quality values reveals that the SA develops much more quickly than the GA. In the end it seems that the SA has found some optimum, as the curve has reached a plateau.

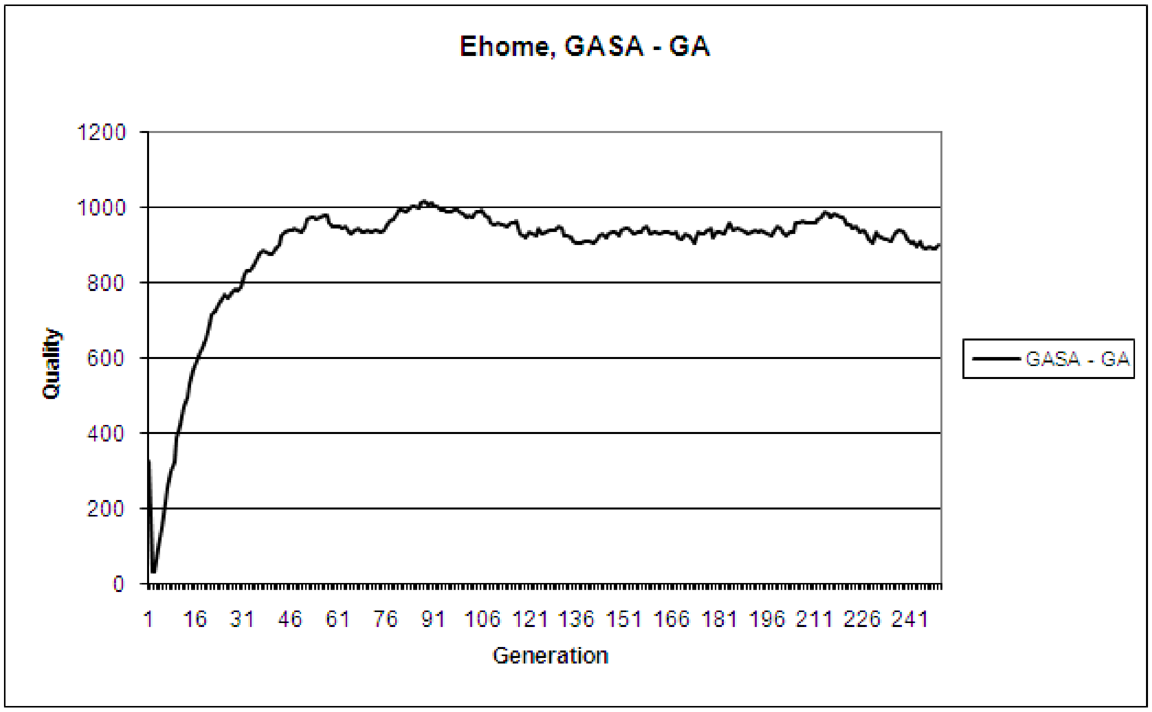

Fig. 5. GA portion of GASA quality curve for ehome

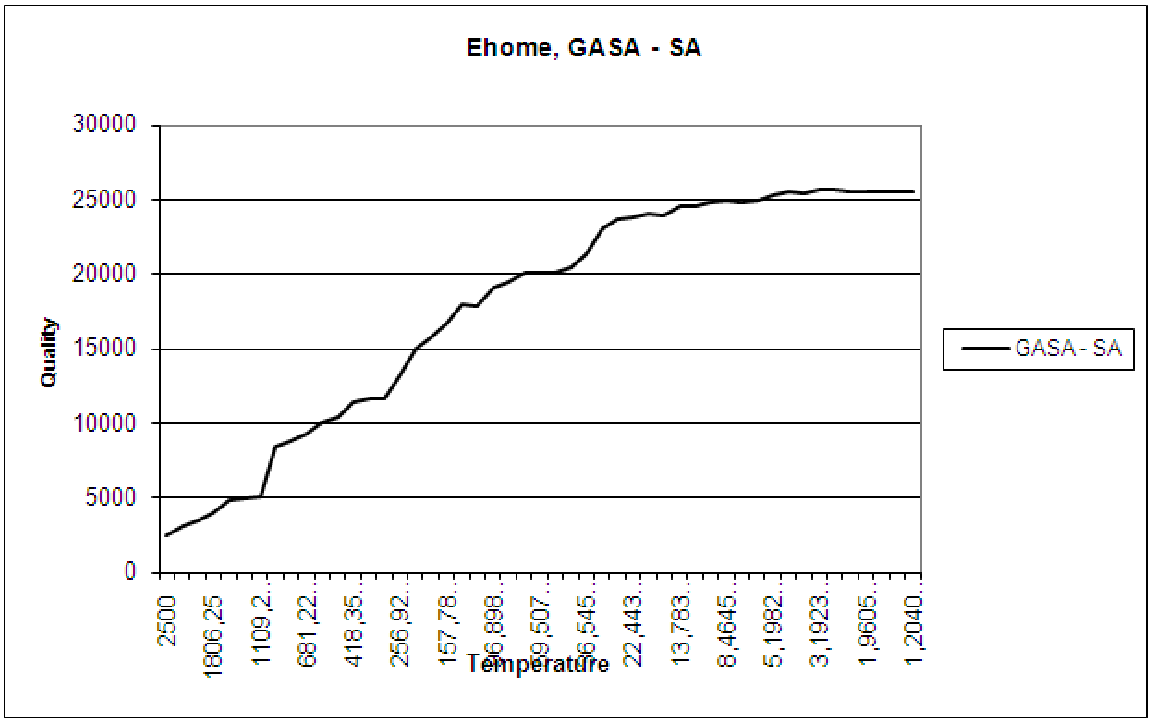

Fig. 6. SA portion of GASA quality curve for ehome

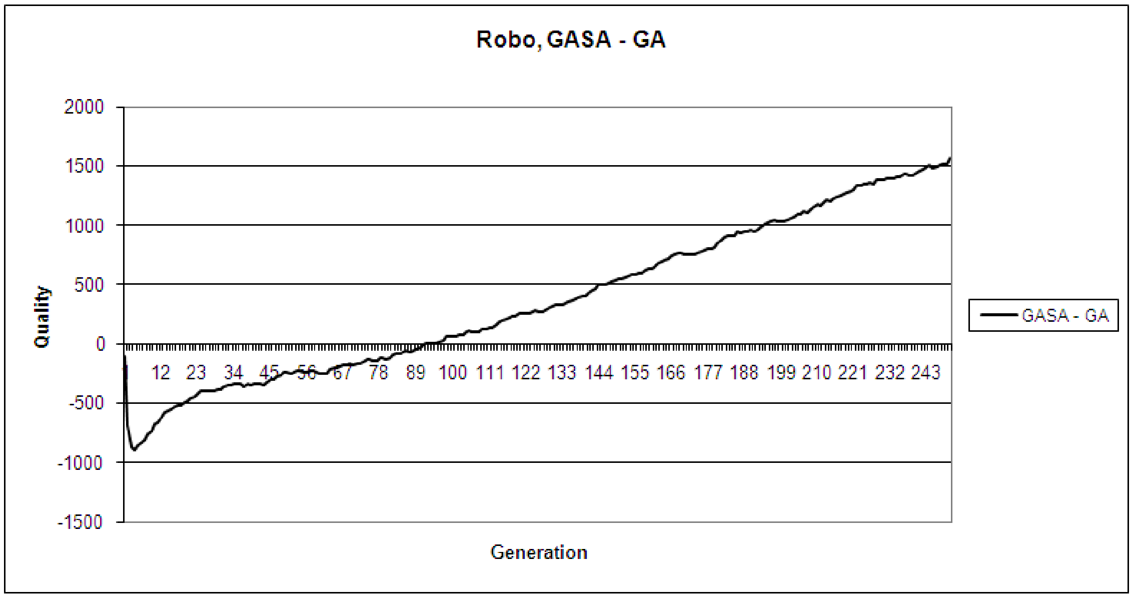

Fig. 7. GA portion of GASA quality curve for robo

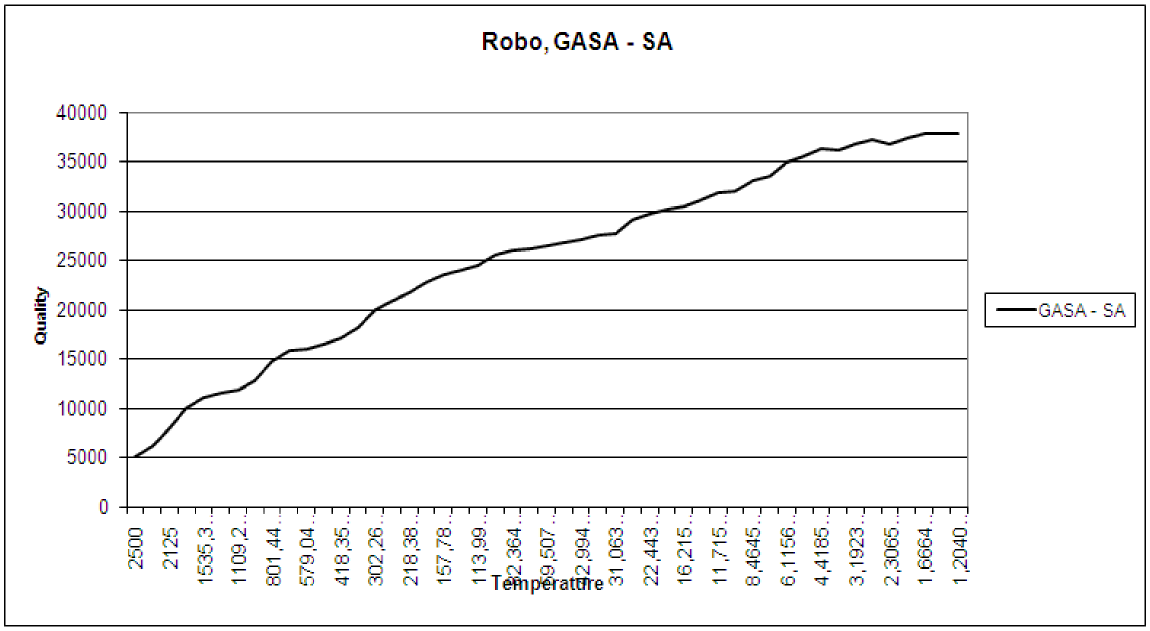

Fig. 8. SA portion of GASA quality curve for robo

6 Discussion

In Section 5 we discussed the quality curves of the experiments made with the SA algorithm. Naturally, the actual UML graphs given as output should also be examined to get a wholesome idea of whether the results with extreme quality values are actually good. In addition to discussing the class diagrams related to the test graphs presented in Section 5 (the GASA tests), we will also discuss the UML graphs achieved when SA was used primarily. The example solutions are given in a simplified format where the design solutions are emphasized, rather than giving the actual class diagrams given by the algorithm, as they would be too space-consuming and difficult to interpret.

6.1 Proposed Architectures with GASA

Using the GASA approach produced very similar solutions for both ehome and robo systems. The solutions were built around the message dispatcher, as nearly all communication between classes (in different null architecture classes) was handled through the message dispatcher. The dispatcher makes the system highly modifiable, as classes do not need to know any details of other classes; they merely send and receive messages through the dispatcher. The architecture is also easy to understand quickly, as the message dispatcher creates a logical center for the system and separates different model classes. However, the message dispatcher creates huge loss in efficiency, as the increased message traffic greatly affects the performance of a system. Thus, it should be used as the primary method for communication or not be used at all, as in the case where it is only partially used the cost in efficiency is bigger than the gain in modifiability.

In addition to the message dispatcher, all solutions achieved with the GASA approach had several instances of the Adapter pattern. The Adapter pattern is easy to apply, as it has very loose preconditions, but it is more costly in terms of efficiency than other patterns. There were usually also several instances of the Template Method pattern, which, in turn, is very low cost in terms of both efficiency (it does not increase the number of calls) and complexity (only one class, no interface). In some cases, however, the algorithm had preferred the Strategy pattern, and there would be many instances of Strategy, while only a few Template Method instances.

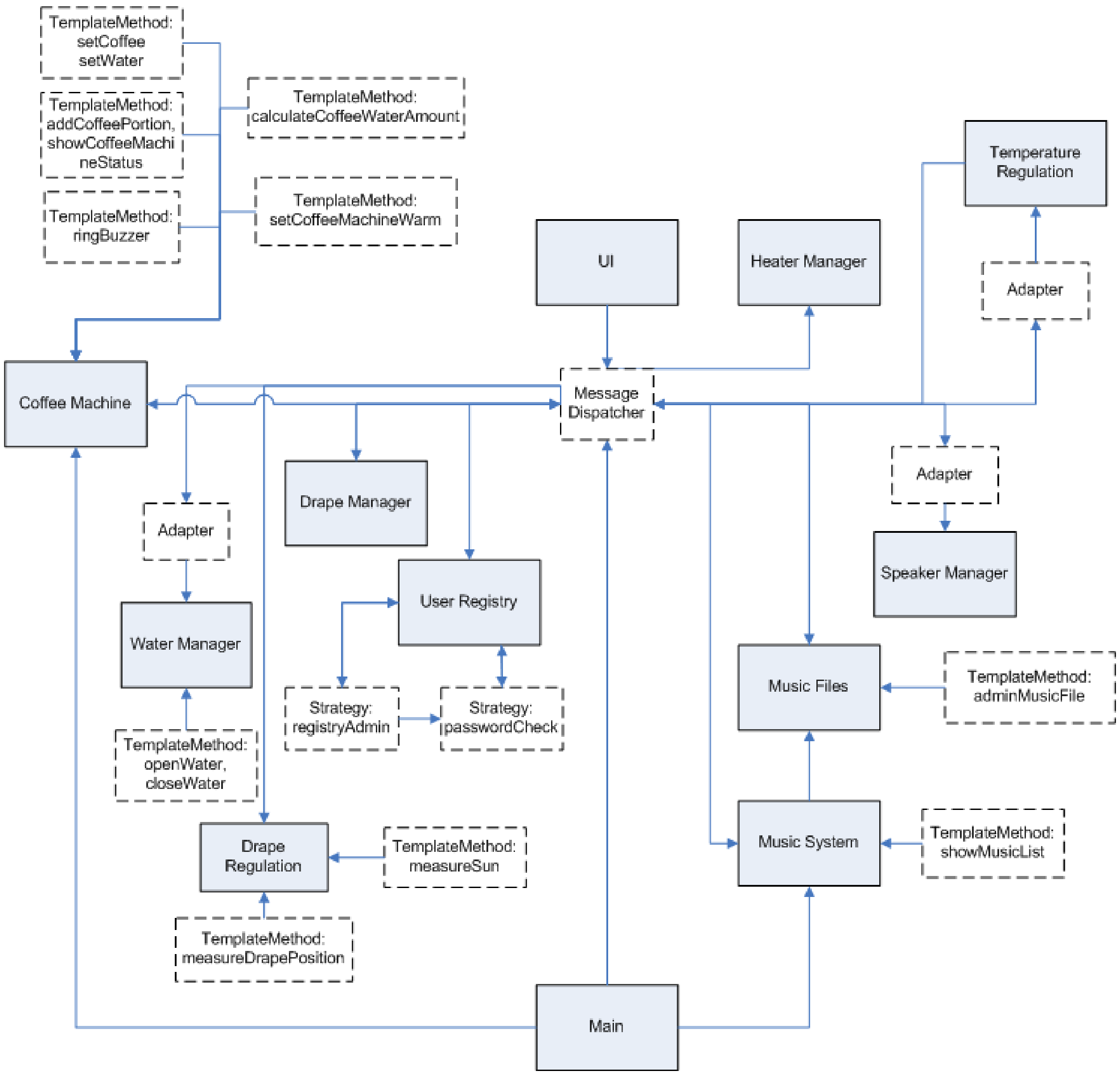
TemplateMethod: setCoffee setWater
TemplateMethod: addCoffeePortion, showCoffeeMachineStatus
TemplateMethod: ringBuzzer
TemplateMethod: calculateCoffeeWaterAmount
TemplateMethod: setCoffeeMachineWarm
Temperature Regulation
UI
Heater Manager
Adapter
Coffee Machine
Message Dispatcher
Drape Manager
Adapter
Adapter
Speaker Manager
Water Manager
User Registry
Music Files
TemplateMethod: adminMusicFile
Strategy: registryAdmin
Strategy: passwordCheck
TemplateMethod: openWater, closeWater
Drape Regulation
TemplateMethod: measureSun
Music System
TemplateMethod: showMusicList
TemplateMethod: measureDrapePosition
Main

Fig.9. Example architecture for ehome, with GASA algorithm combination

An example solution for ehome achieved with GASA is presented in Figure 9. As can be seen, nearly all connections are handled via the message dispatcher, as only calls from the Main component to Music System and Coffee Machine, and from Music System to Music Files are handled directly between the components. The example also shows that the Template Method is used very much to create low-level modifiability. The ehome is particularly suitable for a message dispatcher architecture style, and achieving a high level of message-based communication between components is desirable.

A similar example solution for robo (also achieved with GASA) is presented in Figure 10. As can be seen, the message dispatcher is used here even more intensely than in the case of ehome, as only connections between CombatEngine and Rules and some connections involving the SimulationObject are not using the message dispatcher, even though the amount of components is larger than in the case of ehome. In this example there are also several Adapter, TemplateMethod and Strategy patterns, and the usage of these different patterns is more balanced than in the case of ehome, where the Template Method was the dominating pattern. However, while using the message dispatcher in these proportions is desirable if it is chosen as the primary architecture style, if we consider the type of system the robo is (a framework), in real life a message dispatcher would probably not be the best option. All the components are actually tightly linked, and the design should concentrate more on extendibility and the actual functionality of the system. Also, as robo is a gaming application, using the message dispatcher in this extent would probably lead to significant disadvantage in terms of efficiency, which is particularly undesirable when the system needs to respond quickly

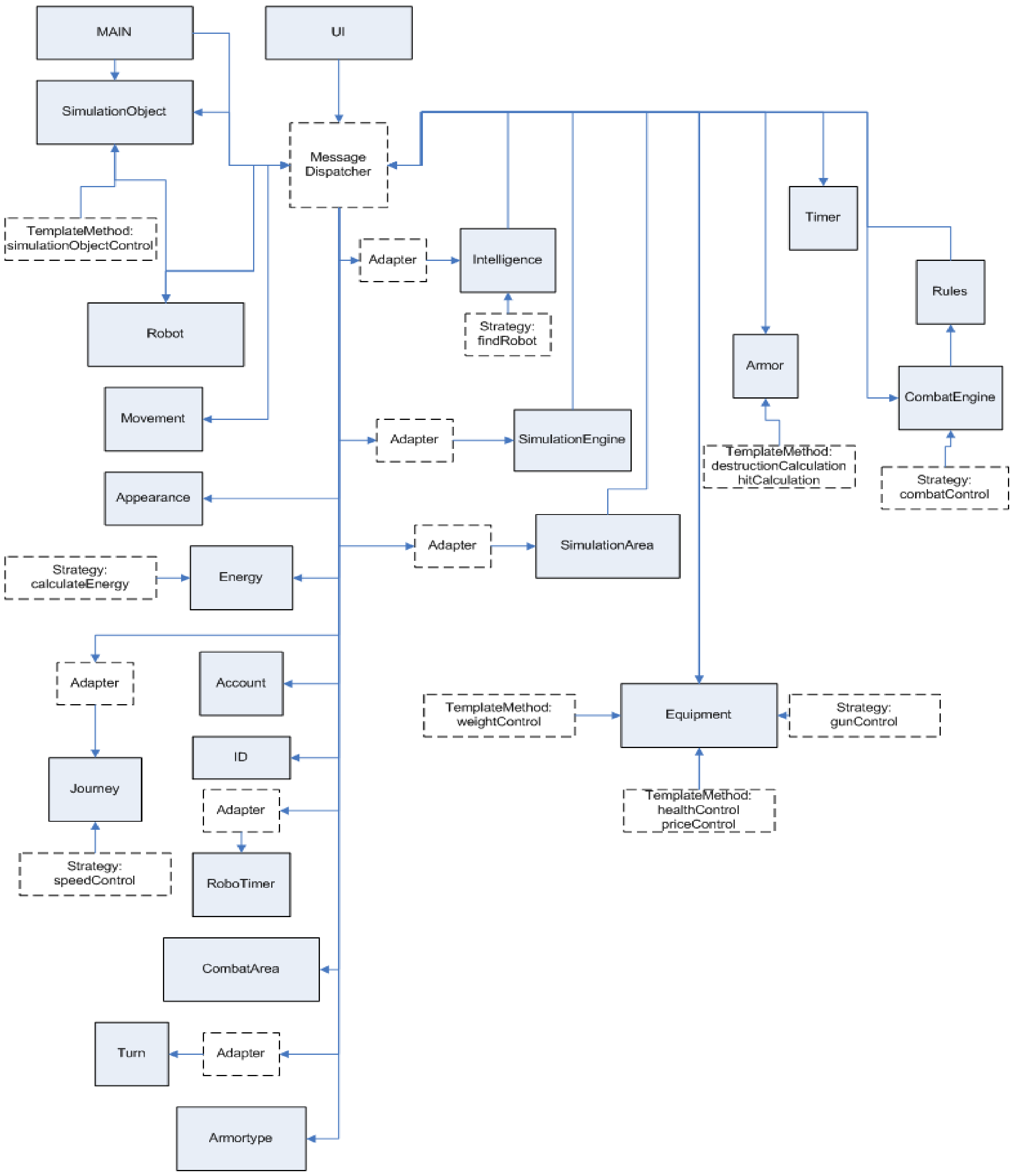

Fig.10. Example architecture for robo, with GASA algorithm combination

To summarize, using the message dispatcher gives a clear focal point in the solutions, and the full potential of the message dispatcher is used. It should also be pointed out that solutions achieved after only running the GA (i.e., the seeds for the SA) often had the message dispatcher, but its usage was mostly quite minimal, as only a couple of components were communicating via the dispatcher. Thus, the SA algorithm has a significant influence in achieving a much better level of usage in the final solution. In addition, low-level design patterns are used to further fine-tune the solution at class-level.

6.2 Proposed Architectures Based on SA

As mentioned, we also performed tests with only SA and by combining SA to GA by using the SA produced solution as a seed for the GA. The produced solutions were very similar for all cases of the SA (high temperature, standard, and fast annealing) and the SAGA approach.

In these cases, the message dispatcher architecture style did not appear in any of the solutions for either system. As for the patterns, the Adapter pattern was clearly the most popular in all the solutions for both systems. For the robo system, there were very few instances of other patterns; only a couple of Template Method or Strategy patterns could be found in the solutions. The solutions for robo seemed quite difficult to understand at a glance; the structure depends greatly on the null architecture, and as all classes are by default given an interface, the minimum amount of classes/interfaces is 44 for the robo system. When the patterns are added (even if only a few) the architecture easily becomes quite complex. The solutions for ehome were significantly easier to understand, as the amount of classes/interfaces that appear "by default" is roughly half the amount of classes for robo system. Curiously enough, there seemed to also be slightly more appearances of the Strategy and Template Method patterns in the ehome solutions than there were for robo, but the ehome solutions still seemed more understandable.

Thus, it appears that the SA by itself is incapable of introducing solutions that produce delayed reward, such as the message dispatcher architecture style. Also, even if the GA is able to introduce such solutions after being given the seed from the SA, it will take exceptionally long before the reward will overcome the cost, as the SA has already developed the solutions a great deal, and the GA may have to reverse the design process (i.e., apply the "remove" transformations) in order to apply needed changes. The results of merely SA based systems are, thus, unsatisfactory.

7. Conclusions and Future Work

We have presented an approach that uses SA in software architecture synthesis. A null architecture is given as input and architecture styles and design patterns are used as transformations when searching for a better solution in the neighborhood. The solution is evaluated with regard to modifiability, efficiency and complexity. The experimental results achieved with this approach show that SA on its own is not able to produce good quality solutions in terms of quality values or the resulting UML class diagrams. Attempts of improving the SA based solution with GA were also unsuccessful in increasing the quality values. However, when combining GA and SA so that the SA fine-tunes a basic solution achieved with the GA, both the quality values and the class diagrams are very good. Moreover, as SA is significantly faster than the GA, the result was obtained much quicker than would have been possible by using only GA. Thus, it is concluded that while SA is not sophisticated enough to be able to introduce complex alterations that require several transformations and produce delayed reward, it is able to quickly improve solutions where the base for such alteration has already been made.

It should be noted though, that SA seems to act very “single-mindedly”. When SA was used on its own, no solutions contained the message dispatcher architecture style. When SA was used after the GA, all the solutions used the message dispatcher architecture style very heavily, whether it was actually desired or not. Thus, it appears that the mechanism in SA that should prevent it from being stuck to a local optimum is not sufficient to divert the search in the case of software architecture synthesis.

In our future work we will concentrate on practical issues, and improve our basic implementation so that patterns (which are currently hardcoded), could be added at will. This will significantly increase the search space, but will also make the need for an algorithm to handle a large amount of patterns even greater. Moreover, the larger the system is and the more computation is required, the more there will also be need for a way to quicken the evolutionary process.

Acknowledgments

This work was funded by the Academy of Finland (project Darwin). The authors would like to thank professor Kai Koskimies for helpful discussions.

References

[1] A. Aleti, S. Björnander, L. Grunske, I. Meedeniya, ArcheOptrix: an extendable tool for architecture optimization of AADL models, In: *Proceedings of ICSE’09 Workshop on Model-Based Methodologies for Pervasive and Embedded Software*, 2009; 61-71.

[2] M. Amoui, S. Mirarab, S. Ansari, C. Lucas, A genetic algorithm approach to design evolution using design pattern transformation, *International Journal of Information Technology and Intelligent Computing* **1**, 2006; 235-245.

[3] C.R. Aragon, D. S. Johnson, L.A. McGeoch, C. Schevon, Optimization by simulated annealing: An experimental evaluation; part II, graph coloring and number partitioning, *Operations Research*, **39**(3), 1991; 378-406.

[4] J. Bansiya, C. G. Davis, A hierarchical model for object-oriented design quality assessment, *IEEE Transactions on Software Engineering*, **28** (1), 2002; 4–17.

[5] T. Bodhuin, M. Di Penta, L. Troiano, A search-based approach for dynamically re-packaging downloadable applications, In: *Proceedings of the Conference of the Center for Advanced Studies on Collaborative Research (CASCON’07)*, 2007; 27-41.

[6] S. Bouktif, H. Sahraoui, G. Antoniol, Simulated annealing for improving software quality prediction, In: *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2006)*, 2006; 1893-1900.

[7] M. Bowman, L. C. Briand, Y. Labiche, Solving the class responsibility assignment problem in object-oriented analysis with multi-objective genetic algorithms, *IEEE Transaction on Software Engineering* **36**, 6, 2010, 817-837.

[8] S. R. Chidamber, C. F. Kemerer, A metrics suite for object oriented design, *IEEE Transactions on Software Engineering* **20**(6), 1994; 476-492.

[9] J. Clarke, J. J. Dolado, M. Harman, R. M. Hierons, B. Jones, M. Lumkin, B. Mitchell, S. Mancoridis, K. Rees, M. Roper, M. Shepperd, Reformulating Software Engineering as a Search Problem, *IEE Proceedings - Software*, **150** (3), 2003; 161-175.

[10] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[11] N. Gold, M. Harman, Z. Li, K. Mahdavi, A search based approach to overlapping concept boundaries, In: *Proceedings of the 22nd International Conference on Software Maintenance (ICSM 06)*, 2006; 310-319.
[12] M. Harman, S.A. Mansouri, Y. Zhang, Search based software engineering: a comprehensive review of trends, techniques and applications, Technical report TR-09-03, King's College, London, United Kingdom, 2009.
[13] M. Harman, L. Tratt, Pareto optimal search based refactoring at the design level, In: *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2007)*, 2007; 1106–1113.
[14] A. C. Jensen, B. H. C. Cheng, On the use of genetic programming for automated refactoring and the introduction of design patterns, In: *Proceedings of the 2010 Genetic and Evolutionary Computation Conference (GECCO'10)*, 2010; 1341-1348.
[15] D. S. Johnson, C.R. Aragon, L. A. McGeoch, C. Schevon, Optimization by simulated annealing: An experimental evaluation; part I, graph partitioning, *Operations Research*, **37**(6), 1989; 865-892.
[16] S. Kirkpatrick, C. Gelatt, M. Vecchi, Optimization by simulated annealing, *Science*, **220**, 1983; 671-680.
[17] P.J.M van Laarhoven, E. Aarts, *Simulated Annealing: Theory and Applications*, Kluwer, 1987.
[18] N. Metropolis, A.W. Rosenbluth, M.N. Rosenbluth, A. H. Teller, E. Teller, Equation of state calculation by fast computing machines, *Journal of Chemical Physics*, **21**, 1953; 32-40.
[19] Z. Michalewicz, *Genetic Algorithms + Data Structures = Evolutionary Programs*, Springer-Verlag, 1992.
[20] M. Mitchell, *An Introduction to Genetic Algorithms*, MIT Press, 1996.
[21] M. O'Keeffe, M. Ó Cinnéide, Towards automated design improvements through combinatorial optimization, In: *Workshop on Directions in Software Engineering Environments (WoDiSEE2004), W2S Workshop − 26th International Conference on Software Engineering,* 2004; 75-82.
[22] M. O'Keeffe, M. Ó Cinnéide, Search-based software maintenance, In: *Proceedings of CSMR'06*, 2006; 249-260.
[23] M. O'Keeffe, M. Ó Cinnéide, Search-based refactoring for software maintenance, *Journal of Systems and Software,* **81** (4), 2008; 502-516.
[24] O. Räihä, A survey on search-based software design, *Computer Science Review*, **4**(4), 2010; 203-249.
[25] O. Räihä, K. Koskimies, E. Mäkinen, Genetic synthesis of software architecture, In: *Proceedings of the 7th International Conference on Simulated Evolution and Learning (SEAL'08)*, Springer LNCS 5361, 2008; 565-574.
[26] O. Räihä, K. Koskimies, E. Mäkinen, Scenario-based genetic synthesis of software architecture, In: *Proceedings of the 4th International Conference on Software Engineering Advances (ICSEA'09)*, 2009; 437-445.
[27] O. Räihä, K. Koskimies, E. Mäkinen, Empirical study on the effect of crossover in genetic software architecture synthesis, In: *Proceedings of the World Congress on Nature and Biologically Inspired Computing (NaBIC'09)*, IEEE, 2009; 619-625.

[28] O. Räihä, K. Koskimies, E. Mäkinen, Complementary crossover for genetic software architecture synthesis, In: *Proceedings of the 10th International Conference on Intelligent Systems Design and Applications (ISDA'10)*, 2010; 359-366.
[29] O. Räihä, K. Koskimies, E. Mäkinen, Multi-objective Genetic Synthesis of Software Architecture, manuscript.
[30] O. Räihä, K. Koskimies, E. Mäkinen, T. Systä, Pattern-based genetic model refinements in MDA, *Nordic Journal of Computing,* **14** (4) 2008; 322-339.
[31] O. Seng, M. Bauyer, M. Biehl, G. Pache, Search-based improvement of subsystem decomposition, In: *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2005),* 2005; 1045-1051.
[32] O. Seng, J. Stammel, D. Burkhart, Search-based determination of refactorings for improving the class structure of object-oriented systems, In: *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2006)*; 2006, 1909-1916.
[33] M. Shaw, D. Garlan, *Software Architecture − Perspectives on an Emerging Discipline,* Prentice Hall, 1996.
[34] C. L. Simons, I. C. Parmee, Single and multi-objective genetic operators in object-oriented conceptual software design, In: *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'07),* 2007;1957-1958.
[35] C. L. Simons, I. C. Parmee, A cross-disciplinary technology transfer for search-based evolutionary computing: from engineering design to software engineering design, *Engineering Optimization*, **39** (5) 2007; 631-648.
[36] E. Trikia, Y. Colletteb, P. Siarry, A theoretical study on the behavior of simulated annealing leading to a new cooling schedule, *European Journal of Operational Research*, **166,** 2005; 77-92.