Acta Cybernetica **21** (2013) 235–265.

Simulated Annealing for Aiding Genetic Algorithm in Software Architecture Synthesis*

Outi Sievi-Korte[†], Erkki Mäkinen[‡], and Timo Poranen[‡]

Abstract

The dream of software engineers is to be able to automatically produce software systems based on their requirements. Automatic synthesis of software architecture has already been shown to be feasible with genetic algorithms. Genetic algorithms, however, easily become very slow if the size of the problem and complexity of mutations increase as GAs handle a large population with much data. Also, for purely scientific interest it is worthwhile to investigate how other search algorithms handle the problem of software architecture synthesis. The present paper studies the possibilities of using simulated annealing for synthesizing software architecture. For this purpose we have two goals: 1) to study whether a simpler search algorithm can handle synthesis and 2) if a seeded algorithm can provide quality results faster than a simple genetic algorithm. We start from functional requirements which form a base architecture and consider three quality attributes, modifiability, efficiency and complexity. Synthesis is performed by adding design patterns and architecture styles to the base architecture. The algorithm thus produces a software architecture which fulfills the functional requirements and also corresponds to the quality requirements. 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 seed solution achieved with a genetic algorithm. The main contribution is thus a new seeded algorithm for software architecture design.

Keywords: search-based software engineering, simulated annealing, software design, genetic algorithm, software architecture

1 Introduction

The ultimate goal of software engineering is to be able to automatically produce software systems based on their requirements. In Model Driven Architecture

*This work was funded by the Academy of Finland (project Darwin).

[†]Tampere University of Technology - corresponding author, E-mail: outi.sievi-korte@tut.fi, Address: Department of Software Systems, Korkeakoulunkatu 1, P.O.Box 553, 33101 Tampere, Finland

[‡]University of Tampere, E-mail: {erkki.makinen, timo.t.poranen}@uta.fi

DOI: 10.14232/actacyb.21.2.2013.3

(MDA), class level designs of the software can already be transformed straightforwardly into code [10]. However, a human is still required to interpret the given quality requirements and build the class level design, or architecture, based on which code can be written. This process is time consuming and requires expertise, as systems become increasingly large and complex and the quality requirements are often conflicting. Errors in design phase are unfortunately common, and have a large impact on the functionality of the system. Our goal is to automate the process of turning requirements into software architecture, where quality requirements are not only met but also optimized to suit the preferences of the client.

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 [39] and design patterns [11]. We argue that software architecture comes in parts: the functional requirements, the quality requirements and the actual architectural design solutions. The functional and quality requirements can only be elicited manually, but combining them to design solutions and, thus, producing a complete architecture, which is more than the sum of its parts, can be done automatically. Hence, we see the formation of software architecture as a series of transformations beginning with a very crude outline of a system with only the basic functionalities and ending with a highly sophisticated design. So far, this has been accomplished by humans. Thus, as we view that software architecture requires combining different entities (design solutions and requirements) and the automatic process is synthetic when compared to man made architectures, we refer to our approach as architecture synthesis.

Seeing software architecture as a combination of design solutions makes it an optimization problem — what is the best way of combining the solutions, with respect to quality requirements? Search-based software engineering (SBSE) studies the application of meta-heuristic algorithms to such software engineering problems [9]. In this field, genetic algorithms (GAs) [22] have been shown to be a feasible method for producing software architectures from functional requirements [29, 33, 34]. However, experiments with asexual reproduction [30] suggest that the crossover operator which is an essential part of GAs might not be critical for producing good architectures, supporting the idea of using a simpler search method. Additionally, the GA easily becomes very slow if the system is large, or if the search leans heavily towards certain mutations (due to preferences of the architect). These heavy mutations combined with a large system meant that the GA, which has to handle an entire population of solutions simultaneously, had to deal with a massive amount of data. It is, thus, natural to ask if other (lighter) 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 (SA) in the process of searching good architectures when functional requirements are given.

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. Contrary to GAs, 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. Contrary to, for example, hill climbing algorithms, SA does allow also temporarily exploring worse solutions than what have been found so far. Due to the nature of the fitness landscape (many small peaks and large dips which lead to high peaks), this is essential in eventually finding a good architecture. Our decision to study SA first is also backed up by the promising studies in related fields where SA has been used for software refactoring [23]. Results from our studies conducted with SA will give us further information on what is required from the synthesis, and we may then possibly study other algorithms, such as particle swarm optimization and ant colony optimization.

It is known that seeding GAs enable them to produce better results faster [17, 36]. Our hypothesis is that a SA algorithm could also be used to quickly produce a seed. An initial population can be generated based on this seed. A significantly smaller number of generations would then suffice to find good solutions with the GA.

As with our GA approach, we begin with the functional requirements of a given 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. As the SA is implemented as close to the GA as possible, our set of research questions thus becomes: How good are the architectures produced by SA (compared to GA)? What kind of fitness values does SA achieve (compared to GA)? How fast is SA (compared to GA)? And finally, how well does a seeded algorithm perform in terms of both quality and speed (compared to GA)?

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 and give the algorithmic presentation for our GA, to be used in the experiments. 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

SBSE considers software related topics as combinatorial search problems. Traditionally, testing has been the clearly most studied area inside SBSE [13]. Other well studied areas include software clustering and refactoring [9, 13, 26]. Using meta-heuristic 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. Approaches dealing with higher level structural units, such as patterns, have also recently gained more interest. We will briefly discuss the studies with the closest relation to our approach. As our method in part combines two algorithms, and the result can be viewed as a seeded algorithm (either SA provides a seed for the GA or vice versa), we will also briefly discuss approaches using seeding.

Amoui et al. [2] use the GA approach to improve the reusability of software by applying architecture design patterns to a UML model. Their goal is to find the best sequence of transformations, i.e., pattern implementations. Used patterns come from the collection presented by Gamma et al. [11]. 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 uses one quality factor (reusability) only, while we use three quality factors, 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. Thus, their approach currently works for refactoring only, and is not able to do forward design, which is our aim.

Simons et al. [44] study using evolutionary, multi-objective search and software agents to aid the software architect in class design. One individual (solution) is thus the design containing all methods and attributes (and their class distribution). Coupling and cohesion are used to calculate fitness. Simons et al. suggest that a global multi-objective search is unnecessary, and the search should be narrowed towards the ”most useful and interesting candidate designs”. They attempt to achieve this by isolating discrete zones from the search space, and then using a local search within these zones. Local search is conducted using a single-objective genetic algorithm, which only considers coupling in the fitness calculations. The designer then obtains the results of these local searches. Simons and Parmee [43] have further enhanced their studies with elegance metrics, which should conform to the desire for symmetry that human designers have.

Räihä et al. [29] have taken the design of software architecture a step further than Simons and Parmee [40, 41] 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. Mutations are implemented as adding or removing an architectural design pattern [11] or an interface or splitting or joining class(es). Implemented design patterns are Façade and Strategy, as well as the message dispatcher architecture style [39].

Räihä et al. [34] have also applied GAs in model transformations that can be understood as pattern-based refinements. In MDA, 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. Räihä et al. [31] have also later on added scenarios, which are common in real world architecture evaluations, to evaluate the fitness of their synthesized architectures. Our work differs from the work of Räihä et al. [31] by using simulated annealing in addition to GA.

Jensen and Cheng [15] present an approach based on genetic programming (GP) for generating refactoring strategies that introduce design patterns. They have implemented a tool, RE-MODEL, 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. [11] is used to identify desirable mini-transformation sequences. Mutations are applied by simply changing one node (transformation), and crossover is applied as exchanging subtrees. 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 close to those of Räihä et al. [29] and the approach used here, the difference being that Jensen and Cheng have clearly a refactoring point of view, while we attempt upstream synthesis, thus expecting less from the architect and relying more on the algorithm, which makes our problem setting far more complex. Our fitness metrics are also different, as we only reward patterns that clearly improve the design — the simple existence of a pattern is not a reason for reward itself.

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 [12] 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. [30, 32].

SA has been used in the field of search-based software engineering for software refactoring [23, 24, 25] and quality prediction [6]. O'Keeffe and Ó Cinnéide [23, 24, 25] 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. [37, 38] who use GA as their search algorithm and Harman and Tratt [14], 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.

In UML software design SA has been used in the context of dynamic parameter control in interactive local search by Simons and Parmee [42]. The level of design is quite similar, as it also deals with classes, methods and attributes. In this study the approach using simulated annealing was shown to be inferior to other method used in parameter control, while dynamic parameter control in general proved to be an efficient way for improving the results. Our approach differs significantly from that of Simons and Parmee, as we use simulated annealing itself in a different way (as the actual search algorithm itself, as opposed to controlling the parameters). We also have a very different mutation setting and problem domain. We have sixteen mutations, while there were only a couple in the presented study, thus the setting for dynamically controlling all the probabilities is much more complex, though we acknowledge the idea worth pursuing (initial experiments with a similar idea have been done in our previous work [34]). All in all, the studies using SA are few, and none use this approach for such a high-level design problem as designing software architecture from requirements

Our approach of combining SA and GA can be seen as a seeded algorithm, as one algorithm provides a developed seed for the other. Julstrom [17] has used the idea of seeding the initial population of a GA with advanced individuals in the rectilinear Steiner problem. The seeded algorithm produced more consistent results and was significantly faster than the algorithm with a randomly created initial population. Ramsey and Greffenstett [36] have studied case-based initialization of GAs in learning systems. In their study, the population of the GA is dynamically initialized with achieved (good) results, which aids in (intentionally) biasing the search towards a certain area, and quickly answering to a changing environment.

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. [20] and later Kirkpatrick et al. [18] generalized the idea for combinatorial optimization.

Algorithm 1 simulatedAnnealing

```
1: Input: Responsibility dependency graph $G$, base architecture $M$, initial temperature $t_0$, frozen temperature $t_f$, cooling ratio $\alpha$, and temperature constant $r$
2: Output: UML class diagram $D$
3: $initialSolution \leftarrow$ encode$(G, M)$
4: $initialQuality \leftarrow$ evaluate$(initialSolution)$
5: $S_1 \leftarrow initialSolution$
6: $Q_1 \leftarrow initialQuality$
7: $t \leftarrow t_0$
8: while $t > t_f$ do
9:    $r_i \leftarrow 0$
10:   while $r_i < r$ do
11:      $S_i \leftarrow$ transform$(S_1)$
12:      $Q_i \leftarrow$ evaluate$(S_i)$
13:      if $Q_i > Q_1$ then
14:         $S_1 \leftarrow S_i$
15:         $Q_1 \leftarrow Q_i$
16:      else
17:         $\delta \leftarrow Q_1 - Q_i$
18:         $p \leftarrow$ UniformProbability
19:         if $p < e^{\frac{-\delta}{t}}$ then
20:            $S_1 \leftarrow S_i$
21:            $Q_1 \leftarrow Q_i$
22:         end if
23:      end if
24:      $r_i \leftarrow r_i + 1$
25:   end while
26:   $t \leftarrow (1 - \alpha) \times t$
27: end while
28: $D \leftarrow$ generateUML$(S_1)$
29: return $D$
```

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 temperature t is decreased to the frozen temperature t_f, where $0 \leq t_f \leq t_0$. The temperature gives the

Algorithm 2 geneticAlgorithm

```
1: Input: formalization of solution, $initialSolution$
2: $population$ ← createPopulation($initialSolution$)
3: while NOT terminationCondition do
4:    for all $chromosome$ in $population$ do
5:       $p$ ← randomProbability
6:       if $p > mutationProbability$ then
7:          mutate($chromosome$)
8:       end if
9:    end for
10:   for all $chromosome$ in $population$ do
11:      $cp$ ← $randomProbability$
12:      if $cp > crossoverProbability$ then
13:         addToParents($chromosome$)
14:      end if
15:   end for
16:   for all $chromosome$ in $parents$ do
17:      $father$ ← $chromosome$
18:      $mother$ ← selectNextChromosome($parents$)
19:      $offspring$ ← crossover($father$, $mother$)
20:      addToPopulation( $offspring$)
21:      removeFromParents ($father$, $mother$)
22:   end for
23:   for all $chromosome$ in $population$ do
24:      calculatefitness($chromosome$)
25:   end for
26:   selectNextPopulation()
27: end while
```

probability of choosing solutions that are worse than the current solution. The result of a transformation that worsens the current solution by d, 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.

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 a ($0 < a < 1$). This is the most frequently used schedule [45]. 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 [45].

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, 16]. In order to determine good parameters for a problem, experimental analysis is often needed. There are also adaptive techniques for detecting the parameters [19]. The SA implementation used in our tests is shown in Algorithm 1. The encoding, transformation and evaluation procedures are discussed in more detail in Section 4. Notice, that our SA only operates with a single solution at a time, and the solution is built by transformations (i.e., moving towards better neighbors).

In Section 5 we compare the present SA and our previous GA implementation [29]. We assume the reader has knowledge of the basic principles of GA, as given by, e.g., Michalewicz [21]. The GA implementation used is given in Algorithm 2. Mutation is executed in the same way as a transformation for simulated annealing, and more details will be given in Section 4. Crossover is a single-point random crossover and selecting the next population is made with a rank-based roulette wheel selection. For more details on how crossover and selection is implemented in our approach, we refer to [27].

The result of GA is the best solution found during the search process. Thus, in that sense, both SA and GA are single solution algorithms and their comparison is straightforward. 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 [29, 31, 33, 34]. The initial solution is achieved by encoding functional requirements and thus building a base architecture. The base class structure is derived from the base 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 [29, 31, 33, 34], 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. Use cases are an intuitive starting point in most software projects, and little domain knowledge is required to define them. Thus, use cases are a natural way to begin eliciting the functional requirements of a system. Use cases can, in turn, be refined into sequence diagrams. The refining process requires some effort from the architect but still quite little domain knowledge and is still fairly intuitive, as the architect simply needs to think how different use cases proceed on operation level. From sequence diagrams it is simple to elicit classes (the participants/owners' of lifelines) and operations (calls in the diagram). This results in a base architecture, giving a structural view of the functional requirements of the system at hand but not dealing with the quality requirements. The base 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; the process is similar in the case of robo.

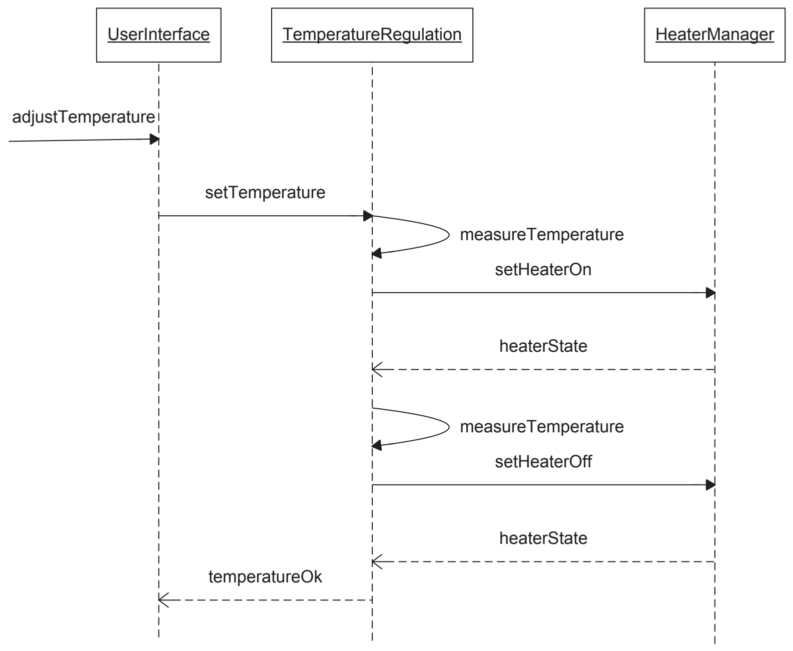


Figure 1: Adjust temperature use case refined

Specifying requirements begins with giving use cases. 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. Here, we will take as an example the adjust room temperature use case. The user simply places a command that the temperature should be adjusted (for the sake of simplicity, we can here consider elevation), and ehome adjusts the temperature by turning on the heater.

The sequence diagram for the temperature adjustment use case is given in Figure 1. The process begins with a call from the user to set the temperature to a new level. The system then calls the temperature regulation component, which measures the current temperature, and then sets the heater on. After the correct temperature is reached, the heater is turned off.

While sequence diagrams already give a good understanding of how the different operations depend on each other, a structural view still needs to be obtained, as patterns cannot be inserted into sequences of calls. Fortunately, sequence diagrams can easily be turned into class diagrams. At this point, the class diagram would not consist of anything but the classes, their methods and attributes, and connections between classes, as defined in the sequence diagram. We have chosen to use sequence diagrams as the basis, as they can be straightforwardly build based on use case diagrams, and use case diagrams are the most intuitive way to start formulating the requirements.

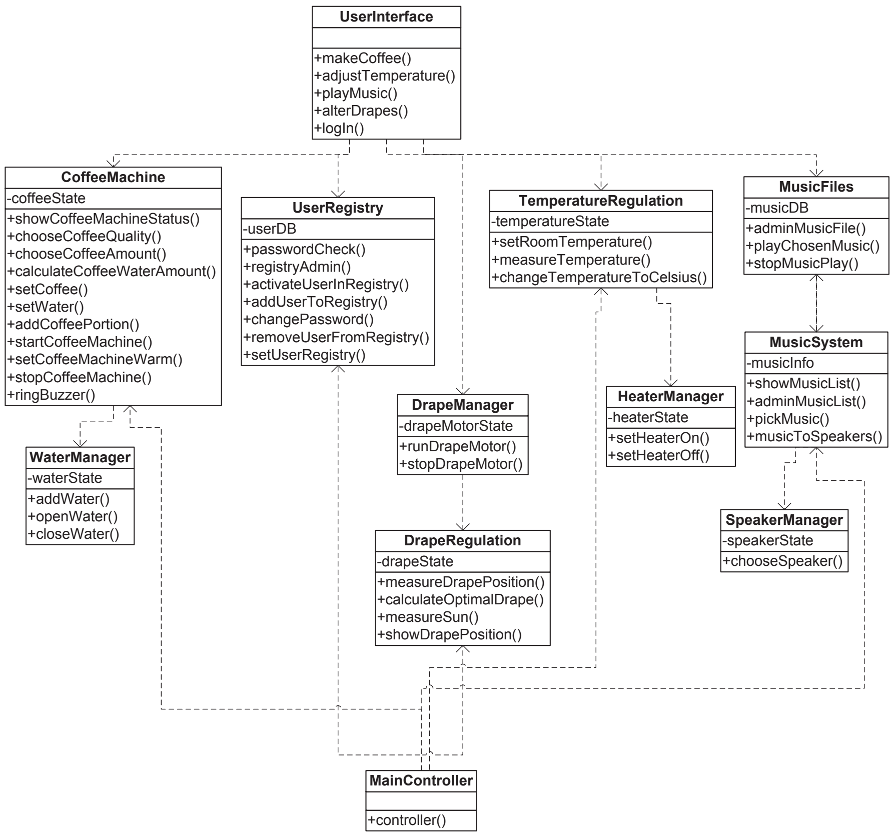


Figure 2: Base architecture for ehome

The base architecture in Figure 2 for the ehome system can be straightforwardly derived from the sequence diagrams. We depict architecture as a class diagram, as we consider the architecture to be the classes or components of a system the interfaces and other communication mechanisms between them. Thus, a class view is natural for our purposes.

The messages in the sequence diagram become the operations and the objects/components become the classes. Also, if the need for a data source is detected or the object clearly has a state, they will become attributes in the classes. For example, in Figure 1 both the `Temperature Regulation` and `Heater Manager` have states, such as on or off for the `Heater Manager`. The base architecture only contains use relationships, as no more detail is given for the algorithm at this point. The base architecture represents the basic functional decomposition of the system. A base architecture for robo (which can be achieved by performing the same steps as did with ehome) is given in Figure 3.

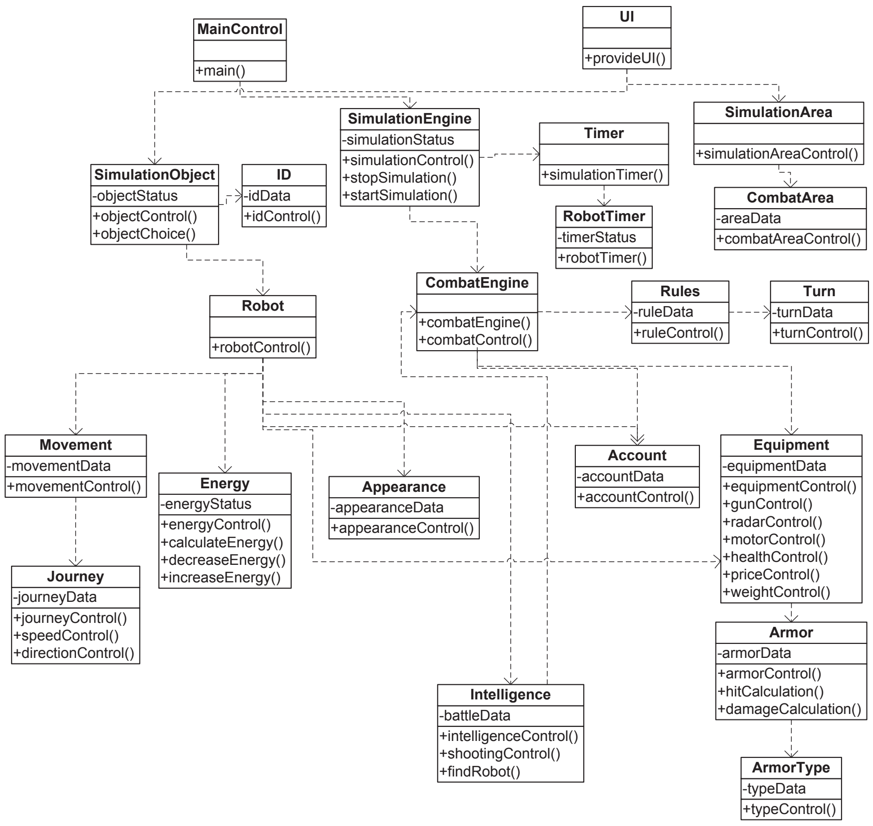


Figure 3: Base architecture for robo

After the operations are derived from the use cases, some properties of the operations can be estimated to support the 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 (called hereafter the variability of an operation) than ringing the buzzer when the coffee is done. Measuring the position of drapes requires more information than running the drape motor (which can be interpreted as the required parameter size), and playing music quite likely has a higher usage frequency than changing the password for the system. Relative values for the chosen properties can similarly be estimated for all operations. Here we have used the scale of Low (1), Medium (3) and High (5). This optional information, together with operation call dependencies, is included in the information subjected to encoding.

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 ("f" as in functional for methods, "d" as in data for attributes), frequency f_i, parameter size p_i and variability v_i. Secondly, there is the information regarding 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 $OD_i \subseteq (O_i)$ 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 pre-determined base architecture class MC_i. The dispatcher is given a separate field as opposed to other patterns for efficiency reasons.

The base architecture is encoded as a vector $V < ov_1, ov_2, \ldots, ov_n >$ of vectors $ov_1, ov_2, \ldots, ov_n$ for the algorithm. 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_i. The same encoding works for both SA and GA. For GA, the chromosome is the vector V, and each vector ov_i is a supergene, which contains the fields described above.

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

Figure 4: Operation vector ov_i

We will give an example from the ehome system of how the given data structure works. In the base architecture phase, if the `TemperatureRegulation` class is given #ID 2 (and the interface #ID 2), for operation `measureTemperature` (#id 9) the ov_9 would have the following values: O_9 = {#idSetRoomTemperature}, n_9 = measureTemperature, d_9 = f, $p_9 = 3$, $f_9 = 1$, $v_9 = 3$, $C_9 = 2$, $I_9 = 0$, $D_9 = 0$, $OD_9 = \emptyset$, $MC_9 = 2$, $P_9 = \emptyset$. The interface has value 0, as `measureTemperature` is only required by `setRoomTemperature`, which is in the same class, and thus does need an interface to access this operation. The fields for message dispatcher and pattern have $\emptyset$ values, as no architectural solutions are included in the base architecture. As the operation is here located in the original base architecture class, the values for C and MC are the same. Note, that the encoding is indeed operation-centered. Thus, modifications to the architecture are considered from the viewpoint of how a particular operation can be accessed, and not how two classes communicate with each other. In practice, the base architecture is encoded into a text file, which is given as input for the algorithm, with each operation in its own line.

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 [39] (message dispatcher and client-server), medium-level design patterns [11] (`Façade and Mediator`), and low-level design patterns [11] (`Strategy, Adapter` and `Template Method`). This selection of patterns and styles allows us to see how well the algorithm handles different types of changes. High-level patterns have a larger impact, as they usually affect large parts of the architecture, while lower level patterns only affect small parts. 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 pre-conditions. 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 base architecture class, and the pattern found in the supergenes 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 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 [33] to conquer this problem. However, when evaluating the applicability of simulated annealing, we found it more practical to use a single weighted fitness, as we wanted to maintain SA as "pure" as possible (local and efficient), even though there are multi-objective versions of SA as well (e.g., [46]).

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 class 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, if we want to. When w_i is the weight for the respective sub-function sf_i, the evaluation function $f_c(x)$ (which should be maximized) for solution x can be expressed as

$$f_c(x) = w_1 \times sf_1 - w_2 \times sf_2 + w_3 \times sf_3 - w_4 \times sf_4 - w_5 \times sf_5. \tag{1}$$

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):

$$sf_1 = |\textit{calls to interfaces}| \times \sum_{k=0}^{i}(v_k) + |\textit{calls through dispatcher}|) \times \sum_{k=0}^{d}(v_k),$$

$$\begin{aligned} sf_2 = {} & |\textit{direct calls between operations in different classes}| \times \sum_{k=0}^{c}(v_k)) \\ & + |\textit{calls between operations within same class}| \times \sum_{k=0}^{s}(v_k) \times 2, \end{aligned}$$

$$sf_3 = |\mathit{operations\ dependent\ of\ each\ other\ within\ same\ class}| \times \sum_{k=0}^{w}(p_k) + |\mathit{used\ operations\ in\ same\ class}| \times \sum_{k=0}^{u}(p_k) + |\mathit{depending\ operations\ in\ same\ class}| \times \sum_{k=0}^{e}(p_k),$$

$$sf_4 = \sum |\mathit{ClassInstabilities}| + (2 \times |\mathit{dispatcherCalls}| + |\mathit{serverCalls}|) \times \sum_{k=0}^{ds}(f_k) + |\mathit{calls\ between\ operations\ in\ different\ classes}|\,, \mathit{and}$$

$$sf_5 = |\mathit{classes}| + |\mathit{interfaces}|\,.$$

In sf_1, i is the number of operations called through an interface, d is the number of operations called through dispatcher, and v is the variability value of an operation (as in Fig. 4). The variability values v of those operations that are involved in interface or dispatcher calls, respectively, are summed. In sf_2, c is the number of calls from a different class to an operation with no interface and with variability value v_k, sc is the number calls from within the same class to an operation with variability value v_k. The calculation is similar to that in sf_1, as variability values of operations are summed if said operations are called based on given criteria. Calls within class are given a constant multiplier 2, as it is considered that a call within class bonds two operations and thus has double the negative effect on modifiability. The w, u and e in sf_3 are the numbers of the types of calls as specified in sf_3, similarly as in sf_1 and sf_2. In sf_3, however, the parameter size values p are summed instead of variability values. It should also be noted, that in sf_1, most patterns also contain an interface. In sf_3, "used operations in same class" means a set of operations in class C, which are all used by the same operation from class D. Similarly, "depending operations in same class" mean a set of operations in class K, which all use the same operations in class L. In sf_4, ds is the number of calls through dispatcher or server where the called operation's frequency value is f_k. The multiplier 2 for calls relayed by the message dispatcher is given as there are always two calls when the message dispatcher is used - one from the calling class to the dispatcher and one from the dispatcher to the receiving class.

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). The selected two systems are very different in nature and structure, which would lead to very different architectures. Choosing these two different systems shows that the algorithm is not confined to any particular system, but can be generally used for any type of system. Most of the parameters used in our tests originate from the previous tests reported in [29, 31, 34], 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, as we did not want to favor any particular quality attribute over another, but aimed for balanced designs. Also, by setting the weights to 1 we do not need to consider the effect of the weights in fitness curves.

The GA used in the combination experiments is based on our previous implementations [29, 31, 34]. The GA uses the same encoding, transformations (mutations) and quality function as defined here for the SA. As stated in Section 3, the crossover operator is a single-point random crossover and 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 [29, 31, 34, 27].

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). A lower starting temperature had also been tested previously with no obvious benefits [35]. The values were selected based on trial-and-error experiments. 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 initial value is the same as where the GA starts in the curves given in the following section). The high temperature tests for both systems took approximately 10 seconds per run and the fast annealing tests less than 5 seconds per run, standard test runtime is reported in the following. 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. Also in this case the end quality value was worse than the value for the initial solution. Again, the runtime for the SAGA seeded algorithm is reported in the following when compared to other approached. All experiments were run for 20 times.

5.2 Using a Combination of GA and SA

As using the SA alone or for producing a seed did not produce good results, we 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). The experiments were run for 20 times and presented fitness curves are the average curves of the 20 runs. We have chosen to show average curves, as we are, after all, interested in how the algorithms behave in general, not individual runs. 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.

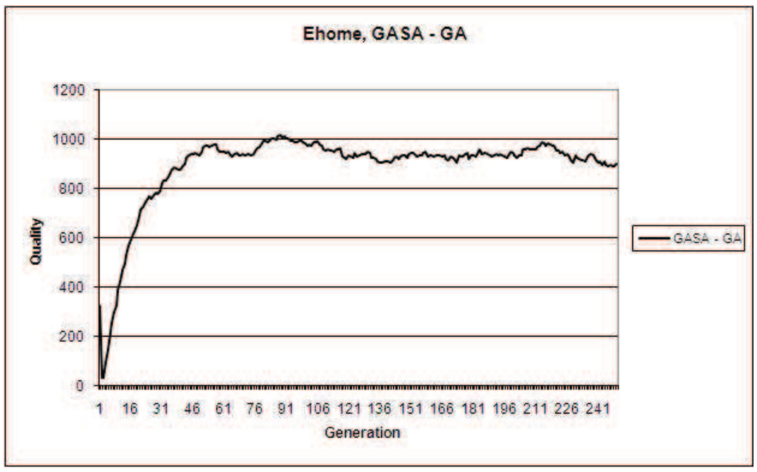


Figure 5: GA portion of GASA quality curve for ehome

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. The GA curves represent the average of the elite (top 10% of the population) (given as an average over the 20 runs), and the SA curves are naturally simply the average of fitness values of the 20 runs. 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 and because the GA curve is the average of elite, while SA handles only one solution.

As can be seen in Figure 5, the GA begins with a short plummet, after which the quality (fitness) begins to develop steadily. We expect the plummet to be an effect of using the message dispatcher very early on. When the message dispatcher is used sparingly (as is the case after only a few mutations), its penalty is greater than its reward. 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

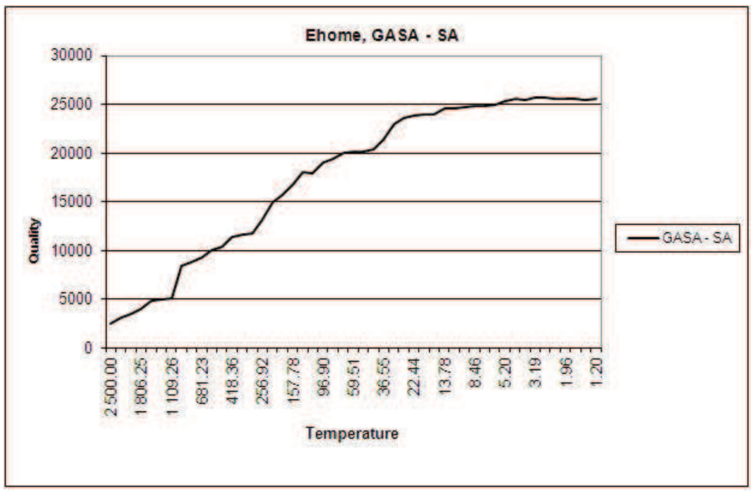


Figure 6: SA portion of GASA quality curve for ehome

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, appears 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 the curve has reached a plateau, giving reason to believe that some optimum has been found.

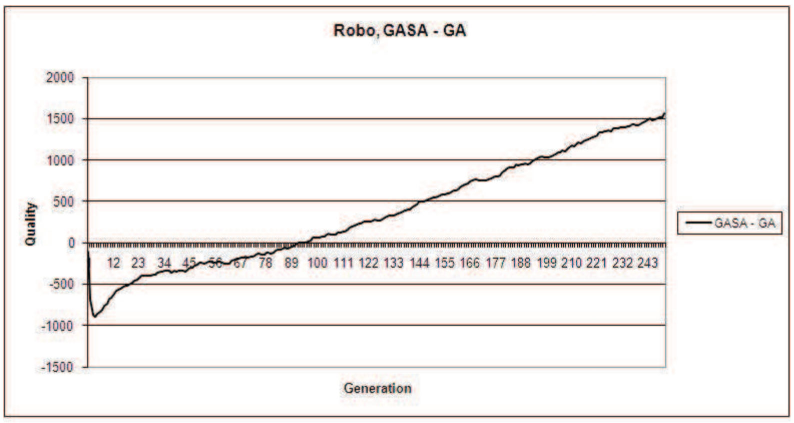


Figure 7: GA portion of GASA quality curve for robo

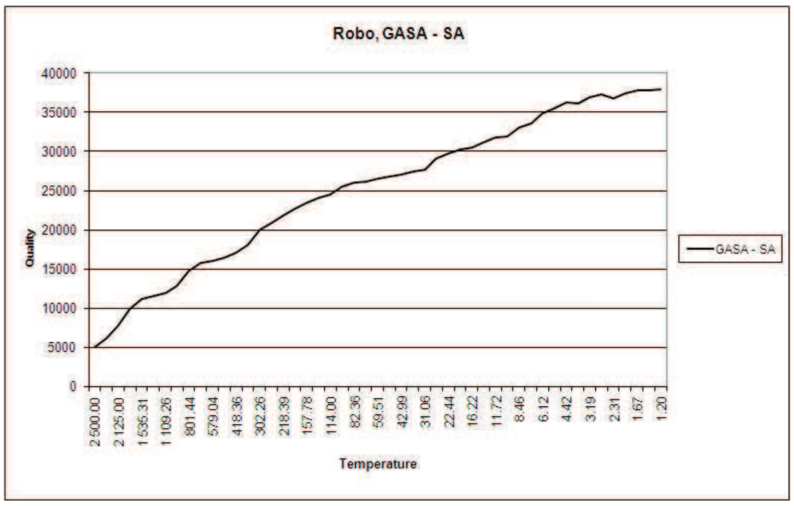
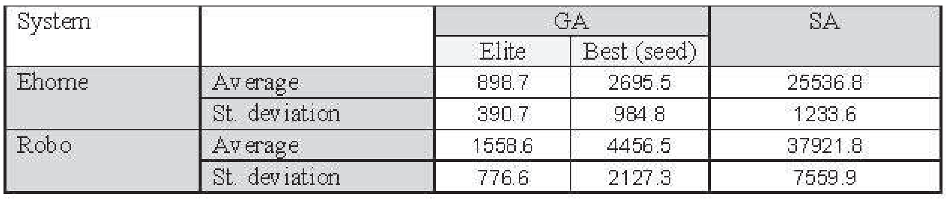


Figure 8: SA portion of GASA quality curve for robo

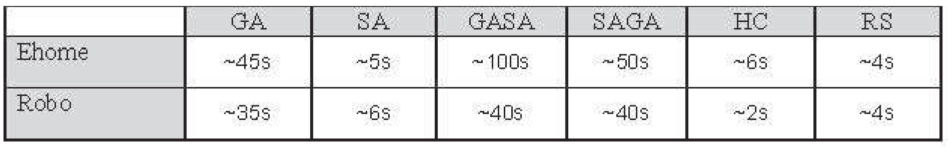
We have calculated the average fitnesses and standard deviations of GASA runs in Table 1. The average of the (averages of) elite is naturally the value where GA fitness curves end (Figures 5 and 7). The average of best (seed) is the average of the absolute best individuals provided by GA, which are then given as a seed for SA for further development. For SA we only have one value, as the algorithm only handles one individual at a time. From Table 1 we can see that the deviation especially in the case of GA is quite large, and the algorithm is not as stable as could be hoped. However, the deviation within the solutions after the SA (i.e., the final solutions from the seeded algorithm) is much smaller. There was no clear correlation between the elite fitness after GA and the fitness value after SA.



System		GA		SA
		Elite	Best (seed)	
Ehome	Average	898.7	2695.5	25536.8
	St. deviation	390.7	984.8	1233.6
Robo	Average	1558.6	4456.5	37921.8
	St. deviation	776.6	2127.3	7559.9

Table 1: Statistical markers

Finally, we have compared the runtimes of GA and SA and their combinations to random search (RS) and hill climbing (HC). The runtimes have been collected in Table 2. RS was run for 3500 iterations (same amount of iterations as SA with standard parameters) and HC was allowed 150 attemps of finding a neighbor after each ascent. All algorithms were run 20 times. The average fitness value achieved with RS was -1915 for ehome and -4266 for robo. For HC, in 50% of the cases the algorithm only managed to ascend once, after which the algorithm terminated as 150 attempts at finding a better neighbor failed. In the rest of the cases HC was able to ascend 6-12 times. Due to such inconsistent results, average fitness values do not provide good information. However, HC is clearly not a suitable algorithm for this problem. As can be seen in Table 2, all local searches are significantly faster than GA or combinations with GA. It should be noticed, though, that the GA performs at least 25000 fitness evaluations (100 in a population times 250 generations and additionally the evaluations of offspring), while the SA only performs 3500 fitness evaluations (with the selected parameters). Also, crossover is a very time consuming operation for the GA. As a conclusion, GA and GASA are clearly the slowest algorithms, but produced just as clearly the best results.



	GA	SA	GASA	SAGA	HC	RS
Ehome	~45s	~5s	~100s	~50s	~6s	~4s
Robo	~35s	~6s	~40s	~40s	~2s	~4s

Table 2: Runtimes for different algorithms

6 Discussion

In Section 5 we discussed the quality curves of the experiments made with the SA algorithm. Naturally, the 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 high-level 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. As the format is free form, we have not included class relations, but simple use relations only. There are no methods or attributes present in the solutions that were not there in the base architecture.

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 base architecture classes) was handled through it. 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 the 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.

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, as the message dispatcher is then used to its full potential and enables independency between components. The `Adapters` for `Water Control` and `Speaker Manager` are also particularly well placed, as these components are intuitively such that they could be replaced with new ones (in an ehome we may want to change the water faucet or upgrade to better speakers without changing the underlying kitchen or music systems), and thus the interface might change. The `Template Methods` for `Coffee Machine, Temperature Regulation` and `Music System` are also well chosen, as the specialized operations are such that alternative versions are easily conjured. Other `Adapters, Template Methods` and `Strategies` are acceptable, but a human designer would probably not apply them.

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. 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 mes-

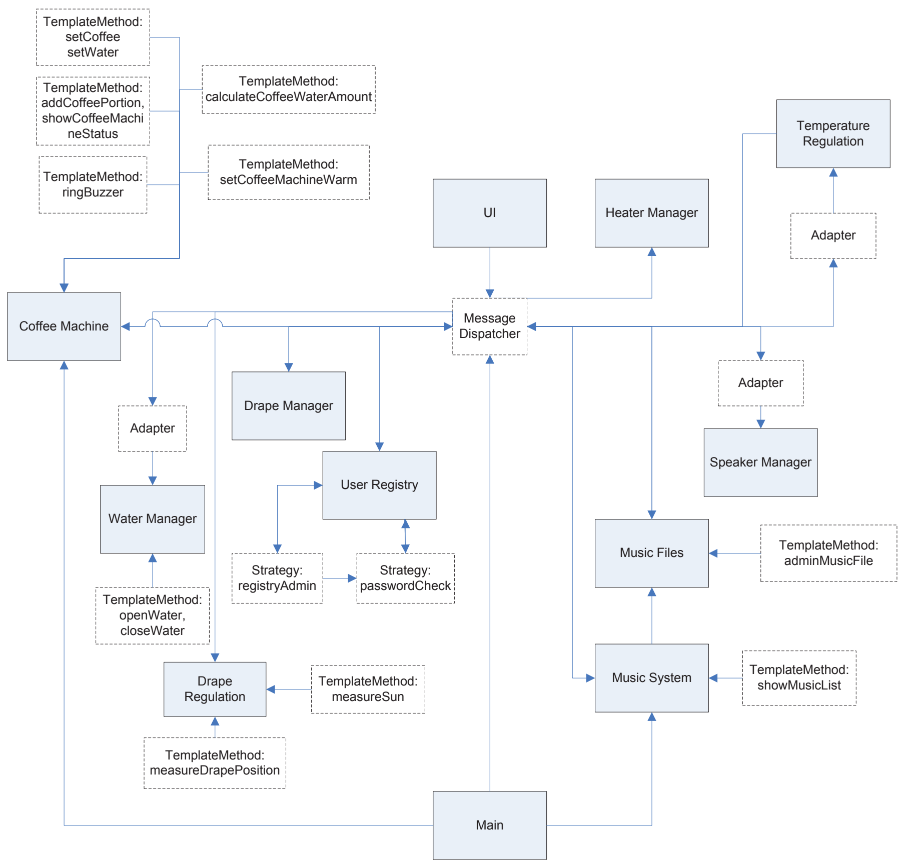

TemplateMethod: setCoffee setWater
TemplateMethod: addCoffeePortion, showCoffeeMachineStatus
TemplateMethod: ringBuzzer
TemplateMethod: calculateCoffeeWaterAmount
TemplateMethod: setCoffeeMachineWarm
Coffee Machine
UI
Heater Manager
Temperature Regulation
Adapter
Message Dispatcher
Adapter
Drape Manager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

Figure 9: Example architecture for ehome, with GASA algorithm combination

sage 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. The SA (or GA), however, does not have such high-level knowledge of the type of system it is dealing with and bases the design simply on the quality values, which are achieved from general structural decisions only. For robo, 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. In the proposed solution for robo, the `Template Method` and `Strategy` patterns are all intelligently used, as they consider operations and classes where the need for specialization is easily

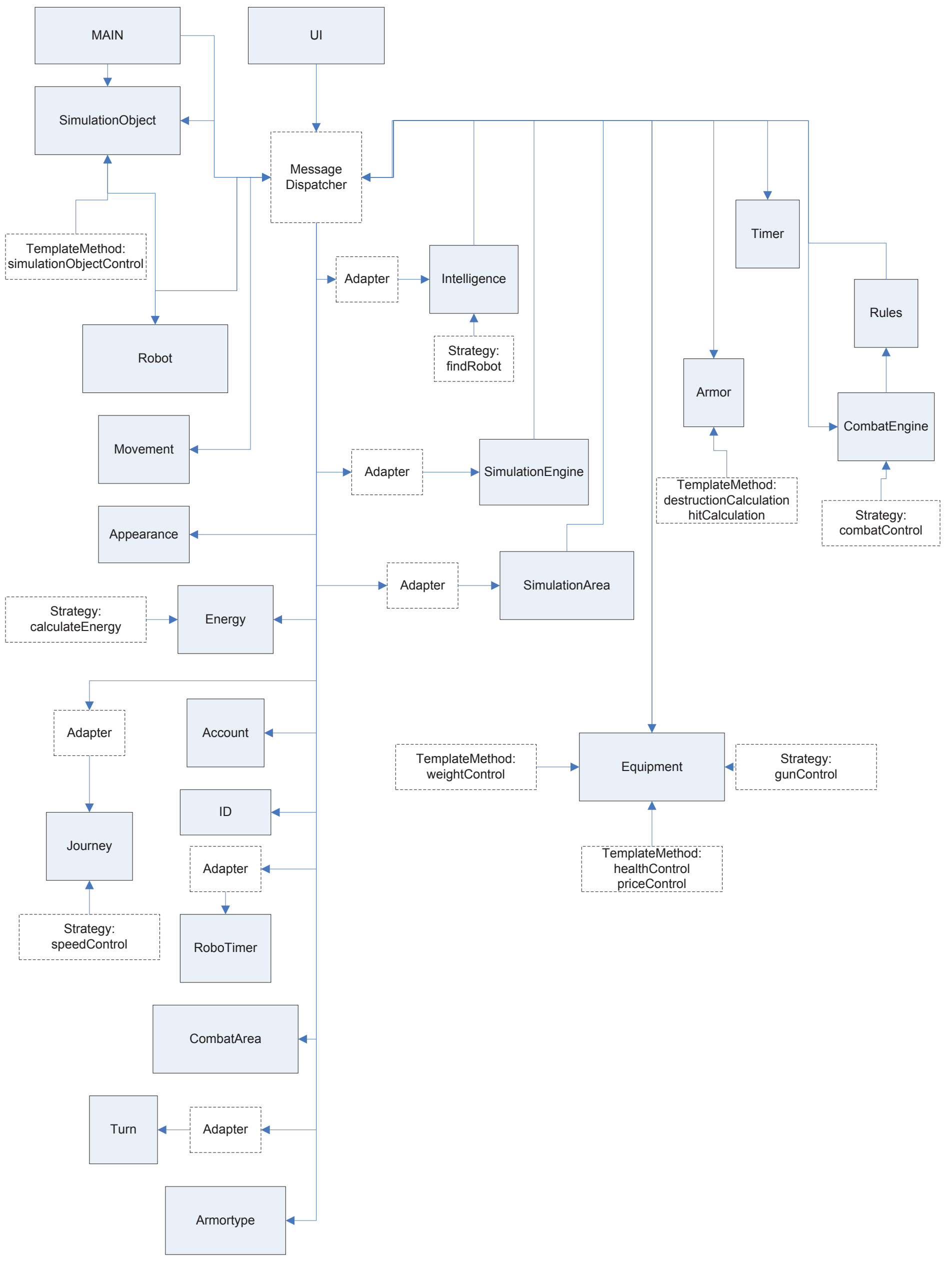


Figure 10: Example architecture for robo, with GASA algorithm combination

seen. The `Adapters`, however, are not used particularly well. 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 base 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 in robo, but the ehome solutions still seemed more understandable.

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 base 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.

When compared to the manual process, any of the presented algorithms (GA, SA, GASA or SAGA) performs significantly faster than a human designer. A human designer would need several hours to peform the design task, while our algorithms manage in mere minutes. In terms of quality, the GA and GASA come quite close to results from a human designer. Previous studies have shown that GA is at a level of a college student [28], and GASA manages to produce better quality and faster results. Thus, in relation to the ultimate goal of automating software engineering, this paper brings us closer to that goal by providing a more efficient way of automating software architecture design while also producing better quality results than what have been previously achieved with GA alone.

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. Thus, we will also be doing experiments on very large systems to further see how much the seeded algorithm can outperform the GA in terms of time.

8 Acknowledgements

The authors would like to thank professor Kai Koskimies for helpful discussions and the anonymous referees for their valuable comments.

References

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

[2] Amoui, M., Mirarab, S., Ansari, S., and Lucas, C. A genetic algorithm approach to design evolution using design pattern transformation. *International Journal of Information Technology and Intelligent Computing*, 1:235–245, 2007.

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

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

[5] Bodhuin, T., Di Penta, M., and Troiano, L. A search-based approach for dynamically re-packaging downloadable applications. In *Proceedings of the Conference of the Center for Advanced Studies on Collaborative Research (CASCON07)*, pages 27–41, 2007.

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

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

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

[9] Clarke, J., Dolado, J. J., Harman, M., Hierons, R. M., Jones, B., Lumkin, M., Mitchell, B., Mancoridis, S., Rees, K., Roper, M., and Shepperd, M. Reformulating software engineering as a search problem. *IEE Proceedings - Software*, 150(3):161–175, 2003.

[10] Frankel, D. S. *Model Driven Architecture - Applying MDA to Enterprise Computing.* Wiley Publishing, Inc., 2003.

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

[12] Gold, N., Harman, M., Li, Z., and Mahdavi, K. A search based approach to overlapping concept boundaries. In *Proceedings of the 22nd International Conference on Software Maintenance (ICSM 06)*, pages 310–319. IEEE, 2006.

[13] Harman, M., Mansouri, S. A., and Zhang, Y. Search based software engineering: a comprehensive review of trends, techniques and applications. Technical report TR-09-03, King's College, London, United Kingdom, 2009.

[14] Harman, M. and Tratt, L. Pareto optimal search based refactoring at the design level. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2007)*, pages 1106–1113. ACM, 2007.

[15] Jensen, A. C. and Cheng, B. H. C. On the use of genetic programming for automated refactoring and the introduction of design patterns. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2010)*, pages 1341–1348. ACM, 2010.

[16] Johnson, D. S., Aragon, C. R., McGeoch, L. A., and Schevon, C. Optimization by simulated annealing: An experimental evaluation; part I, graph partitioning. *Operations Research*, 37(6):865–892, 1989.

[17] Julstrom, B. A. Seeding the population: improved performance in a genetic algorithm for the rectilinear Steiner problem. In *Proceedings of the ACM Symposium on Applied Computing (SAC94)*, pages 222–226. ACM, 1994.

[18] Kirkpatrick, S., Gelatt, C., and Vecchi, M. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[19] Laarhoven van, P. J. M. and Aarts, E. *Simulated Annealing: Theory and Applications*. Kluwer, 1987.

[20] Metropolis, N., Rosenbluth, A. W., Rosenbluth, M. N., Teller, A. H., and Teller, E. Equation of state calculation by fast computing machines. *Journal of Chemical Physics*, 21:32–40, 1953.

[21] Michalewicz, Z. *Genetic Algorithms + Data Structures = Evolutionary Programs*. Springer-Verlag, 1992.

[22] Mitchell, M. *An Introduction to Genetic Algorithms*. MIT Press, 1996.

[23] O'Keeffe, M. and Ó Cinnéide, M. Towards automated design improvements through combinatorial optimization. In *Workshop on Directions in Software Engineering Environments (WoDiSEE2004), W2S Workshop - 26th International Conference on Software Engineering*, pages 75–82. IEEE, 2004.

[24] O'Keeffe, M. and Ó Cinnéide, M. Search-based software maintenance. In *Proceedings of Conference on Software Maintenance and Re-engineering (CSMR'06)*, pages 249–260. IEEE, 2006.

[25] O'Keeffe, M. and Ó Cinnéide, M. Search-based refactoring for software maintenance. *Journal of Systems and Software*, 81(4):502–516, 2008.

[26] Räihä, O. A survey on search-based software design. *Computer Science Review*, 4(4):203–249, 2010.

[27] Räihä, O. *Genetic Algorithms in Software Architecture Synthesis.* PhD thesis, University of Tampere, 2011.

[28] Räihä, O., Hadaytullah, Koskimies, K., and Mäkinen, E. Synthesizing architecture from requirements: A genetic approach,. In *Relating Software Requirements and Architectures*, pages 307–331. Springer, 2011.

[29] Räihä, O., Koskimies, K., and Mäkinen, E. Genetic synthesis of software architecture. In *Proceedings of the 7th International Conference on Simulated Evolution and Learning (SEAL08)*, pages 565–574. Springer, 2008.

[30] Räihä, O., Koskimies, K., and Mäkinen, E. Empirical study on the effect of crossover in genetic software architecture synthesis. In *Proceedings of the World Congress on Nature and Biologically Inspired Computing (NaBIC09)*, pages 619–625. IEEE, 2009.

[31] Räihä, O., Koskimies, K., and Mäkinen, E. Scenario-based genetic synthesis of software architecture. In *Proceedings of the 4th International Conference on Software Engineering Advances (ICSEA09)*, pages 437–445. IEEE, 2009.

[32] Räihä, O., Koskimies, K., and Mäkinen, E. Complementary crossover for genetic software architecture synthesis. In *Proceedings of the 10th International Conference on Intelligent Systems Design and Applications (ISDA10)*, pages 359–366. IEEE, 2010.

[33] Räihä, O., Koskimies, K., and Mäkinen, E. Generating software architecture spectrum with multi-objective genetic algorithms. In *Proceedings of the Third World Congress on Nature and Biologically Inspired Computing (NaBIC11)*, pages 29–36. IEEE, 2011.

[34] Räihä, O., Koskimies, K., Mäkinen, E., and Systä, T. Pattern-based genetic model refinements in MDA. *Nordic Journal of Computing*, 14(4):322–339, 2008.

[35] Räihä, O., Mäkinen, E., and Poranen, T. Using simulated annealing for producing software architectures. Technical Report D-2009-2, University of Tampere, Tampere, Finland, 2009.

[36] Ramsey, C. L. and Grefenstett, J. J. Case-based initialization of genetic algorithms. In *Proceedings of the 5th International Conference on Genetic Algorithms*, pages 84–91. Morgan Kaufmann Publishers, 1993.

[37] Seng, O., Bauyer, M., Biehl, M., and Pache, G. Search-based improvement of subsystem decomposition. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2005)*, pages 1045–1051. ACM, 2005.

[38] Seng, O., Stammel, J., and Burkhart, D. 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)*, pages 1909–1926. ACM, 2006.

[39] Shaw, M. and Garlan, D. *Software Architecture - Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[40] Simons, C. L. and Parmee, I. C. A cross-disciplinary technology transfer for search-based evolutionary computing: from engineering design to software engineering design. *Engineering Optimization*, 39(5):631–648, 2007.

[41] Simons, C. L. and Parmee, I. C. Single and multi-objective genetic operators in object-oriented conceptual software design. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2007)*, pages 1957–1958. ACM, 2007.

[42] Simons, C. L. and Parmee, I. C. Dynamic parameter control of interactive local search in UML software design. In *Proceedings of IEEE International Conference on Systems, Man and Cybernetics*, pages 3397–3404. IEEE Press, 2010.

[43] Simons, C. L. and Parmee, I. C. Elegant object-oriented software design via interactive, evolutionary computation. *IEEE Transactions on Systems, Man and Cybernetics, Part C Applications and Reviews*, 42(6):1797–1805, 2012.

[44] Simons, C. L., Parmee, I. C., and Gwynllyw, R. Interactive, evolutionary search in upstream object-oriented class design. *IEEE Transactions on Software Engineering*, 36(6):798816, 2010.

[45] Trikia, E., Colletteb, Y., and Siarry, P. A theoretical study on the behavior of simulated annealing leading to a new cooling schedule. *European Journal of Operational Research*, 166:77–92, 2005.

[46] Varadharajan, T. K. and Rajendran, C. A multi-objective simulated-annealing algorithm for scheduling in flowshops to minimize the makespan and total flowtime of jobs. *European Journal of Operational Research*, 167:772–795, 2005.

Received 20th August 2012