Proceedings of the Estonian Academy of Sciences,
2013, **62**, 1, 16–26
doi: 10.3176/proc.2013.1.03
Available online at www.eap.ee/proceedings

COMPUTER SCIENCE

Interleaving human and search-based software architecture design

Sriharsha Vathsavayi*, Hadaytullah, and Kai Koskimies

Department of Software Systems, Tampere University of Technology, Korkeakoulunkatu 1, P.O. Box 553, 33101 Tampere, Finland

Received 21 August 2011, revised 7 April 2012, accepted 25 October 2012, available online 20 February 2013

Abstract. An approach for semi-automated design of software architecture is proposed. The approach makes use of a search-based architecture synthesis technique exploiting genetic algorithms. An interactive process of software architecture design is proposed, where the automatic search-based generation of architectural fragments interleaves with the decisions of a human architect. To support such a process, tool mechanisms are proposed and implemented. The approach is studied using a sample system, whose architecture is designed following the interactive process model.

Key words: software architecture, genetic algorithms, semi-automated, tool support.

1. INTRODUCTION

Computer-aided design (CAD) has been a well-established technology in various fields of manufacturing for decades. However, traditionally CAD tools provide a fairly low level of intelligence and automation: they could be often characterized by customized drawing tools rather than tools helping the designer to come up with a solution for the design problem at hand. Once the design has been completed, the tool can often assist in the production of the actual artifact according to the design. In contrast, the step from requirements to design is less understood and, consequently, lacking tool support. This is particularly the situation in software engineering, where various computer-aided software engineering (CASE) tools and application generators provide management and automation facilities for different activities in the software engineering lifecycle, but hardly any support for producing a design solution from requirements. Interestingly, in other areas of technical design, fairly advanced tools exist for automated generation of complex design based on rules, patterns, and constraints (e.g. CityEngine [1]). Is software design inherently so difficult that it cannot be (even partially) automated, or is software development just a so immature discipline that the regularities of the design are not yet sufficiently well understood for building tools?

The problem of automated (or semi-automated) software architecture design has attracted increasing research interest in recent years. Basically, there are two main approaches: one can either (i) encode the human architectural reasoning and architectural design rules into a tool, or (ii) encode the properties of a good architecture into a tool and let the tool find an optimal (in terms of the goodness properties) candidate in the search space. In both cases, there are two options: the tool can assume human interaction and involvement during the construction of the architecture, or it can work in principle autonomously.

The former approach leads more naturally to a tool that is used interactively: the tool assumes a design process that is similar to the human architect's mental model, and can therefore be easily aligned with human involvement. The latter approach, in contrast, leads more naturally to an autonomous tool, since the search process employed by the tool can be (and probably is) completely incomprehensible for the architect. In that case, the only way the architect can interact with the tool is to change the input of the tool (for example, the definition of "goodness"), and restart the tool with the new input.

A good example of the first approach is the SEI Architecture Expert (ArchE), a semi-automated assistant for architecture design [2]. In this tool, the design knowledge is codified as a reasoning framework that is applied to direct the design process. Using the reasoning

* Corresponding author, sriharsha.vathsavayi@tut.fi

framework, the tool proposes different tactics to improve the design in certain aspects, and the architect can select tactics which the tool applies, resulting in a new architecture. The challenge in this approach is to make the reasoning framework intelligent enough to cope with all possible design situations that may arise. On the other hand, since this is an interactive approach, occasional failure of the tool is not really a problem if the tool can nevertheless handle most of the typical situations.

The latter approach requires no understanding of the design process, but on the other hand, it requires a definition of a "good" architecture. This is the challenge and potential weakness of the latter approach, as even human architects easily disagree on what is a "good" architecture. On the other hand, if a satisfactory definition can be found, this approach leads to a fairly straightforward automation, as the design is effectively transformed into a search problem.

Several authors have recently studied search-based software design (for a survey, see [3]). In general, work in this area is mostly concentrating on improving the existing design rather than producing an upfront design based on requirements analysis. For example, search-based solutions for the problem of refactoring a design have been proposed by many authors (e.g., [4]), as well as search-based clustering [5], subsystem decomposition [6], and class responsibility assignment [7]. Closer to our work, Amoui et al. [8] use a genetic approach to improve the reusability of software by applying architecture design patterns [9] to an existing architecture. The goal is to find the best sequence of pattern implementations.

However, as noted above, search-based approaches are less amenable to interactive working modes, since the search process itself is a black-box. On the other hand, an interactive mode would be highly desirable, as it is unlikely that a fully automated process could produce truly high-quality architectural designs. Rather, the tool should assist the architect in the course of the design, bringing up options, pointing out problems, and generating partial designs, very much in the style of ArchE. In this paper, we propose a possible approach to interactive search-based software architecture design tool. This proposal is based on our earlier work on genetic software architecture synthesis [10,11], applying genetic algorithms as the search technique. The main contributions of this paper are a characterization of an interactive search-based design process, a proposal for the tool mechanisms supporting such a process, and a small case study demonstrating the potential benefits of the approach.

We proceed as follows. In the next section we briefly summarize our basic approach for genetic software architecture synthesis and the tool supporting that approach. In Section 3 we discuss the interactive search-based design process, and in Section 4 we propose extensions to the basic tool to enable interactive genetic synthesis of software architecture. In Section 5 we present a case study in which we discuss the application of the proposed mechanisms. Finally, we conclude with some remarks on the possible future directions of this work in Section 6.

2. BACKGROUND

2.1. Genetic software architecture synthesis

Genetic algorithms are based on Darwinian evolution. In computer science, genetic algorithms are used to solve difficult problems with large search space [12]. The goal is to find an as good as possible solution in reasonable time. Each solution is represented as a *chromosome* and a collection of them is called a *population*. Furthermore, an evolution cycle is composed of *generations* of chromosomes. Each generation undergoes a reproduction cycle, where *mutations* and *crossovers* are applied to produce new chromosomes [13]. *Selection* occurs to select the best chromosomes of a generation to start off the next generation. A chromosome with a better *fitness* value is considered superior to the others. The fitness value is calculated using the *fitness function*.

Our genetic algorithm [11] introduces different design patterns and architectural styles (called here collectively patterns) to improve the system's quality. The goal is to find the right combination of the patterns that can improve the system qualities like modifiability, efficiency, and understandability. The patterns we have used are Message Dispatcher, Client-server, Mediator, Façade, Strategy, Adapter, and Template Method [9,14].

An evolution can be started by providing the target system's *null architecture* to the genetic algorithm. The null architecture is a rudimentary architecture containing only functional decomposition of the system without any consideration for the quality attributes (and without patterns). The null architecture is basically a UML class diagram of the system. In our approach, the null architecture can be systematically constructed on the basis of sequence diagrams that have been refined from the use cases of the system [10].

Moreover, attributes like frequency of use, execution time, variability, and call cost can be associated with each operation in the null architecture. These optional parameters are used in the fitness function to facilitate the measuring of modifiability, efficiency, and understandability. The values for the quality attributes are calculated using metrics adopted from Chidamber and Kemerer [15]. Additionally, a weight is also assigned to each quality attribute to emphasize or reduce its importance in the fitness function.

The genetic algorithm first converts the null architecture into a chromosome. Then it generates an initial population of the architectures by randomly inserting the patterns. During evolution, mutations and crossovers are applied to produce new improved architectures. A mutation actually introduces or removes a pattern to/from architecture. The mutation probabilities are

the probabilities given for the application of different patterns in the architecture, allowing the designer to favour or suppress the appearance of certain patterns. The frequency of application of a mutation or crossover depends on the probability assigned to it.

The elite of a population is used as the basis for the next population. This process keeps going until the last generation is reached. The best architecture of the last generation is the proposed solution.

2.2. Darwin tool

We have built Darwin tool [16,17] to facilitate the use of genetic software architecture synthesis. The tool has been implemented as an Eclipse [18] plug-in, providing necessary user interface controls for setting the various parameters and views to examine the results.

In order to create the null architecture and to see the final proposal, Darwin has been fully integrated with a UML-based CASE tool called UML2Tool [19]. Furthermore, using UML2Tool's use case diagram editor, a user can even start her work from scratch. She can start by identifying abstract use cases and then process them into more refined use cases. These refined use cases can then be grouped into subsystems to form the initial null architecture in the UML2Tool's class diagram editor.

Darwin's user interface also includes *Mutation* and *Weights* views to view/modify mutation/crossover probabilities and the quality attributes weights. These values can also be modified during an evolution and the changes are immediately in effect. Moreover, the parameter values (associated with operations) can also be modified before starting the evolution.

In *Settings view*, a user can set the total number of generations and population and elite sizes. A user can also choose the method for the calculation of overall fitness of a generation. There are two options for it, either averaging elite's fitness values or selecting the fitness of the best individual in the generation. Choosing an option depends on what kind of fitness (i.e., either fitness of the best individual or average fitness of the best individuals) of a generation the user wants to observe. The overall fitness of a generation is drawn as the fitness graph during an evolution.

An evolution can be started, paused, stopped, or resumed using the buttons provided on the user interface. Moreover, during an evolution Darwin shows the fitness values as a graph, called *Fitness Graph*, in real time. It is useful because the user can immediately see the effect of different input parameters, probabilities, and weights on the graph. Furthermore, Darwin's *Generation* view provides a list of all individuals in a selected generation, allowing the user to study a particular generation in more detail. Finally, a *Sequence Diagram* view is added to Darwin for giving sequence diagrams that are used to derive the null architecture. An open source case tool Papyrus [19] is used for realizing the sequence diagram view.

3. INTERACTIVE SEARCH-BASED SOFTWARE ARCHITECTURE DESIGN PROCESSES

The interactive search-based design approach involves a human architect in the genetic architecture synthesis. This kind of interactive, semi-automated architecture design process can be exploited in various development scenarios. Here we will more closely explore two general cases: the interactive, incremental development of software architecture from scratch, and the revision of an existing architecture due to changed requirements.

3.1. Incremental semi-automatic architecture generation

The idea of incremental, semi-automated architecture development is based on the assumption that architecting involves a great number of "routine" decisions that an automated search-based process can guess, but also some more intricate decisions that only a human architect can make. In the case of a problem having multiple design options, the architect uses the solution which she has used previously [20]. Thus, the benefits of automated and manual architecting can be combined by letting the architect interfere with the automated process.

The workflow of the incremental architecture generation is presented in Fig. 1. The process starts with gathering requirements as use cases. The use cases are then refined into sequence diagrams, which can be systematically transformed into the null architecture of the system [21]. An initial architecture proposal is produced by applying genetic architecture synthesis on the null architecture. More detailed information about generating architecture proposal using genetic architecture synthesis is presented in Hadaytullah et al. [16].

The software architect then observes that some parts of the architecture proposal are suboptimal, and she makes manual corrections into the architecture. Modifications can be made either by adding or removing patterns from the architecture. In addition to modifications, the architect can freeze patterns. As genetic architecture synthesis applies patterns randomly, it is possible that some patterns may not appear in the resulting architectural proposals. For example, in a distributed system a dispatcher is necessary, but genetic architecture synthesis may produce proposals without a dispatcher. To avoid this, the architect can introduce the dispatcher and freeze it, implying that the dispatcher will be retained in the architecture during the automated architecture generation.

The modified architecture is then subjected to genetic architecture synthesis. The genetic algorithm tries to improve the architecture without modifying the frozen patterns. The resulting architecture proposals contain frozen patterns plus new patterns inserted by the genetic algorithm. The process of the architect modifying the resulting architecture proposals and applying genetic architecture synthesis can be repeated until

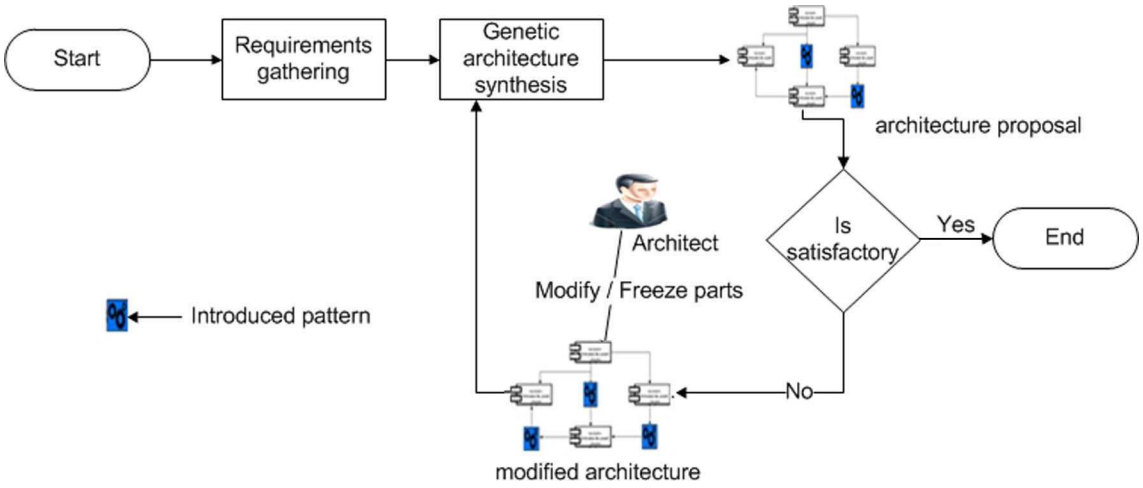

Fig. 1. Incremental architecture generation work flow.

a satisfied architecture is produced. We will study a concrete example of this type of process in Section 5.

3.2. Changing requirements

A system often undergoes some changes during the maintenance phase due to changed requirements. Here we are particularly interested in a situation where some quality requirements of the system change. For example, future evolution scenarios of the system may imply that a certain subsystem, or some parts in that subsystem, needs to be more modifiable. The genetic approach [10] allows specifying even individual operations that are possibly subject to changes in the future, so that the fitness function will take into account solutions supporting the modifiability of that operation. Another case could be that the performance of some subsystem turns out to be suboptimal, and the architecture needs to be revised to fix that. In both cases, the architect can then give the necessary input to the search-based tool, specifying the new quality requirements. In addition, the architect freezes those parts of the architecture that are not related to the changed requirements. Then the genetic architecture synthesis is applied on the architecture. In addition to frozen parts, the resulting architecture proposal contains solutions addressing new requirements. Finally, the architect can accept or reject the proposal.

4. TOOL SUPPORT

We have extended Darwin with three new mechanisms to support architect involvement in the genetic architecture synthesis. The first mechanism is the ability to take the *existing architecture as input*, introducing the capability of applying genetic architecture synthesis on an existing architecture. The existing architecture can be either an architecture designed by the architect, which contains predefined architectural decisions or an architecture proposal produced by genetic architecture synthesis. To realize this mechanism, an architecture view (as shown in Fig. 2) is added to Darwin. The architecture view is a class diagram editor realized using UML2Tool, allowing the manual insertion of patterns. It can be used to introduce an existing architecture as input to the genetic architecture synthesis. The genetic architecture synthesis uses the given input architecture as a seed and tries to improve the architecture by adding new solutions. In addition to the architecture, the requirements of the system under design also have to be provided. The sequence diagram view can be used to give requirements. The capability of giving the existing architecture as input, together with the integration with a UML class diagram editor, makes it possible for the architect to construct the architecture incrementally, adding or removing some patterns after genetic architecture synthesis, continuing the genetic architecture synthesis with the revised architecture, making again some manual editing, etc.

The second mechanism is *freezing patterns* in the architecture. It allows the architect to fix the patterns, which she wants to retain in the genetic architecture synthesis. The architecture view contains controls for freezing a pattern. The exact way of freezing a pattern depends on the type of the pattern. If a class introduces the pattern, the corresponding class is frozen (for example Strategy), but in the case of a dispatcher, the links between the dispatcher and the classes that communicate with the dispatcher are frozen.

For example, consider an architecture (shown in Fig. 3a) in which two classes A and B use the message dispatcher to communicate with each other. The architect wants to apply genetic architecture synthesis on the architecture, but wants the dispatcher communication between the two classes to persist. Then she can freeze the connection links between the dispatcher and classes A and B. The proposal that resulted from genetic

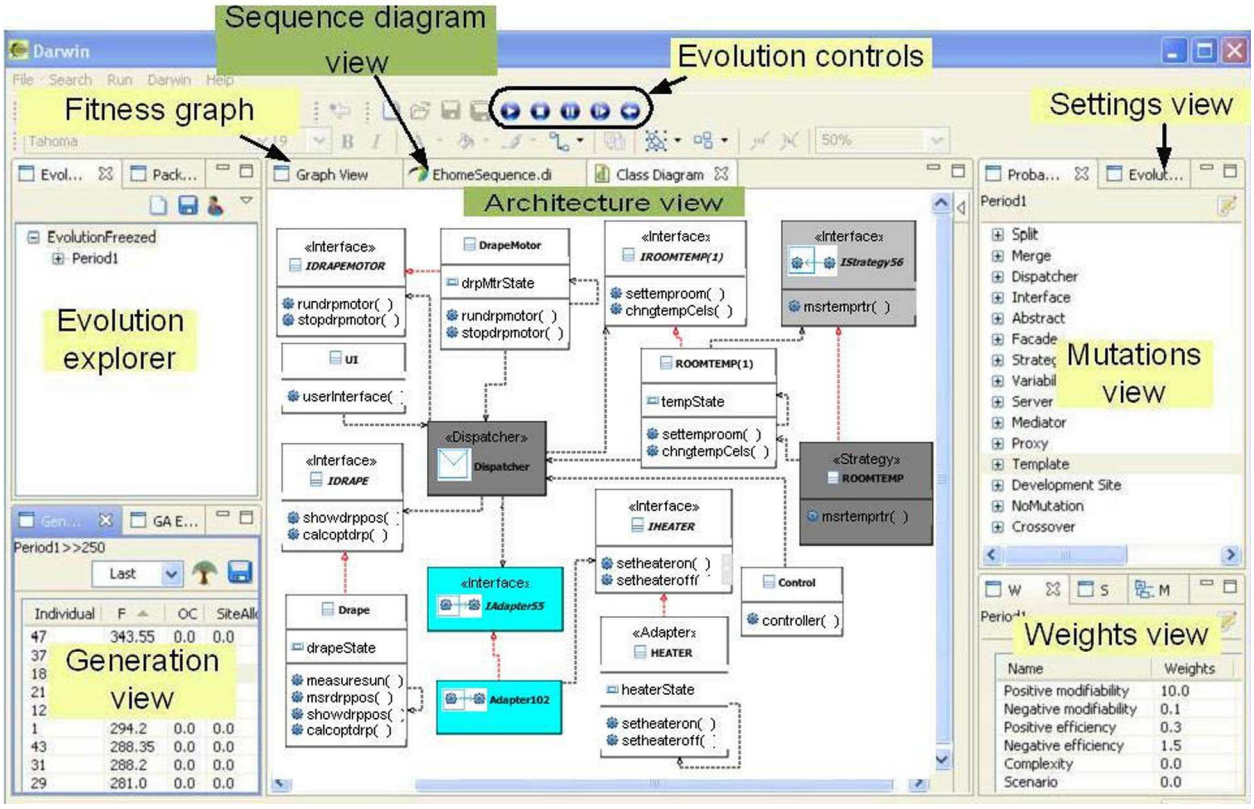

Fig. 2. Darwin user interface (architecture view with frozen patterns).

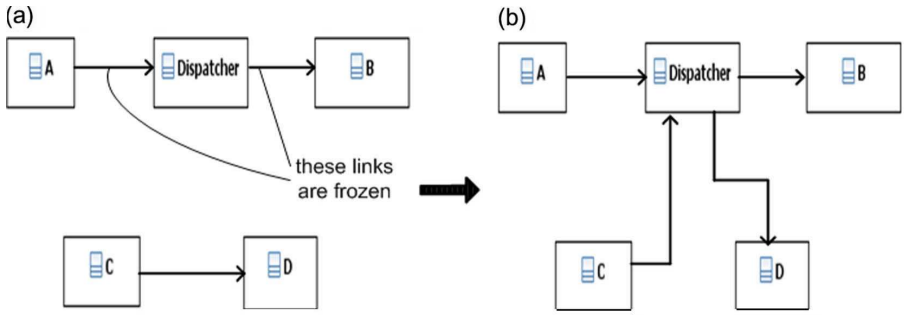

Fig. 3. Freezing dispatcher connection: (a) before genetic architecture synthesis, (b) after genetic architecture synthesis.

architecture synthesis retains the frozen connection links. In addition, the genetic algorithm may use the dispatcher for communication between classes C and D as shown in Fig. 3b, which were directly communicating previously. To distinguish between frozen patterns and other parts of the architecture, the frozen patterns are shown with dark grey in Fig. 2 (Strategy pattern ROOMTEMP and the dispatcher are frozen).

The third mechanism is to *withdraw patterns* from the architecture. The architect can avoid unwanted patterns completely by giving them probability 0. However, sometimes an architect may just want to express that a certain pattern suggested by the genetic algorithm in a certain context is not appropriate. For this purpose, the architecture view contains controls to mark certain patterns appearing in the architecture as unwanted. A context can be a subsystem, a class or an operation where a pattern can be applied. For example, an architecture emphasizing modifiability may contain many instances of Strategy and Template Method patterns, but this complicates the architecture and in many cases the proposed variability may be actually unnecessary. Then the architect can mark the unnecessary patterns as unwanted, and these patterns will not appear in those contexts in subsequent evolution.

5. CASE STUDY: SEMI-AUTOMATED ARCHITECTURE GENERATION

In this case study, our aim is to produce architecture using the incremental architecture generation process and examine the quality of the produced architecture. We used an embedded system called e-home as the target system. E-home is an imaginary control system for a computerized home. It has interfaces for controlling various home devices, like the coffee machine, drapes, audio, heating devices, etc. To simplify the study, we will here concentrate on two subsystems of e-home, drape and temperature control. Although this case study is fairly limited, it is sufficiently realistic to expose the major characteristics, advantages, and drawbacks of our approach. We will also present results of a small experiment which suggest that the incremental software architecture synthesis is competitive with good software engineering students.

In the first iteration an architecture proposal is generated from requirements. In our approach, this means the construction of the null architecture from the functional requirements expressed as use cases. Let us assume that the use cases for this system consist of changing room temperature and moving drapes. These use cases are then refined into sequence diagrams. A sequence diagram for temperature control is shown in Fig. 4. The sequence diagrams are transformed into a class diagram depicting the null architecture as shown in Fig. 5. This transformation is done by the Darwin tool. The operations can be given properties describing their expected characteristics (sensitiveness to changes, call frequency, time consumption). In this case we anticipate that for the operation settemproom the sensitiveness to changes is medium, call frequency is high, and time consumption is low. Such characterizations are not mandatory, but help the genetic algorithm to come up with good solutions. The next step is to set the genetic parameters and execute the genetic algorithm. The mutation and crossover probabilities found after some experimentation are applied to the algorithm. The modifiability sub-fitness is slightly weighed over other sub-fitnesses to produce modifiable architectures. The algorithm is executed for a population size of 100 individuals and 250 generations. As a result, the algorithm produces a set of architecture proposals. The architecture produced for the best individual is shown in Fig. 6. The classes related to the introduced patterns are darkened. Note that the classes that have been generated due to new inserted patterns are named according to the pattern involved.

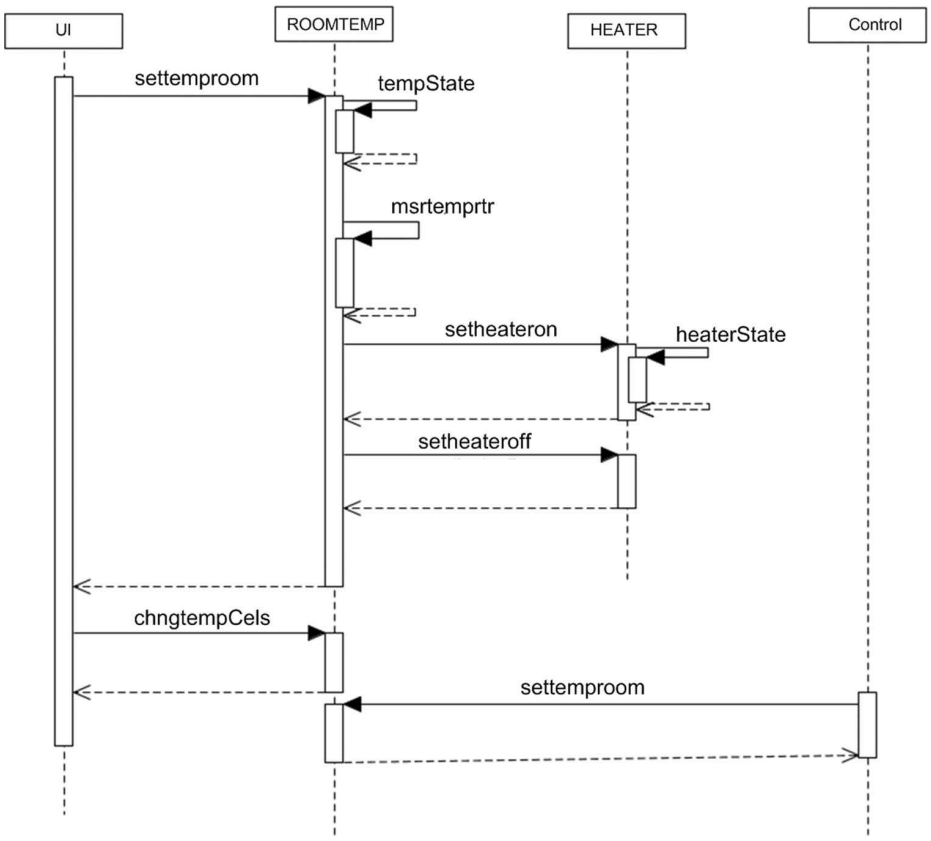

Fig. 4. Sequence diagram for temperature control of the e-home system.

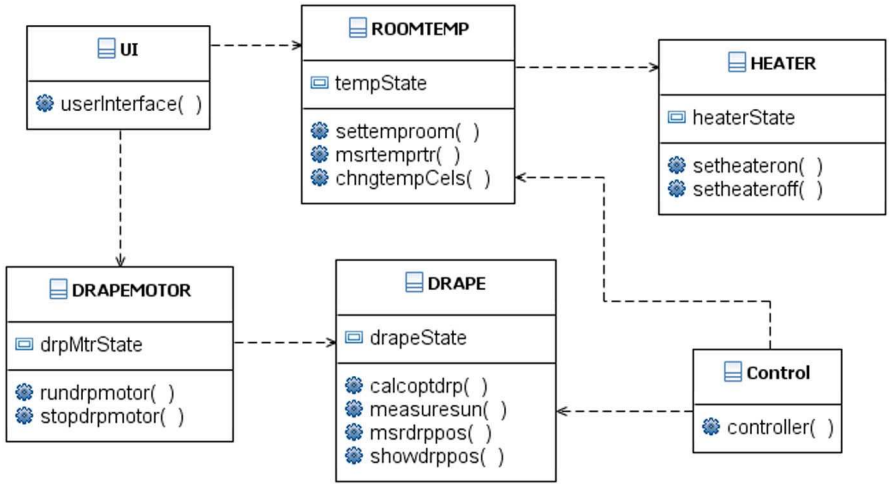

Fig. 5. Null architecture for the e-home system.

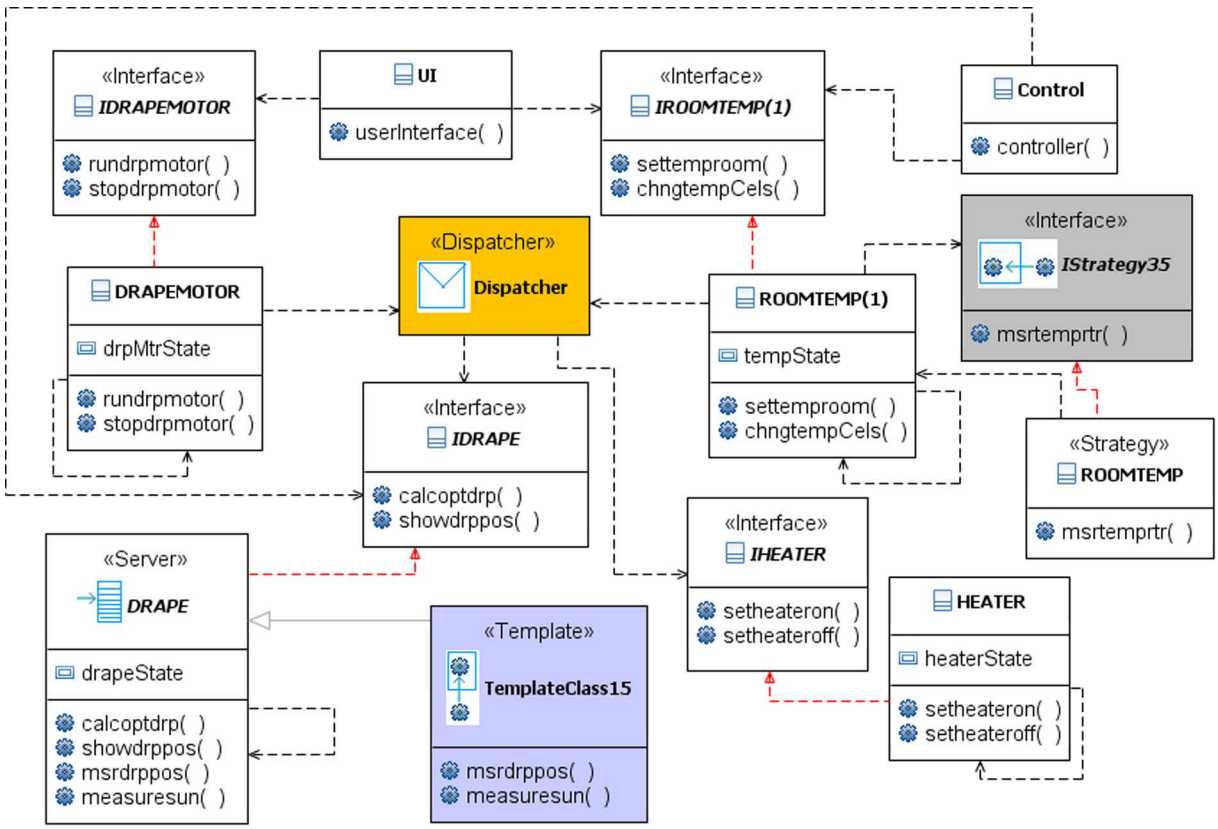

Fig. 6. Architecture proposal resulting after the first iteration.

As can be seen from Fig. 6, classes ROOMTEMP(1) and DRAPEMOTOR are communicating with their client dependencies HEATER and DRAPE through the message dispatcher. The architect decides that it is sensible to use the dispatcher also for other communication. The architecture is modified such that class Control also uses the message dispatcher for communicating with classes ROOMTEMP(1) and DRAPE. Further, the architect freezes the dispatcher connections between Control and its client dependencies, and Strategy class ROOMTEMP. The next step is to apply the genetic algorithm on the modified architecture. The genetic

algorithm is executed with the same parameters as used in the first iteration, although the genetic algorithm could be applied with varying mutation probabilities and fitness weights to obtain architecture proposals with varying quality. Generated architecture proposals retained the frozen solutions, but other solutions have also been inserted into the architecture by the genetic algorithm. The architecture of the best individual produced by the genetic algorithm is presented in Fig. 7. As can be seen, the Strategy pattern and dispatcher are retained, and new Adapter patterns are introduced into the architecture.

The resulting architecture can be further improved by repeating the second iteration on the architecture (with further modifications). As can be seen from the resulting proposal, class UI uses class ROOMTEMP directly. It is modified such that class UI uses class ROOMTEMP through message dispatcher, and then the Strategy class ROOMTEMP(1) and dispatcher connection between UI and ROOMTEMP are frozen. The genetic algorithm is again executed with the same parameters as used in the first iteration. The resulting architecture proposal of the best individual is shown in Fig. 8. As can be seen, class UI communicates now through the message dispatcher and new patterns are introduced into the architecture. The second iteration can be repeated until a satisfied architecture is produced. This kind of iteration process with manual involvement in the genetic architecture synthesis can produce better architectures. However, the architecture in Fig. 8 is already quite close to what a good human architect might design.

The resulting architecture proposal was evaluated against two different architecture proposals designed manually for the e-home system. These architecture proposals were the best proposals (judged on the basis of their evaluations as exam answers) designed by engineering students from a software architecture course at Tampere University of Technology (TUT). The students were given essentially the same information that is used as input for the genetic architecture synthesis. In addition, students were given a brief explanation of the purpose and functionality of the system. They were asked to design the architecture for the system, using only the same architecture styles (message dispatcher and client-server) and design patterns (Façade, Mediator, Strategy, Adapter, Template Method) that were available for genetic architecture synthesis. On average, it took students 40 min to produce a design. The time consumed for generating the resulting architecture proposal was about 5 min, which includes the time consumed for modifying the architecture and freezing the decisions (3 min), and executing all the three iterations (2 min).

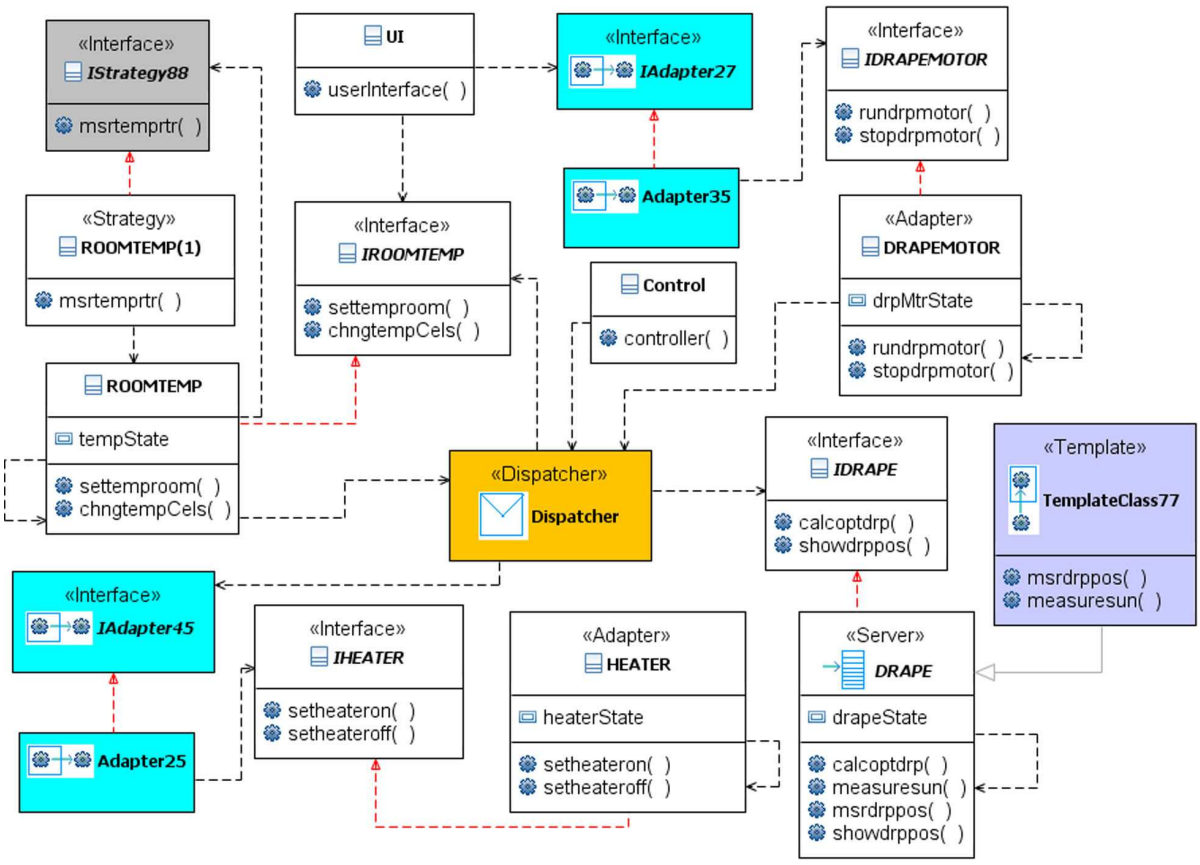

Fig. 7. Architecture proposal resulting after the second iteration.

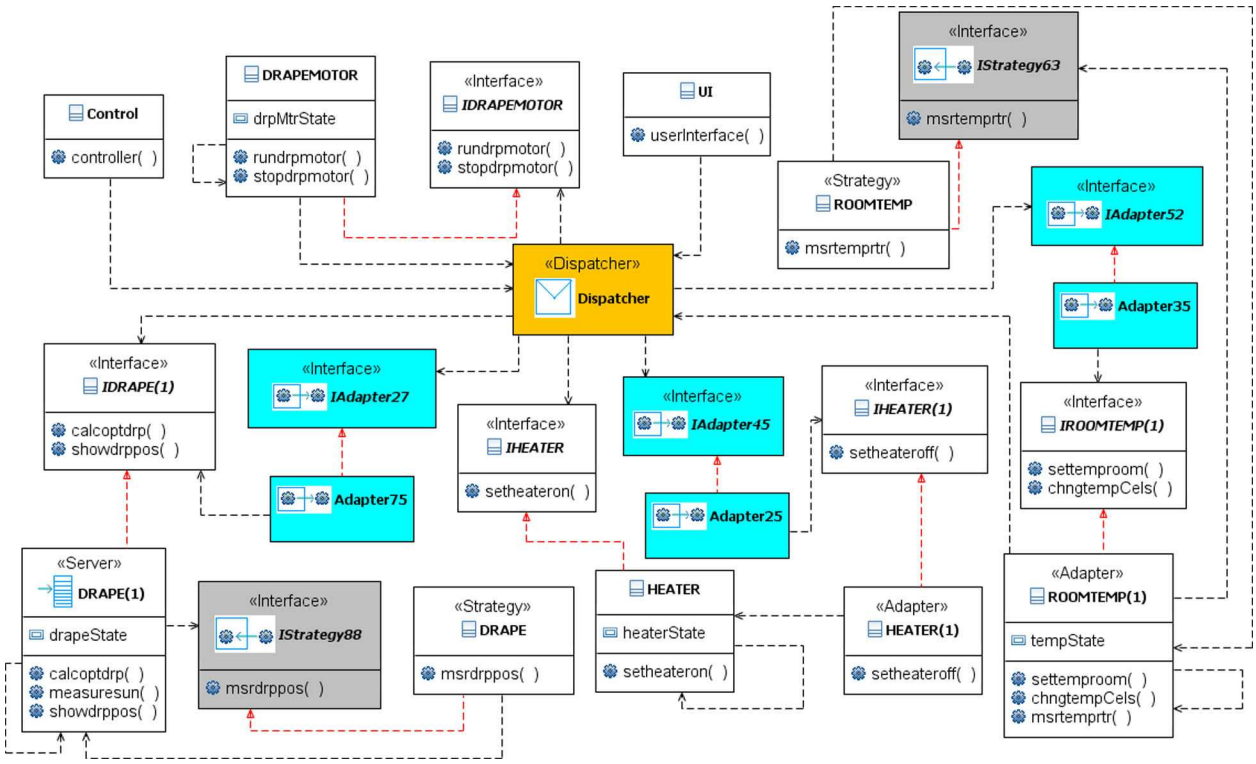

Fig. 8. Architecture proposal resulting after re-iterating the second iteration.

The student architecture proposals (S1, S2) together with the resulted architecture proposal (R) were presented for three software engineering experts in the TUT faculty. The experts were senior teachers or PhD students with great experience in software architectures. The solutions were edited in such a way that it was not possible for the experts to know which solution was synthesized. The experts were asked to order the solutions according to the overall quality of the architecture. Two experts ordered the solutions as (R, S1, S2) and the third expert ordered as (R, S2, S1).

We also performed some experiments to see how the inclusion of manual knowledge affects the fitness graph. Here we used an evolution with two periods. The first period generates architectures from requirements using the genetic architecture synthesis, whereas the second period uses the semi-automated architecture design approach. We utilized a population size of 100 individuals and 250 generations for both the periods. The curves are averages of ten test runs. Mutation probabilities and fitness weights are given after some experimentation. The calculated fitness value for a generation is the average of fitnesses of 10 best individuals of the generation. The average fitness graph generated for the tests is presented in Fig. 9. The fitness increases linearly after the first period, and jumps to a considerably higher level after the manual step, as can be expected. Note that an increase in the population size would increase the fitness, as big populations have better possibilities of having more good individuals. Similarly, an increase in the number of generations also increases the fitness, as individuals have more generations to evolve. However, the change in the fitness is not radical (for more information, see [11]).

Although this case study was small, it demonstrates the advantages and disadvantages of semi-automated search-based architecture design. The first iteration requires the specification of use cases as sequence diagrams (and, effectively, deciding the functional decomposition of the system) and some genetic parameters, but after that the architecture proposals are produced in seconds, and the architect can judge the sensibility of a given architecture instead of creating one herself. In this case, the required interaction input from the designer is limited to few manual corrections and freezing actions. Naturally the quality of the final result depends heavily on the quality of human interference. Still, the results of the small experiment suggest that, with very little human interaction, architectural solutions comparable to those produced by senior software engineering students can be achieved.

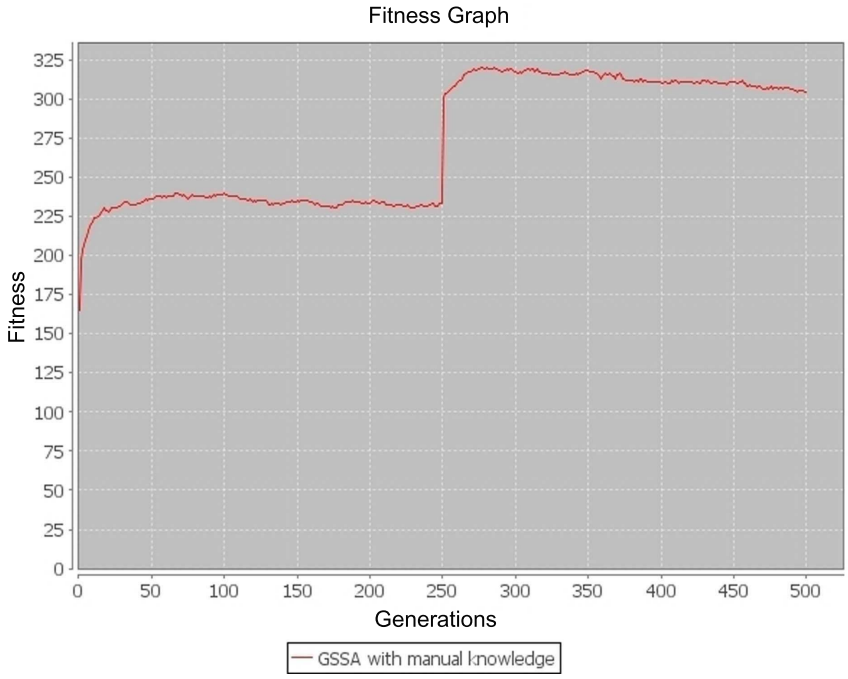

Fig. 9. Average fitness graph for ten test runs.

6. CONCLUDING REMARKS

We have demonstrated how interleaving search-based automated software architecture synthesis and manual intervention can combine the best parts of automated design and human design, speeding up the construction of an architecture that is close to the quality of a human-designed architecture. This approach can be supported by three extensions to a search-based (genetic) architecture synthesis technique: allowing the existing architectures to be used as input for the search process, allowing the freezing of certain parts of the input architecture, and allowing marking unwanted patterns from the architecture.

In our approach, the architect expresses the key functional requirements as use cases which are presented as sequence diagrams between the major units of the system, and the quality requirements as weights of certain quality attributes. In addition, the space of possible architectures is determined by a repository of patterns (mutations) available for the genetic algorithm. Since a pattern usually supports certain quality attributes at the cost of weakening others, the pattern repository is critical for the method. Ideally, the patterns in the repository should support various quality attributes, so that the genetic algorithm can pick those which fit the given requirements. An interesting topic for future research would be to construct a dynamically expanding pattern repository which is able to adopt new patterns when a designer makes a decision to apply a particular pattern that is not yet in the repository.

Obviously, more extensive case studies involving realistic systems are needed to validate the approach. However, even with the fairly modest case study in this paper, the results are encouraging. The approach retains the main benefits of the automated approach, while avoiding the problem of generating "theoretically" good solutions which, however, fail to satisfy the human expert. The fact that a proposal produced by the genetic algorithm is not optimal by human standards is not really a problem in this approach: the architect can decide which parts in the architecture are good, freeze them, make possibly other corrections, and resubmit the revised architecture to the automated process. A notable advantage of the automated part is that the produced proposal may suggest fresh, viable solutions that the architect, restricted by her previous experiences, would not even think of. This is particularly beneficial if the tool exploits a large, growing knowledge base of architectural patterns.

We see this work as a first step towards the practical usage of search-based approaches in software architecture design. Applying the proposed approach on a practical example and evaluating the architecture proposals against the architecture proposals designed by the professionals are our first priorities when considering future work. An intriguing possibility would be to combine our approach with the deterministic approach of ArchE [2]: our incremental generation technique could be integrated with the interactive process of ArchE, so that the architect could make the modifications to the architecture guided by the architecture knowledge framework of ArchE.

ACKNOWLEDGEMENT

This work is a part of the Darwin project, funded by the Academy of Finland.

REFERENCES

1. Parish, Y. I. H. and Muller, P. Procedural modeling of cities. *ACM SIGGRAPH 2001*, Los Angeles, August 2001, 301–308.
2. Diaz-Pace, A., Kim, H., Bass, L., Bianco, P., and Bachmann, F. Integrating quality-attribute reasoning frameworks in the ArchE design assistant. In *Proceedings of the 4th International Conference on Quality of Software-Architectures: Models and Architectures*. Springer LNCS, 2008, 171–188.
3. Räihä, O. A survey on search-based software design. *Computer Sci. Rev.*, 2010, **4**, 203–249.
4. Quaum, F. and Heckel, R. Local search-based refactoring as graph transformation. In *Proceedings of the 1st Symposium on Search-Based Software Engineering*. 2009, 43–46.
5. Harman, M. and Tratt, L. Pareto optimal search based refactoring at the design level. In *Proceedings of GECCO'07*. 2007, 1106–1113.
6. Seng, O., Bauyer, M., Biehl, M., and Pache, G. Search-based improvement of subsystem decomposition. In *Proceedings of GECCO'05*. 2005, 1045–1051.
7. Bowman, M., Brian, L. C., and Labiche, Y. *Solving the Class Responsibility Assignment Problem in Object-Oriented Analysis with Multi-Objective Genetic Algorithms*. Technical report, Carleton University, 2007.
8. Amoui, M., Mirarab, S., Ansari, S., and Lucas, C. A GA approach to design evolution using design pattern transformation. *Int. J. Inf. Technol. Intell. Comput.*, 2006, **1**, 235–245.
9. Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
10. Räihä, O., Hadaytullah, Koskimies, K., and Mäkinen, E. Synthesizing architecture from requirements: a genetic approach. In *Relating Software Requirements and Architecture, Ch 18* (Avgeriou, P., Grundy, J., Hall, J. G., Lago, P., and Mistrik, I., eds). Springer-Verlag, 2011, 307–331.
11. Räihä, O., Koskimies, K., and Mäkinen, E. Genetic synthesis of software architecture. In *Proceedings of SEAL'08*. Springer LNCS, 2008, 565–574.
12. Michalewicz, Z. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer, 1992.
13. Mitchell, M. *An Introduction to Genetic Algorithms*. The MIT Press, 1996.
14. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. *Pattern-Oriented Software Architecture: A System of Patterns, vol. 1*. John Wiley and Sons, 1996.
15. Chidamber, S. R. and Kemerer, C. F. A metrics suite for object oriented design. *IEEE Trans. Software Eng.*, 1994, **20**(6), 476–492.
16. Hadaytullah, Vathsavayi, S., Räihä, O., and Koskimies, K. Tool support for software architecture design with genetic algorithms. In *Proceedings of ICSEA'10*. IEEE CS Press, August 2010, 359–366.
17. Darwin research project WWW site. http://practise.cs.tut.fi/project.php?project=darwin (last viewed February 2011).
18. Eclipse WWW site. http://www.eclipse.org (last viewed February 2011).
19. Eclipse's Model Development Tools WWW site. http://www.eclipse.org/modeling/mdt (last viewed February 2011).
20. Van Heesch, U. and Avgeriou, P. Mature architecting – a survey about the reasoning process of professional architects. In *Proceedings of the 9th IEEE/IFIP Working Conference on Software Architecture (WICSA)*. IEEE Computer Society, 2011, 260–269.
21. Hadaytullah, Räihä, O., and Koskimies, K. Genetic approach to software architecture synthesis with work allocation scheme. In *Proceedings of APSEC'10*. IEEE CS Press, 2010, 70–79.

Tarkvara arhitektuuri disain projekteerija otsuste ja masinotsimise vaheldumise kaudu

Sriharsha Vathsavayi, Hadaytullah ja Kai Koskimies

On esitatud poolautomaatne tarkvara arhitektuuri projekteerimise meetod, mille aluseks on geneetiliste algoritmide tehnika. On kirjeldatud interaktiivset tarkvara arhitektuuri projekteerimise protsessi, kus tarkvara struktuuri fragmentide otsimispõhine süntees vaheldub inimesest projekteerija otsustustega. On välja pakutud ja programmidena realiseeritud ka tööriistad, mis seda protsessi toetavad. Meetodi kasutatavuse uurimiseks on artiklis skitseeritud nn tark kodu (e-kodu) ja kirjeldatud selle juhtimissüsteemi arhitektuuri interaktiivset projekteerimist.