Evolutionary Computing and Autonomic Computing: Shared Problems, Shared Solutions?

A.E. Eiben

Vrije Universiteit Amsterdam

Abstract. The purpose of this paper is to present evolutionary computing (EC) and to identify a number of issues where EC and autonomic computing, a.k.a. self-*, are mutually relevant for each other. We show that Evolutionary Algorithms (EA) form a metaheuristic that can be used to tackle the problem of self-optimisation in autonomic systems and suggest that an evolutionary approach can also help solving other challenges in autonomic computing. Meanwhile, an evolving system can be seen as a special case of an autonomic system. From this perspective, the quest for parameterless EAs can be positioned in a broader context and it can be expected that some solutions invented within autonomic computing can be transferred to EC.

1 Introduction

This position paper is aiming at linking evolutionary computing and autonomic computing. Autonomic computing is assumed to be known, therefore it is not reviewed here. Evolutionary computing is discussed emphasizing those facets that are most relevant to make the main points about the mutual relevance of the two areas.

We argue that the evolutionary mechanism is inherently capable of optimising a collection of entities. This capability comes forth from the interplay of three basic actions: reproduction, variation, and selection. Whenever the entities in question reproduce they create a surplus, variation during reproduction amounts to innovation of novel entities[1], and finally selection takes care of promoting the right variants by discarding the poor ones. This process has led to the Homo Sapiens on Earth and to numerous superior solutions of engineering and design problems in evolutionary computing [6]. Technically, an evolutionary process can be perceived as a generate-and-test search algorithm regulated by a number of parameters and it has two very interesting properties from a self-* perspective. First, evolution is able to evolve itself, that is, to tune its own parameters on-the-fly. Second, it is able to adapt itself to changing circumstances, that is, to track optimal solutions after the objective function is changed.

We also argue that evolutionary computing is one of the key technologies that can help meeting some of the grand challenges of autonomic computing. EC is widely applicable, it requires almost no assumptions about the problem to be

[1] That is, in our case reproduction is not simply cloning.

solved, an evolutionary solver can be usually developed with limited efforts and it produces good quality solutions at acceptable computational costs under a wide range of circumstances. To illustrate our point we describe an evolutionary approach to self-optimisation in a distributed system. Our example is a problem concerning web services offered to a large number of users via a (possibly large) number of servers. The quality of service is the key measure to be optimised and re-optimised if the circumstances change, for instance, if the behaviour of the users changes over time. The key to our approach is to have each user session regulated by a number of parameters and allow variations in these parameters. Adding selection based on the quality of service belonging to given parameter values introduces survival of the fittest and makes the system evolutionary.

The paper is organised as follows. In Section 2 a general introduction to EC is given. Section 3 provides more details on a specific type of EAs, evolution strategies, and illustrates how self-adaptation works in EC. Thereafter, in Section 4, an example application is described and an evolutionary approach is presented to realise self-optimisation in the system. Besides specifying a concrete EA to solve this problem, we also consider general properties of an evolutionary approach in such a context. The paper is concluded by Section 5, where we discuss how and why developments in these two fields can be expected to help solving great challenges in the other field.

2 Evolutionary Computing in a nutshell

Evolutionary Computing encompasses a variety of so-called evolutionary algorithms [2, 5, 6] that all share a common underlying idea: given a population of individuals, the environmental pressure causes natural selection (survival of the fittest), which causes a rise in the fitness of the population over time. The main principle behind evolution, be it natural or computer simulated, can be summarised as follows. If a collection of objects satisfies that

- they are able to reproduce,
- their offspring inherits their features,
- these features can undergo small random, undirected variations
- these features effect their reproduction probabilities,

then the features of these objects will change over time in such a way that they will fit their environment better and better.

In a formal setting, the environment is represented by a given quality function to be maximised.[2] The population is created by randomly generating a set of candidate solutions, i.e., elements of the function's domain, and the quality function is used as an abstract fitness measure – the higher the better. Based on this fitness, some of the better candidate solutions are chosen to seed the next generation by applying recombination and/or mutation to them. Recombination

[2] Handling minimisation problems only requires a trivial mathematical transformation.

is an operator applied to two or more selected candidates (the so-called parents) and results one or more new candidates (the children). Mutation is applied to one candidate and results in one new candidate. Executing recombination and mutation leads to a set of new candidates (the offspring) that compete – based on their fitness (and possibly age)– with the old ones for a place in the next generation. This process can be iterated until a candidate with sufficient quality (a solution) is found or a previously set computational limit is reached.

In this process there are two fundamental forces that form the basis of evolutionary systems:

– Variation operators (recombination and mutation) create the necessary diversity and thereby facilitate novelty.
– Selection acts as a force pushing quality. As opposed to variation, selection reduces diversity.

Based on the biological analogy one often distinguishes phenotypes and genotypes of candidate solutions. The phenotype of a candidate is its "outside", the way it looks and/or acts. The genotype denotes the code, the "digital DNA", that encodes or represents this phenotype. It is an important to note that variation and selection act in different spaces.

– Variation operators act on genotypes. Mutation and recombination never take place on phenotypical level, for instance, changing a leg into a wing. Rather, variation effects on the level of genes that determine the phenotype.
– Selection acts on phenotypes. A gene is never evaluated directly, it has to be expressed as a physical feature or behaviour and it is this feature or behaviour that gets evaluated by the environment and influences the survival and reproduction capabilities.

The combined application of variation and selection generally leads to improving fitness values in consecutive populations. It is easy (although somewhat misleading) to see such a process as if the evolution is optimising, or at least "approximising", by approaching optimal values closer and closer over its course. Alternatively, evolution it is often seen as a process of adaptation. From this perspective, the fitness is not seen as an objective function to be optimised, but as an expression of environmental requirements. Matching these requirements more closely implies an increased viability, reflected in a higher number of offspring. The evolutionary process makes the population increasingly better at being adapted to the environment.

It is important to note that many components of such an evolutionary process are stochastic. During selection fitter individuals have a higher chance to be selected than less fit ones, but typically even the weak individuals have a chance to become a parent or to survive. For recombination of individuals the choice of which pieces will be recombined is random. Similarly for mutation, the pieces that will be mutated within a candidate solution, and the new pieces replacing them, are chosen randomly. The general scheme of an evolutionary algorithm can is given in Fig. 1 in a pseudocode fashion.

```
BEGIN
  INITIALISE population with random candidate solutions;
  EVALUATE each candidate;
  REPEAT UNTIL ( TERMINATION CONDITION is satisfied ) DO
    1 SELECT parents;
    2 RECOMBINE pairs of parents;
    3 MUTATE the resulting offspring;
    4 EVALUATE new candidates;
    5 SELECT individuals for the next generation;
  OD
END
```

Fig. 1. The general scheme of an evolutionary algorithm in pseudocode

It is easy to see that EAs fall in the category of generate-and-test algorithms. The evaluation (fitness) function represents a heuristic estimation of solution quality, and the search process is driven by the variation and the selection operators. Evolutionary algorithms possess a number of features that can help to position them within in the family of generate-and-test methods:

– EAs are population based, i.e., they process a whole collection of candidate solutions simultaneously.
– EAs mostly use recombination to mix information of more candidate solutions into a new one.
– EAs are stochastic.

Example: The Travelling Salesman Problem
In the Travelling Salesman Problem (TSP) the task is to find a tour (Hamiltonian circle) through n given locations with minimal length. An evolutionary approach considers tours as phenotypes that are evaluated by their length, the shorter a tour the higher its fitness. An appropriate genotype can be for instance a permutation of n location IDs with the obvious genotype-phenotype mapping. The essence of designing an EA for the TSP is to specify appropriate variation and selection operators (followed by defining the initialisation procedure, termination condition, etc). To keep things really simple one could decide to use mutation as the only variation operator and chose to mutate a permutation by swapping the values on two randomly chosen positions. As for selection –remember, it is independent from what genotypes we use– one can use fitness proportional random drawing. Whenever a good individual needs to be selected from a population of m, any candidate c_i is selected by a probability $p_i = fitness(c_i)/\sum_{i=1}^{m} fitness(c_i)$

As mentioned before there are different EA variants. The most important types, or "dialects", are (in alphabetical order) evolution strategies, evolutionary programming, genetic algorithms, and genetic programming [2, 5, 6]. These dialects differ only in technical details. For instance, the representation of a candidate solution is often used to characterise different streams. Typically, the candidates are represented by (i.e., the data structure encoding a solution has the form of) strings over a finite alphabet in genetic algorithms (GA), real-valued vectors in evolution strategies (ES), finite state machines in classical evolutionary programming (EP), and trees in genetic programming (GP). These differences have a mainly historical origin. Technically, a given representation might be preferable over others if it matches the given problem better; that is, it makes the encoding of candidate solutions easier or more natural. It is important to note that the recombination and mutation operators working on candidates must match the given representation. Thus, for instance, in GP the recombination operator works on trees, while in GAs it operates on strings. As opposed to variation operators, selection takes only the fitness information into account; hence it works independently from the actual representation.

Technically, an EA has numerous parameters. The precise list of parameters and the way they are set are depending on the type of EA at hand. However, in all cases one has to arrange the population, selection, and variation. The following list illustrates some common parameters.

- Population size: the number of candidate solutions (typically kept constant during a run). A small population allows faster progress but increases the risk of getting stuck in a local optimum because it can only maintain fewer alternatives, hence less diversity.
- Selection pressure: the extent of bias preferring the good candidates over weak ones. High selection pressure causes faster progress but increases the risk of getting stuck in a local optimum by being too greedy. Zero selection pressure degrades evolutionary search into random walk.
- Mutation magnitude[3]: the parameter regulating the influence of mutation, e.g., how often, how big, etc. Using more/larger mutation speeds up the search but can prevent fine tuning on the optimum.

In the early days of EC it has been claimed that EAs have robust parameters, i.e., that EAs are to a large extent insensitive to the exact parameter values. Later on this claim has been revised and the contemporary view acknowledges that using the right parameter values can make a big difference in algorithm performance. The effects of setting the parameters of EAs has been the subject of extensive research by the EA community and recently there is much attention paid to self-calibrating EAs. The ultimate goal is to have a parameter-free algorithm that can calibrate itself to any given problem while solving that problem. For an extensive treatment of this issue [4] and [6, Chapter 8] are recommended, [1] provides an experimental comparison between EAs using different levels of self-calibration.

[3] *Mutation magnitude* is not an established technical term in EC. It is used here as an umbrella term covering the commonly used ones, like mutation rate, mutation step size, etc.

3 Evolution strategies and self-adaptation

In this section we outline evolution strategies. Hereby we present a member of the evolutionary algorithm family in details and illustrate a very useful feature in evolutionary computing: self-adaptation. In evolutionary computing self-adaptivity means that some parameters of the EA are varied during a run in a specific manner: the parameters are included in the chromosomes and co-evolve with the solutions. This feature is inherent for evolution strategies, i.e., from the earliest versions ESs are self-adaptive, and during the last couple of years other EAs are adopting self-adaptivity.

Evolution strategies are typically used for continuous parameter optimization problems, i.e., functions of the type $f : \mathbb{R}^n \to \mathbb{R}$, using real-valued vectors as candidate solutions. Parent selection is done by drawing λ individuals with a uniform distribution from the population of μ, where $\lambda > \mu$ (very often μ/λ is about 1/7). After creating λ offspring and calculating their fitness the best μ of them is chosen *deterministically* either from the offspring only, called (μ, λ) selection, or from the union of parents and offspring, called $(\mu + \lambda)$ selection. Recombination in ES is rather straightforward, two parent vectors $\bar{u}$ and $\bar{v}$ create one child $\bar{w}$, where

$$w_i = \begin{cases} (u_i + v_i)/2 & \text{in case of intermediary recombination} \\ u_i \text{ or } v_i \text{ chosen randomly} & \text{in case of discrete recombination} \end{cases} \tag{1}$$

The mutation operator is based on a Gaussian distribution requiring two parameters: the mean, which is always set at zero, and the standard deviation σ, which is interpreted as the mutation step size. Mutations then are realised by replacing components of the vector $\bar{x}$ by

$$x'_i = x_i + \sigma \cdot N(0,1), \tag{2}$$

where $N(0,1)$ denotes a random number drawn from a Gaussian distribution with zero mean and standard deviation 1. By using a Gaussian distribution here, small mutations are more likely then large ones. The particular feature of mutation in ES is that the step-sizes are also included in the chromosomes. In the simplest case one σ that acts on each x_i, in the most general case a different one for each position $i \in \{1, \ldots, n\}$. A typical candidate is then $\langle x_1, \ldots, x_n, \sigma_1, \ldots, \sigma_n \rangle$ and mutations are realised by replacing individual $\langle x_1, \ldots, x_n, \sigma_1, \ldots, \sigma_n \rangle$ by $\langle x'_1, \ldots, x'_n, \sigma'_1, \ldots, \sigma'_n \rangle$, where

$$\sigma' = \sigma \cdot e^{\tau \cdot N(0,1)} \tag{3}$$

$$x'_i = x_i + \sigma' \cdot N_i(0,1) \tag{4}$$

and τ is a parameter of the method.

By this mechanism the mutation step sizes are not set by the user, they (the $\bar{\sigma}$ part) are co-evolving with the solutions (the $\bar{x}$ part). To this feature it is essential to modify the σ's first and mutate the x's with the new σ values.

The rationale behind it is that an individual $\langle \bar{x}, \bar{\sigma} \rangle$ is evaluated twice. Primarily, it is evaluated directly for its viability during survivor selection based on $f(\bar{x})$. Secondarily, it is evaluated for its ability to create good offspring. This happens indirectly: a given $\bar{\sigma}$ evaluates favourably if the offspring generated by using it turns viable (in the first sense). Thus, an individual $\langle \bar{x}, \bar{\sigma} \rangle$ represents a good $\bar{x}$ that survived selection and a good $\bar{\sigma}$ that proved successful in generating this good $\bar{x}$.

Observe that using self-adaptive mutation step sizes has two advantages: 1) the user does not have to bother about it, the EA does it itself, 2) parameter values are changing during the run. In general, modifying algorithm parameters during a run is motivated by the fact that the search process has different phases and a fixed parameter value might not be appropriate for each phase. For instance, in the beginning of the search exploration takes place, where the population is wide spread, locating promising areas in the search space. In this phase large leaps are appropriate. Later on the search becomes more focused, exploiting information gained by exploration. During this phase the population is concentrated around peaks on the fitness landscape and small variations are desirable.

There are various techniques in evolutionary computing to adjust algorithm parameters (also called strategy parameters) on-the-fly [6, Chapter 8]. Self-adaptivity is one such technique, where the parameters are changed by the algorithm itself with only minimal influence from the user. In case of self-adaptation of parameters the algorithm is performing two tasks simultaneously: It is solving a given problem and it is calibrating (and repeatedly re-calibrating) itself for solving that problem. While in theory this implies a computational overhead that could lead to reduced performance, the practice of ES –and many other EAs adopting self-adaptive features– show the opposite effect.

A convincing evidence for the power of self-adaptation is provided in the context of changing fitness landscapes. In this case the objective function is changing and the evolutionary process is aiming at a moving target. When the objective function changes, the present population needs to be re-evaluated, and quite naturally the given individuals may have a low fitness, since they have been adapted to the old objective function. Often the mutation step sizes will prove ill-adapted: they are too low for the new exploration phase required. The experiment presented in [8] illustrates how self-adaptation is able to reset the step sizes after each change in the objective function without any user intervention. Fig. 2 shows that the location of the optimum is changed after every 200 generations (*x-axes*) with a clear effect on the average best objective function values (*y-axis, left*) in the given population. Self-adaptation is adjusting the step sizes (*y-axes, right)* with a small delay to larger values appropriate for exploring the new fitness landscape, thereafter the values of σ start decreasing again once the population is closing in on the new optimum.

Over the last decades much experience has been gained over self-adaptation in ES. The accumulated knowledge has identified necessary conditions for self-adaptation:

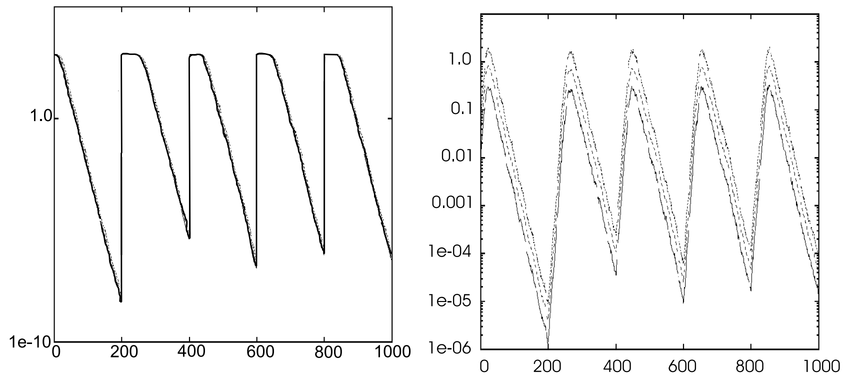

Fig. 2. Moving optimum ES experiment on the 30 dimensional sphere function. See text for explanation.

- $\mu > 1$ so that different strategies are present
- Generation of an offspring surplus: $\lambda > \mu$
- A not too strong selective pressure (heuristic: $\lambda/\mu = 7$, e.g., (15,100))
- (μ, λ)-selection (to guarantee extinction of misadapted individuals)
- Recombination also on strategy parameters (especially intermediate recombination)

4 The web service example

In this section we show how an evolutionary approach can be used to build autonomic computing systems, or at least how EC can be utilized to solve some of the key problems raised within autonomic computing, in particular that of real-time self-optimisation. To this end, we introduce an example problem that serves to illuminate the matter. Note, that the point is not to solve the example problem, but to show how the generic "evolutionary trick" can be applied to solve challenges in autonomic computing.

4.1 The web service example: the optimisation problem

Let us assume some web service to a large number of visitors. Without loss of generality we can also assume that the service is provided by a number of service units, e.g., M web-servers offering the same service through the same URL such that the visitors do not even notice whether their session is conducted by unit A or unit B. The main task here is to maximise the service level g. We can assume that the service level g is defined as some combination of the time spent with obtaining the service (shorter session, higher service level) and the degree to which a request could be satisfied (higher degree, higher service level). Furthermore, we postulate that each session conducted with a visitor of the given web site is regulated by a parameterized procedure, using a parameter vector $\bar{p}$. This parameter vector can consist of values encoding, for instance, colour, arrangement, etc. of the web pages used, the (type of) messages presented to the visitor, applied pricing strategy, the sub-page hierarchy, subroutines used in a session, ordering of databases consulted in a session, etc. Formally, the values within $\bar{p}$ can be Booleans, integers, reals, or even a mixture of them and we have a parameter optimization problem, since we want to use those $\bar{p}$ vectors that maximise g, that is, we want to conduct sessions that maximise service level. In the following we will illustrate how this can be done in self-* style using an evolutionary approach.

4.2 The web service example: individuals, population, fitness

The basis of this evolutionary approach is to consider a given $\bar{p}$ as an individual whose fitness is $g(\bar{p})$ and to set up a system where a population of individuals undergoes variation and selection.

To introduce a population we must allow that different values of $\bar{p}$ are used simultaneously, i.e., visitor 1 can be serviced by using a procedure belonging to $\bar{p}_1$, while the interaction with visitor 2 can take place by using $\bar{p_2}$. After finishing a session the quality of the parameter $\bar{p}$ used in the session can be determined by calculating the corresponding service level $g(\bar{p})$. It can be argued that calculating $g(\bar{p})$ should be based on more than one sessions with $\bar{p}$. Technically, this is a simple extension having no influence for the present discussion. Having specified parameter vector $\bar{p}$ and the utility function g we have the most fundamental requirement for an evolutionary process: an evaluation function or fitness function applicable to a population of individuals. Then, at all times we can maintain a set of N parameter values (and call N the population size). Invocation of parameter values, that is assigning some $\bar{p}$ from the given population to a new session, must be also regulated in some way, but these details are not important for the present discussion either. What is important is the distinction between the pool of service units (consisting of M elements) and the pool of N $\bar{p}$ values, being the population to be evolved. The key to real-time self-optimisation of the system consisting of the service units is to evolve this population of parameter values. Technically this requires variation and selection operators.

4.3 The web service example: variation

Variation can be handled in a rather straightforward way: using common mutation operators from EC we can specify small random perturbations to a given value $\bar{p}$, yielding $\bar{p}'$. Alternatively, if there are no appropriate off-the-shelf mutation operators, one can design application specific mutation – this is mostly not too difficult. For a well-defined procedure we also have to define when to apply variation. A simple heuristic for this is to create a child $\bar{p}'$ to $\bar{p}$ as soon as $\bar{p}$ gets evaluated, i.e., $g(\bar{p})$ is calculated. Of course, there is no need to restrict ourselves to mutation only, and also recombination can be used to create new individuals. Here again common recombination operators from EC can be applied to two parent vectors $\bar{p}_1$ and $\bar{p}_2$, or designed for the specific needs. (This might be more difficult than inventing mutation operators.) Depending on the operator the result can be one or two new vectors. For specifying when recombination is applied we can use a heuristic similar to that concerning mutation.

4.4 The web service example: selection

Selection is a bit more complicated than variation in our case. To begin with parent selection, we can keep it very simple and unbiased, that is, not related to the fitness of the individuals (utility of the parameter vectors). This can be achieved by the heuristic mentioned above and mutate every individual after it gets evaluated, regardless to its fitness. As for recombination we can apply this heuristic too but we also need to specify how to select a second parent $\bar{p}_2$, for a given $\bar{p}_1$. Here we can use a random choice based on the uniform distribution, giving every other individual in the population an equal chance.

Concerning survivor selection we will consider two options: local competition and global competition. The basic idea behind local competition is that each newborn child competes with its parent(s) directly. In this case each new $\bar{p}'$ must be used in a session as soon as possible after its "birth" to calculate its utility, that is, its fitness value. This might imply a requirement for the invocation procedure, but we do not discuss this aspect here. What is important for meaningful selection is that a parent $\bar{p}$ is kept in the population (and probably used again) until its offspring $\bar{p}'$ gets evaluated. When $g(\bar{p})$ and $g(\bar{p}')$ are both known then we select either of them based on g and delete the other one. This selection can be deterministic (keep the winner) or probabilistic giving the winner a higher chance to survive. Note that some form of additional population management might be required here if we allow that a waiting parent (a given $\bar{p}$ whose offspring $\bar{p}'$ is not evaluated yet) can be invoked and used in a session. This extra bookkeeping is needed to ensure that no individual is being deleted too early, yet minimising the period during which parents and offspring under evaluation co-exist.[4]

Global competition is based on the idea to let parents and children co-exist for a while and consequently to let populations grow. During such a predefined period of growth, called epoch, no individual is deleted. The length of an epoch can be specified as a given number of fitness evaluations (parameter vector invocations), successful sessions, wall-clock time, etc. Children born in this period are added to the population without restriction and are being used to seed sessions, thereby getting evaluated. At the end of an epoch, the population size can be reset to its initial value N by selecting N individuals for survival based on their fitness. Here again, the selection can be deterministic (keeping the best N) or probabilistic giving better individuals a higher chance to survive.

[4] Notice that such a co-existence would mean that the population size is not constant.

4.5 The web service example: system review

Our web service application has a number of properties worth further consideration. From the perspective of the whole system, it is an example of self-optimisation. Starting with a set of randomly generated or manually engineered session handling strategies (that is, a population of vectors $\bar{p}$), the system is continuously improving the service level (optimising $g(\bar{p})$). In principle, the system is also able to cope with changing circumstances, for instance changes in the types of visitors requiring new strategies to provide high quality service. Population-based search methods, like EAs, are in general capable of tracking moving optima, although for applications where coping with time varying objectives is essential specific extensions might be required to boost this property, cf. [3] and [6, Chapter 13.4].

From an evolutionary computing perspective we can observe that the EA as described above has no selection pressure (i.e., positive bias towards fitter candidates) during parent selection, only during survivor selection. This is, in principle, no problem. To prevent degradation to random walk, an EA must have selection pressure *somewhere*, either in parent or in survivor selection, but not necessarily in both. Many common EAs have fitness-related bias only in one selection procedure, e.g., generational GAs have no survivor selection (all children survive), while evolution strategies "lack" parent selection. It would not be difficult, however, to add bias when selecting parents in our system. We could simply require that invocation of a vector $\bar{p}$ from the population for a new session be based on fitness information (the utility function g).

As opposed to regular EAs, where population updates are neatly arranged consecutively, here we have a completely asynchronous process, where at a given time some individuals might undergo evaluation (by being used in a user session), some others might be mutated (because their session has just been finished), and yet others might be being deleted. For this reason, the evolutionary process in our example shows more resemblance with natural evolution than most EAs do. Technically, we could say that our EA is performing distributed optimisation in the sense that different candidate solutions are processed independently (in different sessions), possibly on different machines (web servers). A good solution found in some "corner" of the system, however, can quickly proliferate – in an evolutionary system highly fit individuals will always dominate weaker ones and spread over time.

Another aspect where our system is more "natural" than many other EAs is the behaviour based evaluation. In most EA applications the problem at hand can be modelled in such a way that the fitness function is a straightforward input-output mapping, a formula. Think, for instance, of the TSP example in Section 2 of this paper, where we only need a simple sum of distances of the edges represented by a permutation to calculate its fitness. In the web service example application a candidate solution has to *do* something, rather than just *be* something. A trivial consequence of this is that fitness evaluations can take a long time, in our case a whole session with a visitor of the web site. In general, this implies that relatively few candidates can be evaluated in a given amount of time. In other words, evolution will be relatively slow. Large populations and/or many generations are usually advantageous for getting good results, but in our case these might not be feasible. This might cause progress at a slow rate and necessitate special (application dependent) tricks to obtain satisfactory performance.

Last, but not least, let us note that there is no self-adaptation, or any other mechanism, in this EA to change its own parameters on-the-fly. The EA is applied for real-time (self-)optimisation of the system providing the web services without optimising itself. This shows that self-adaptation on EA level is not a requirement for self-optimisation on system level.

5 Links between evolutionary and autonomic computing

From the self-* perspective we can summarise the most important properties of evolutionary algorithms as follows:

1. EAs form a (meta)heuristic that can be used to solve optimisation problems. By the presence of a population of candidate solutions EAs are inherently suited to cope with time varying optimisation objectives.
2. EAs need to be optimised themselves, in particular, their parameter settings have to be determined appropriately for maximum performance. Because evolutionary search consists of different stages, optimal parameter values depend on time.
3. EAs are capable to perform real-time self-optimisation. To this end, self-adaptation is a particularly successful technique that is able to determine appropriate algorithm parameters following the progress of the search process, thus handling time dependency of optimal parameters given a stationary problem. Furthermore, it can also deal with changing objectives, resetting and re-optimising parameters automatically, without any user intervention.
4. EAs are inherently distributed and parallelisable because different members of the population can be naturally allocated to different processors.

It is rather clear from this list that evolutionary computing in general, and existing techniques within evolutionary computing in particular, can be used to meet some canonical challenges in autonomic computing, for instance, self-optimisation. Additionally, the evolutionary paradigm can serve as a source of inspiration, or let us say as a generic approach, to achieve other self-* properties, like self-configuration or self-healing. There exists related work also advocating population based approaches, such as multi-agent systems and ant-colony optimisation [9, 7].

To see the relevance of autonomic computing to evolutionary computing let us recall the problem of parameter control in EC. During the last decade it become increasingly clear within the field that the numerous EA parameters have a complex relationship with each other, or more precisely, a combined, non-linear influence on algorithm performance. Since non-linear problems with many interacting parameters belong to the niche of EC, it is a natural idea to use an evolutionary system to optimise itself on-the-fly, cf. [4] and [6, Chapter 8]. Self-optimisation or self-configuration has thus became one of the great challenges of evolutionary computing. Existing techniques, like self-adaptation of mutation step-sizes, can solve this problem partially, but a completely parameterless EA requires much more, for instance regulating selection pressure, population size, mutation and recombination parameters *simultaneously*. From this perspective, an evolving system can be seen as a special case of an autonomic system and it can be expected that some solutions invented within autonomic computing can be transferred to EC, meaning indeed that the two fields would share solutions to common problems.

Acknowledgement

I am grateful for M. Jelasity for the discussions about a suitable application example.

References

1. T. Bäck, A.E. Eiben, and N.A.L. van der Vaart. An empirical study on GAs "without parameters". In M. Schoenauer, K. Deb, G. Rudolph, X. Yao, E. Lutton, J.J. Merelo, and H.-P. Schwefel, editors, *Proceedings of the 6th Conference on Parallel Problem Solving from Nature*, number 1917 in Lecture Notes in Computer Science, pages 315–324. Springer, Berlin, Heidelberg, New York, 2000.
2. T. Bäck, D.B. Fogel, and Z. Michalewicz, editors. *Evolutionary Computation 1: Basic Algorithms and Operators.* Institute of Physics Publishing, Bristol, 2000.
3. J. Branke and H. Schmeck. Designing evolutionary algorithms for dynamic optimization problems. In A. Ghosh and S. Tsutsui, editors, *Advances in Evolutionary Computating: Theory and Applications*, pages 239–262. Springer, Berlin, Heidelberg, New York, 2003.
4. A.E. Eiben, R. Hinterding, and Z. Michalewicz. Parameter control in evolutionary algorithms. *IEEE Transactions on Evolutionary Computation*, 3(2):124–141, 1999.
5. A.E. Eiben and M. Schoenauer. Evolutionary computing. *Information Processing Letters*, 82:1–6, 2002.
6. A.E. Eiben and J.E. Smith. *Introduction to Evolutionary Computing.* Springer, Berlin, Heidelberg, New York, 2003.
7. Luca Maria Gambardella. Engineering complex systems: Ant colony optimization to model and to solve complex dynamic problems. In *SELF-STAR: International Workshop on Self-* Properties in Complex Information Systems.* Bolgna, Italy, June, 2004.
8. F. Hoffmeister and T. Bäck. Genetic self-learning. In F.J. Varela and P. Bourgine, editors, *Toward a Practice of Autonomous Systems: Proceedings of the 1st European Conference on Artificial Life*, pages 227–235. MIT Press, Cambridge, MA, 1992.
9. Gerald Tesauro, David M. Chess, William E. Walsh, Rajarshi Das, Alla Segal, Ian Whalley, Jeffrey O. Kephart, and Steve R. White. A multi-agent systems approach to autonomic computing. Technical Report RC23357 (W0410-015), IBM research Division, October 2004.