Cluster Comput
DOI 10.1007/s10586-010-0122-y

Plato: a genetic algorithm approach to run-time reconfiguration in autonomic computing systems

Andres J. Ramirez · David B. Knoester · Betty H.C. Cheng · Philip K. McKinley

Received: 3 October 2009 / Accepted: 10 February 2010

Abstract Increasingly, applications need to be able to self-reconfigure in response to changing requirements and environmental conditions. Autonomic computing has been proposed as a means for automating software maintenance tasks. As the complexity of adaptive and autonomic systems grows, designing and managing the set of reconfiguration rules becomes increasingly challenging and may produce inconsistencies. This paper proposes an approach to leverage genetic algorithms in the decision-making process of an autonomic system. This approach enables a system to dynamically evolve target reconfigurations at run time that balance tradeoffs between functional and non-functional requirements in response to changing requirements and environmental conditions. A key feature of this approach is incorporating system and environmental monitoring information into the genetic algorithm such that specific changes in the environment automatically drive the evolutionary process towards new viable solutions. We have applied this genetic-algorithm based approach to the dynamic reconfiguration of a collection of remote data mirrors, demonstrating an effective decision-making method for diffusing data and minimizing operational costs while maximizing data reliability and network performance, even in the presence of link failures.

Keywords Autonomic computing · Evolutionary algorithm · Genetic algorithm · Intelligent control · Distributed systems

A.J. Ramirez (✉) · D.B. Knoester · B.H.C. Cheng · P.K. McKinley
Michigan State University, 3115 Engineering Building, East Lansing, MI 48823, USA
e-mail: ramir105@cse.msu.edu

D.B. Knoester
e-mail: dk@cse.msu.edu

B.H.C. Cheng
e-mail: chengb@cse.msu.edu

P.K. McKinley
e-mail: mckinley@cse.msu.edu

1 Introduction

As distributed computing applications grow in size and complexity, it is increasingly difficult to build a system that satisfies all requirements and design constraints to be encountered during its lifetime. Many types of systems must operate continuously, thus disallowing periods of downtime while humans modify code and fine-tune the system. For instance, several studies [5, 30] document the severe financial penalties incurred by companies when facing problems such as data loss and data inaccessibility. As a result, it is important for applications to be able to self-reconfigure in response to changing requirements and environmental conditions [24]. IBM proposed autonomic computing as a means for automating software maintenance tasks [17]. Autonomic computing refers to any system that manages itself based on a system administrator's high-level objectives while incorporating capabilities such as self-reconfiguration and self-optimization. Typically, developers encode reconfiguration strategies at design time, and the reconfiguration tasks are influenced by anticipated future execution conditions [3, 8, 13, 33]. We propose an approach for harnessing genetic algorithms [10] to support the decision-making process of an autonomic system. This approach enables a decision-making process to dynamically evolve target reconfigurations at run time.

Autonomic systems typically comprise three key elements: monitoring, decision-making, and reconfiguration.

Published online: 05 March 2010

Monitoring enables an application to be aware of its environment and detect conditions that warrant reconfiguration; *decision-making* determines which set of monitored conditions should trigger a specific reconfiguration response; and *reconfiguration* enables an application to change itself in order to fulfill its requirements. Many adaptive and autonomic systems have applied rule-based decision-making approaches to match particular events against specific reconfigurations [7, 13]. Others leverage utility-based decision-making approaches to address multiple reconfiguration objectives and dimensions [3, 33]. These approaches, however, enable a system to self-adapt only against scenarios that were considered at design time. Furthermore, as the complexity of adaptive logic grows, designing and managing the set of reconfiguration rules becomes unmanageable and potentially inconsistent [3]. To address these concerns, several researchers have applied evolutionary computation techniques to the design of adaptive and autonomic systems [8, 9, 19]. Although these approaches enable developers to explore richer sets of behavioral models that satisfy system and adaptation requirements, most are applicable only at design time due to the significant amount of computational time required to evolve target reconfigurations.

This paper proposes an approach to incorporate genetic algorithms in the decision-making process of autonomic systems. A *genetic algorithm* is a stochastic search-based technique for finding solutions to optimization and search-based problems [10]. Genetic algorithms comprise a population of individuals that encode candidate solutions in the search space. Domain-specific fitness functions are used to determine the quality of each individual. Genetic algorithms use this fitness information, along with the processes of selection, crossover, and mutation, to direct the search process towards promising parts of the search space. In practice, genetic algorithms can be surprisingly fast in efficiently searching complex, highly nonlinear, and multidimensional search spaces [20]. For this work, we developed Plato, a genetic-algorithm based decision-making process. We demonstrate that Plato can search solution spaces for target reconfigurations, generating suitable reconfigurations in response to changing requirements and environmental conditions. A key benefit of Plato is that developers need not prescribe reconfigurations in advance to address specific situations warranting reconfiguration. Instead, Plato harnesses the power of natural selection to evolve suitable target reconfiguration at run time.

We have applied Plato to the dynamic reconfiguration of an overlay network [2] for diffusing data to a collection of remote data mirrors [11, 14]. Plato evolved suitable target reconfigurations at run time that balanced tradeoffs between functional and non-functional requirements as requirements and environmental conditions changed. Specifically, Plato evolved overlay networks for data diffusion that balanced the competing objectives of minimizing operational costs while maximizing data reliability and network performance, even while facing adverse conditions such as repeated link failures. Our results suggest genetic algorithms provide an effective method of online decision-making in autonomic computing systems, enabling run-time generation of target reconfigurations that balance competing objectives while accounting for environmental dynamics.

The remainder of this paper is organized as follows. In Sect. 2 we provide background material on remote data mirroring systems and genetic algorithms. Section 3 describes Plato's design and how it can be integrated with the decision-making process of an autonomic computing system. Section 4 presents the experimental results obtained when applying Plato to the dynamic reconfiguration of a network of remote data mirrors, followed by discussion in Sect. 5. A preliminary report on this study was presented in [28]. The extended version presented here includes additional details of the Plato design and the experimental setup, as well as an entirely new section assessing the ability of Plato to respond to degrading network failures. Section 6 reviews other approaches to decision-making and dynamic configuration of networks involving genetic algorithms and other evolutionary computation techniques. Lastly, in Sect. 7, we summarize our main findings and discuss future directions for our work.

2 Background

This section reviews two topics fundamental to this paper. First, we describe the concept of remote data mirrors and discuss the challenges that complicate their design. We then describe genetic algorithms and explain the mechanisms by which they efficiently search for suitable solutions amidst complex solution landscapes.

2.1 Remote mirroring

Remote data mirroring is a particular form of data mirroring in which copies of critical data are stored at one or more secondary sites. The main objective of remote data mirroring is to isolate the protected data from failures that may affect the primary copy [15]. Thus, by keeping two or more copies of important information physically isolated from each other, access can continue if one copy is lost or becomes unreachable [11, 35]. In the event of a failure, recovery typically involves either a site failover to one of the remote data mirrors, or data reconstruction. Designing and deploying a remote mirror, however, is a complex and expensive task that should be done only when the cost of losing data outweighs the cost of protecting it [11]. For instance, *ad hoc* solutions

may provide inadequate data protection, poor write performance, and incur high network costs [15]. Similarly, over-engineered solutions may incur expensive operational costs to defend against negligible risks.

Two important design choices for remote data mirrors include the type of network link connecting the mirrors and the remote mirroring protocol. Each network link incurs an operational cost. In addition, each network link has a measurable throughput, latency, and loss rate that determine the overall remote mirror design performance [15]. Remote mirroring protocols, which can be categorized as either synchronous or asynchronous propagation, affect both network performance and data reliability. In *synchronous propagation* the secondary site receives and applies each write before the write completes at the primary site [15]. In batched *asynchronous propagation* [15], updates accumulate at the primary site and are periodically propagated to the secondary site, which then applies each batch atomically. While synchronous propagation achieves zero potential data loss, it consumes large amounts of network bandwidth. Batched asynchronous propagation, on the other hand, achieves more efficient network performance than synchronous propagation, but may have a higher potential data loss.

2.2 Genetic algorithms

Genetic algorithms are stochastic-based search techniques that comprise a population of individuals, where each individual encodes a candidate solution in a chromosome [10]. In each generation, the fitness, or quality, of every individual is calculated, and a subset of individuals is selected, then recombined and/or mutated to form the next generation. For example, Fig. 1 shows two individuals in a population of overlay network topologies. Each individual contains a vector to encode which overlay links are used in the corresponding topology. Specifically, given the underlying network topology vector $\langle AB, BC, CD, AD, AC, BD\rangle$, a 1 indicates the link is part of the overlay and a 0 otherwise. Other link properties, such as the propagation methods described above, can be encoded in such vectors. In addition, each individual has an associated *fitness value* that is determined by evaluating the encoded solution against certain domain-specific fitness functions. This fitness information enables a genetic algorithm to choose promising solutions for further evolution in the next generation.

Genetic algorithms use *crossover* to exchange building blocks between two fit individuals, hopefully producing offspring with even higher fitness values. The two most common forms of crossover in genetic algorithms are one-point and two-point crossover. In *one-point crossover*, a position in the chromosome is selected at random and the parts of two parents after the crossover position are exchanged. In *two-point crossover*, two positions are chosen at random and the segments between them are exchanged. Figure 2 illustrates two-point crossover for the representation described in Fig. 1. Specifically, the link properties between $\langle CD, AD\rangle$, denoted by the underlined genome segment for clarity, are exchanged between both parents to form two new offspring. As a result, Offspring I deactivates link $\langle CD\rangle$ from Parent I and activates link $\langle AD\rangle$ from Parent II. Likewise, Offspring II deactivates link $\langle AD\rangle$ from Parent II, and activates link $\langle CD\rangle$ from Parent I.

Unfortunately, genetic algorithms are susceptible to deceptive landscapes, possibly evolving suboptimal solutions (i.e., local maxima). The key idea behind *mutation* is to introduce genetic variation that may have been lost throughout the population as a result of the selection process [10]. Mutation takes an individual and randomly changes parts of its encoded solution based on some specified mutation rate.

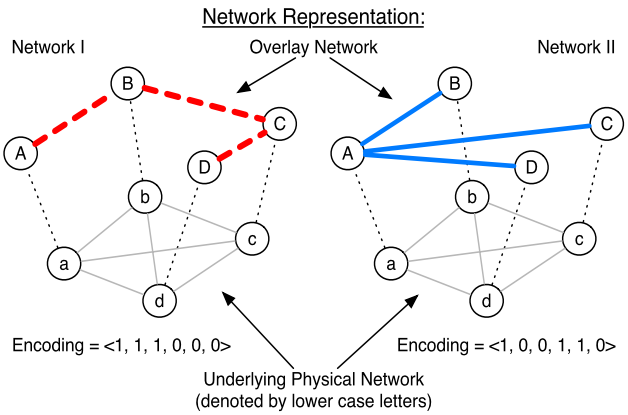

Fig. 1 Encodings of two overlay network solutions as individuals in a genetic algorithm

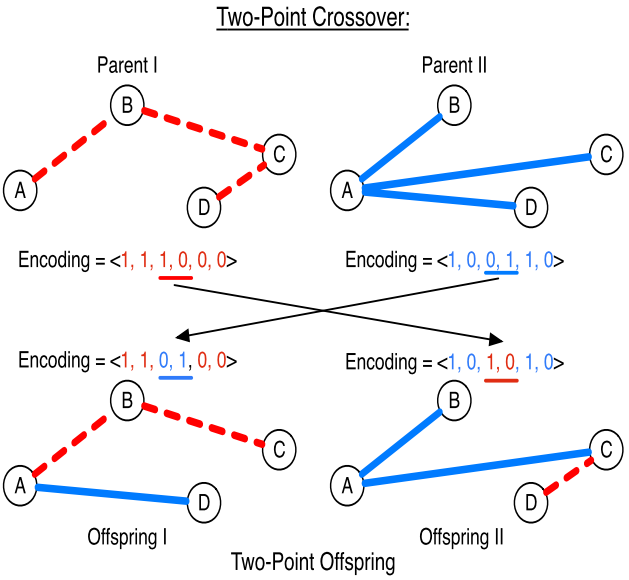

Fig. 2 Performing two-point crossover in overlay networks

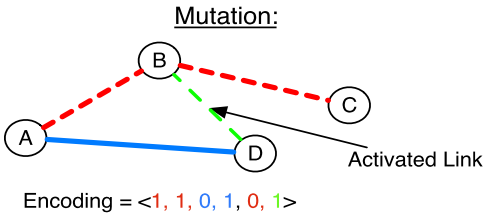

Fig. 3 Performing mutation on an overlay network

For example, Fig. 3 illustrates a sample mutation where link $\langle BD \rangle$ is activated in one of the individuals. The activation of this link would not have been possible with the crossover operator only since link $\langle BD \rangle$ is not active in either parent. As such, the key effect of the mutation operator is to explore additional areas of the solution space that may not be reachable through the crossover operator alone.

Genetic algorithms are typically executed until one of two conditions is met: Either the allocated execution time is exhausted or the algorithm converges upon a particular solution. If the execution time has been exhausted, then the best solution found thus far is considered to be the solution. Otherwise, if the algorithm has converged upon a particular solution, then the solution should be assessed for its quality. In particular, it is possible for the algorithm to converge prematurely upon a suboptimal solution. One strategy often used to address this problem is to reseed and restart the genetic algorithm while altering some configuration parameters such as crossover and mutation rates, forcing the search process to explore different parts of the solution landscape.

3 Proposed approach

We designed Plato with the goal of deciding how and when to diffuse data to a collection of remote data mirrors across a potentially unreliable network [11, 15]. Specifically, given a network containing a set of remote data mirrors, Plato must construct and maintain an overlay network such that data may be distributed to all nodes while satisfying the following requirements. First, the overlay network must remain connected at all times. If the overlay network becomes disconnected for any reason, then it must be reconfigured to re-establish connectivity. Second, it should never be the case that the constructed overlay network exceeds the allocated monetary budget specified by the end-user. Third, data should be distributed as efficiently as possible, that is, the amount of bandwidth consumed when diffusing data should be minimized. All considerations combined, the task of data diffusion is a multi-objective problem in which data must be distributed as efficiently as possible, while minimizing expenses and potential data loss.

Extensive research has been conducted on many aspects of self-adaptive and autonomic systems [17, 24, 33]. As a result, we assume the existence of a monitoring [7, 26, 27, 31] and reconfiguration [7, 29, 36] infrastructure to support self-adaptation with assurance [37, 38] at run time. As Fig. 4 illustrates, in the proposed approach, the decision-making process comprises a genetic algorithm that accepts monitoring information as input and produces target reconfigurations as output. The monitoring infrastructure periodically observes both the system and its execution environment and reports those values to the decision-making process. The decision-making process interprets the monitoring data, determines when a reconfiguration is required, and selects the appropriate target reconfiguration. The reconfiguration infrastructure effects the changes throughout the system using an adaptation driver.

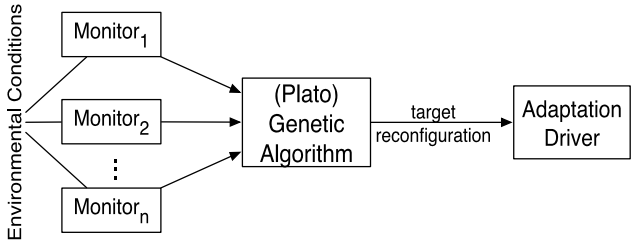

Fig. 4 Overview of the Plato approach

Table 1 Update interval and corresponding data batch sizes for link propagation methods [15]

Time interval	Avg. data batch size
0	0 GB
1 min	0.0436 GB
5 min	0.2067 GB
1 hr	2.091 GB
4 hrs	6.595 GB
12 hrs	15.12 GB
24 hrs	27.388 GB

3.1 Plato design

Representation. Developers must select a suitable representation scheme for encoding candidate solutions as individuals in a genetic algorithm. For example, in Plato, every individual in the population encodes a complete overlay network, where each link is either active or inactive and is associated with one of seven possible propagation methods (Table 1). Table 1 lists the average data batch size associated with each of the seven propagation methods, previously reported by Keeton et al. [15]. This table also illustrates that larger amounts of data can be lost due to a failure as the amount of time for data propagation increases. In particular, synchronous propagation (time interval equal to 0) pro-

vides the maximum amount of data protection while asynchronous propagation with a 24-hour time interval provides the least amount of data protection. Nonetheless, as the level of data protection increases, the overall performance of the network decreases. Specifically, as the average data batch size decreases, fewer opportunities arise for coalescing data writes. As a result, synchronous propagation methods typically incur worse network performance than asynchronous propagation methods with larger data batch sizes [15].

Plato uses a representation scheme similar to the one illustrated in Fig. 1, where each overlay network link can be set to either active or inactive and is associated with one of the seven propagation methods presented in Table 1. In terms of complexity, an individual with this representation scheme comprises a vector of $\frac{n(n-1)}{2}$ links that can be activated and configured to form an overlay network. With 25 nodes, for example, there are $2^{300} * 7^{300}$ possible overlay network configurations. The total number of possible configurations makes it infeasible to exhaustively evaluate all possible configurations in a reasonable amount of time.

GA operators. Crossover and mutation operators are specific to the encoding scheme used. The default crossover and mutation operators are designed to work on fixed-length binary string representations [10]. If a different representation scheme is used to encode candidate solutions for a particular domain, then specialized crossover and mutation operators need to be developed and applied. For example, each individual in Plato encodes a candidate solution that comprises binary, integer, and floating-point values. As a result, Plato uses domain-specific crossover and mutation operators to directly manipulate overlay network topologies. The crossover operator used by Plato (shown in Fig. 2) exchanges link properties between two parents. Specifically, two network links in the link vector are selected at random and the segments between them are exchanged. Likewise, the mutation operator used by Plato (shown in Fig. 3) randomly activates/deactivates a link and changes its propagation method.

GA setup. Developers must set up the GA for the problem being solved. Typical parameters include population size, crossover type, crossover and mutation rates, selection methods, and the alloted execution time, typically expressed in generations. Table 2 specifies the parameters and values used for Plato. In particular, notice that for each generation, Plato performs two-point crossover on 10 individuals, thereby producing 20 offspring for the next generation. Likewise, Plato mutates approximately 5 individuals each generation. To select which individuals *survive* to the next generation, Plato applies tournament selection. In tournament selection, the fitness values of two randomly selected individuals from the population are compared. The individual with the higher fitness value is moved to the next generation.

Table 2 Genetic algorithm parameters and values used in Plato for this study

Parameter	Value
Max. population size	100
Crossover type	Two-point
Crossover rate	10%
Mutation rate	5%
Selection method	Tournament ($K = 2$)
Max generations	2500

3.2 Fitness sub-functions

In general, a single fitness function is not sufficient to quantify all possible effects of a particular reconfiguration when balancing multiple objectives [3, 4]. Instead, developers should define a *set* of fitness sub-functions to evaluate a target reconfiguration according to the optimization dimensions specified by end-users. Each fitness sub-function should have an associated coefficient that determines the relative importance of that sub-function in comparison to others. A weighted sum can be used to combine the values obtained from each fitness sub-function into one scalar value [6, 21, 25].

Plato uses several fitness sub-functions to approximate the effects of a particular overlay network in terms of costs, network performance, and data reliability. Most of the fitness sub-functions used by Plato were derived from studies on optimizing data reliability solutions [15] and modified for our specific problem. This set of sub-functions enables Plato to search for overlay networks that not only satisfy the previously mentioned properties, but that also yield the highest fitness based on how the end-user defined the sub-function coefficients to reflect the priorities for the various fitness sub-functions.

We use the following fitness sub-function to calculate an overlay network's fitness in terms of *cost*:

$$F_{\text{cost}} = 100 - \left(100 * \frac{cost}{budget}\right)$$

where cost is the sum of operational expenses incurred by all active links, and budget is an end-user supplied constraint that places an upper bound on the maximum amount of money that can be allocated for operating the overlay network. This fitness sub-function, F_{cost}, guides the genetic algorithm toward overlay network designs that minimize operational expenses.

Likewise, we use the following two fitness sub-functions to calculate an overlay network's fitness in terms of *performance*:

$$F_{\text{perf}_1} = 50 - \left(50 * \frac{latency_{\text{avg}}}{latency_{\text{wc}}}\right),$$

and

$$F_{\text{perf}_2} = 50 * \left(\frac{band_{\text{sys}} - band_{\text{eff}}}{band_{\text{sys}}} + bound \right),$$

where $latency_{\text{avg}}$ is the average latency over all active links in the overlay network, $latency_{\text{wc}}$ is the largest latency value measured over all links in the underlying network, $band_{\text{sys}}$ is the total available bandwidth across the overlay network, $band_{\text{eff}}$ is the total effective bandwidth across the overlay network after data has been coalesced, and bound is a limit on the best value that can be achieved throughout the network in terms of bandwidth reduction. The first fitness sub-function, F_{perf_1}, accounts for the case where choosing links with lower latency will enable faster transmission rates. Likewise, the second fitness sub-function, F_{perf_2}, accounts for the case where network performance can be increased by reducing the amount of data sent across a network due to write coalescing. We note that the maximum achievable value of $F_{\text{perf}_1} + F_{\text{perf}_2}$ is 100.

Lastly, we use the following two fitness sub-functions to calculate an overlay network's fitness in terms of *reliability*:

$$F_{\text{rel}_1} = 50 * \frac{links_{\text{used}}}{links_{\text{max}}};$$

and

$$F_{\text{rel}_2} = 50 - \left(50 * \frac{dataloss_{\text{potential}}}{dataloss_{\text{wc}}} \right);$$

where $links_{\text{used}}$ is the total number of active links in the overlay network, $links_{\text{max}}$ is the maximum number of possible links that could be used in the overlay network given the underlying network topology, $dataloss_{\text{potential}}$ is the total amount of data that could be lost during transmission as a result of the propagation method (see Table 1), and dataloss_{wc} is the amount of data that could be lost by selecting the propagation method with the largest time window for write coalescing. The first reliability fitness function, F_{rel_1}, accounts for the case where an overlay network with redundant links may be able to tolerate link failures while maintaining connectivity. The second reliability fitness function, F_{rel_2}, accounts for the case where propagation methods leave data unprotected for a period of time. We note that the maximum achievable value of $F_{\text{rel}_1} + F_{\text{rel}_2}$ is 100, the same as the fitness sub-functions for *cost* and *performance*.

The following fitness function combines the previous fitness sub-functions into one scalar value:

$$FF = \alpha_{\text{cost}} * F_{\text{cost}} + \alpha_{\text{perf}} * (F_{\text{perf}_1} + F_{\text{perf}_2}) + \alpha_{\text{rel}} * (F_{\text{rel}_1} + F_{\text{rel}_2}),$$

where α_i's represent weights for each dimension of optimization as encoded into the genetic algorithm by the end user. These coefficients can be scaled to guide the genetic algorithm towards particular designs that reflect different priorities of non-functional properties. For example, if developers want to evolve types of overlay network designs that optimize only with respect to *cost*, then α_{cost} could be set to 1 and α_{perf} and α_{rel} could be set to 0.

To integrate Plato into the application and factor current environmental conditions into the target reconfiguration, developers must define a global view construct that reflects the executing system and its environment. This construct will be updated by the monitoring infrastructure and accessed by the genetic algorithm's fitness sub-functions when evaluating candidate reconfigurations. For instance, although Plato currently simulates the network monitoring process, each candidate overlay network in Plato stores information about the underlying complete network topology. Specifically, each link stores values such as throughput, latency, loss rate, etc. As a result, when Plato evaluates a candidate overlay network, it computes its fitness value based on current system and environmental conditions.

Rescaling sub-functions. If requirements are likely to change while the application executes, then developers should introduce code to *rescale* the coefficients of individual fitness sub-functions at run time. In particular, the fitness landscape is shifted when the coefficients of a fitness sub-function are rescaled. By updating the relevance of each fitness sub-function at run time, the genetic algorithm will be capable of evolving target reconfigurations that address changes in requirements and environmental conditions without specifying *how* the system should be reconfigured.

For example, when an overlay network link fails, Plato automatically doubles the current coefficient for network reliability. Note that although we prescribe how these coefficients should be rescaled in response to high-level monitoring events, we do not explicitly specify target reconfigurations. That is, Plato does not prescribe how many links should be active in the overlay network, or what their propagation methods should be.

4 Case study

We conducted a case study to evolve solutions to address the problem of diffusing data to a set of remote data mirrors across dynamic and unreliable networks. Each experiment focuses on a single aspect of this problem, namely constructing and maintaining an overlay network that enables the distribution of data to all nodes. Different environmental factors and scenarios presented throughout these experiments provide insight with respect to the suitability of genetic algorithms for decision-making in autonomic systems. Note that we simulated these experiments on a MacBook Pro with a 2.53 GHz Intel Core 2 Duo Processor and 4 GB of RAM.

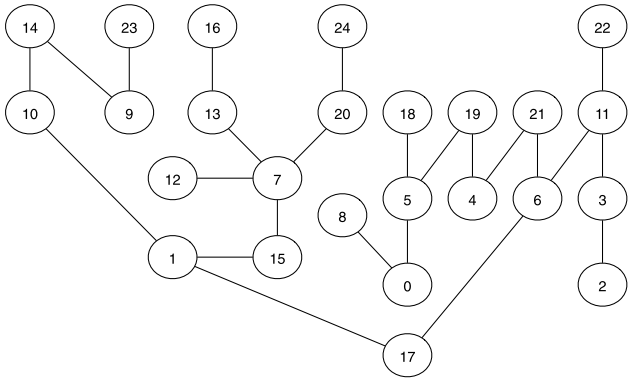

Fig. 5 Overlay network produced when optimizing for cost

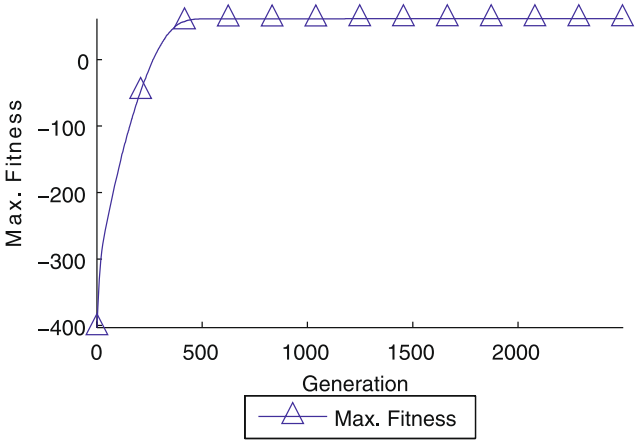

Fig. 6 Fitness of overlay networks when optimizing for cost only

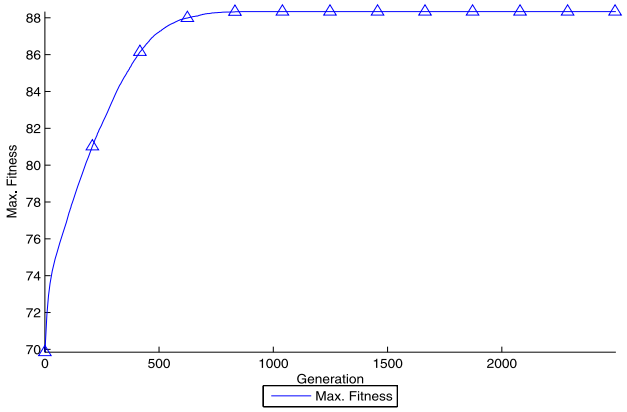

Fig. 7 Fitness of overlay networks when optimizing for reliability only

For each set of results presented, we performed 30 trials of the experiment, to account for the stochastic variation of genetic algorithms, and present the averaged results.

4.1 Single-dimensional optimization

The objective of this study is to confirm that, for degenerate scenarios involving single fitness sub-functions (i.e., operational cost, performance, and reliability), Plato will produce solutions consistent with those that can be predicted. In particular, we configured Plato to consider operational costs only, i.e., $\alpha_{\text{cost}} = 1$, $\alpha_{\text{perf}} = 0$, $\alpha_{\text{rel}} = 0$. As a representative example, consider the evolved overlay network shown in Fig. 5. This overlay network comprises 24 links and connects all remote data mirrors. The genetic algorithm was able to reduce the overlay network to a spanning tree that connects all remote data mirrors while incurring operational costs significantly below the maximum allocated budget.

Figure 6 shows the maximum fitness achieved by the candidate overlay networks as Plato executed. Plato converged upon an overlay network topology with a fitness value of approximately 50, indicating that Plato found overlay networks whose operational costs were roughly 50% of the allocated budget. Although the first few hundred generations obtained negative fitness values due to ill-formed candidate topologies that were either disconnected or exceeded the allocated budget, Plato found suitable overlay network designs by generation 500 (approximately 30 seconds on the MacBook Pro), well within the practical range for applications such as remote data mirroring.

For the next experiment, reliability was chosen as the single optimization criterion, i.e., $\alpha_{\text{cost}} = 0$, $\alpha_{\text{perf}} = 0$, $\alpha_{\text{rel}} = 1$. The evolved overlay network provides the maximum amount of reliability possible by activating all 300 links. Furthermore, the dominant propagation method for this overlay network was synchronous propagation, which minimizes the amount of data that can be lost during transmission.

Figure 7 plots the maximum fitness achieved by Plato in this experiment. Plato converged upon a maximum fitness value of 88. In the context of reliability, a value of 88 means that although the overlay network provides a high-level of data reliability, it is not completely immune against data loss. Specifically, even though all 300 links were activated in the overlay network to provide redundancy against link failures, not every link in the overlay network used a synchronous propagation method. Instead, a few links in the overlay network used asynchronous propagation methods with 1 and 5 minute time bounds. Nonetheless, we note the rapid development of fit individuals achieved by generation 600; by this point, Plato had evolved complete overlay networks with most links using synchronous propagation.

4.2 Multiple-dimensional optimization

The purpose of the next study is to assess whether Plato is able to efficiently balance multiple objectives, namely, minimizing operational costs while maximizing network perfor-

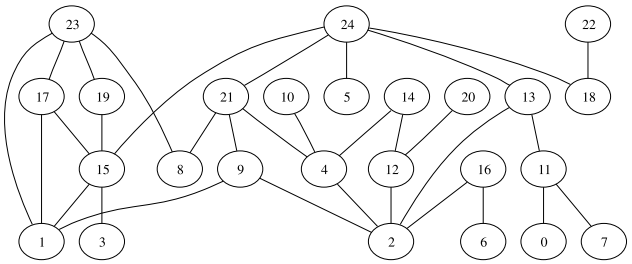

Fig. 8 Overlay network produced when optimizing for cost, performance, and reliability

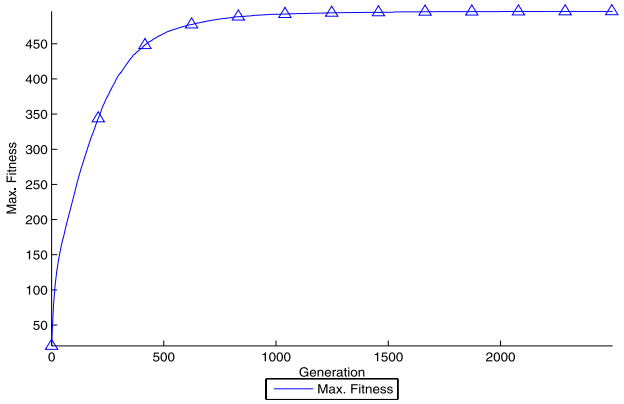

Fig. 9 Maximum fitness of overlay networks when optimizing for cost, performance, and reliability

mance and data reliability. For this particular experiment we configured Plato to produce network designs that emphasize network performance and reliability over operational costs, i.e., $\alpha_{\text{cost}} = 1$, $\alpha_{\text{perf}} = 2$, $\alpha_{\text{rel}} = 2$. Figure 8 shows a representative overlay network design that Plato evolved. This overlay network comprises 32 active links, the majority of which use asynchronous propagation methods with 1 and 5 minute time bounds. Overall, this overlay network provides a combination of performance and reliability while keeping operational expenses well below the allocated budget.

Figure 9 plots the average rate at which Plato converged on the resulting overlay network designs. On average, Plato terminated within 3 minutes. In particular, note that Plato found relatively fit overlay networks by generation 500 (approximately 30 seconds). Thereafter, Plato fine-tuned the overlay network to produce increasingly more fit solutions. For instance, Fig. 10 plots the average number of active links in the evolved overlay networks. At first, the more fit overlay networks were those that comprised the fewest number of active links while still maintaining connectivity. By the end of the run, 8 additional links had been added to the overlay network. Although these additional edges increased the overall operational cost of the overlay network, they also increased the network's fault tolerance against link failures, thus improving the overlay's reliability fitness value.

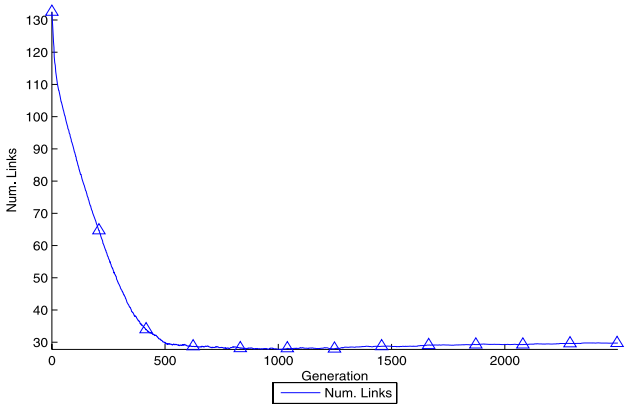

Fig. 10 Number of links active in overlay network when optimizing for cost, performance, and reliability

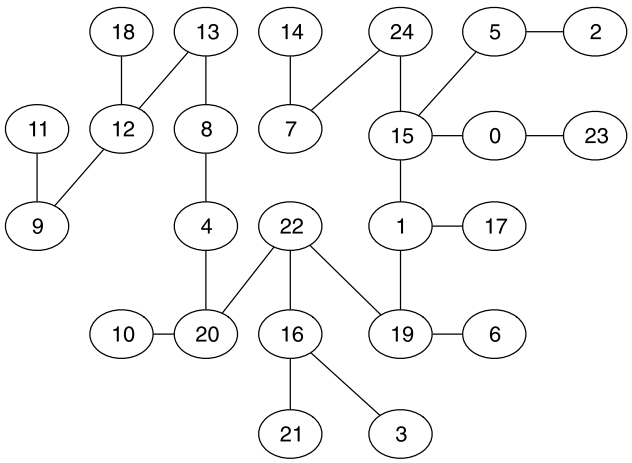

Fig. 11 Initial overlay network topology with cost being the lone design factor

Moreover, subsequent generations achieved higher fitness values by using asynchronous propagation methods of 5 minutes and 1 hour, thus improving network performance while providing some level of data protection during transmission.

4.3 Reconfiguration against link failures

The purpose of the next experiment is to assess the feasibility of using Plato to dynamically reconfigure the overlay network topology in real-time. We configured this experiment as a three-step process where we first ran Plato to produce an initial overlay network design whose primary design objective was to minimize operational costs, i.e., $\alpha_{\text{cost}} = 1$, $\alpha_{\text{perf}} = 0$, and $\alpha_{\text{rel}} = 0$. Although we could have generated a design to account for both cost and reliability, the objective of this experiment was to force the reconfiguration of the overlay network. Figure 11 presents a representative overlay network evolved by Plato that comprises 24 active links, a

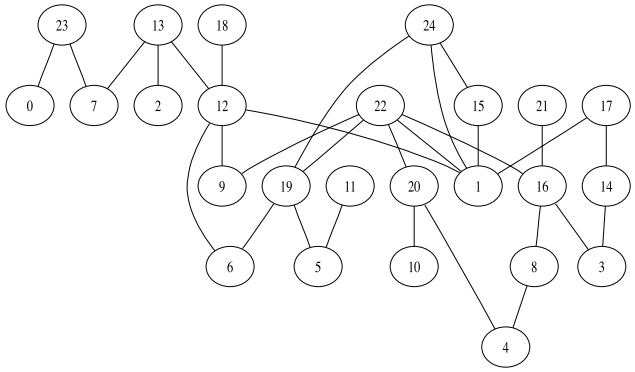

Fig. 12 Overlay network topology evolved in response to a link failure

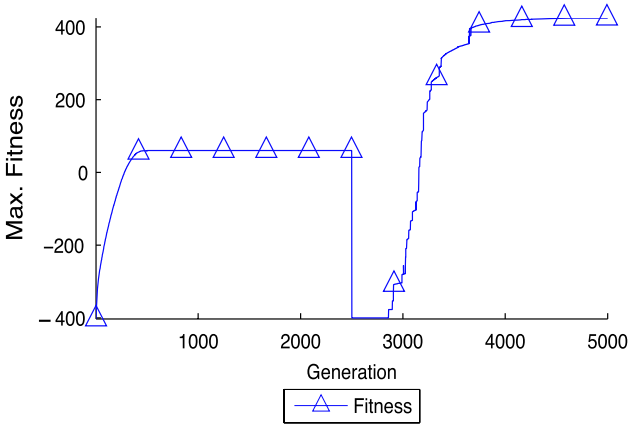

Fig. 13 Maximum fitness achieved before and after reconfiguration

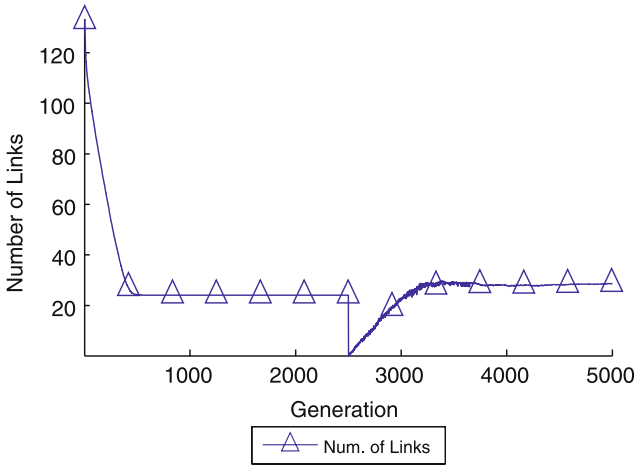

Fig. 14 Number of active links in overlay network before and after reconfiguration

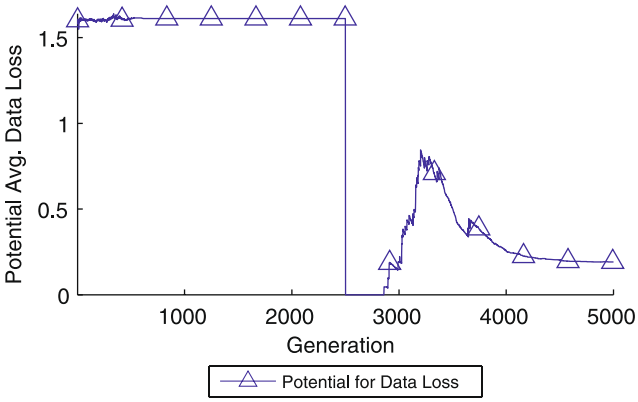

Fig. 15 Potential average data loss across overlay network before and after reconfiguration

spanning tree. We then randomly selected an active link in the overlay network and set its operational status to *faulty*. Since this link failure disconnected the network of remote data mirrors, we ran Plato again to evolve a target reconfiguration that addressed the changes in the underlying network topology.

In response, Plato evolved a new overlay network topology that addressed these environmental changes. Since the initial overlay network suffered from a link failure, the individual fitness sub-functions were automatically rescaled such that reliability became the primary design concern. Whenever an individual was evaluated, if the encoded overlay network made use of the faulty link, then it was severely penalized by assigning it a low fitness value. Figure 12 shows the overlay network that evolved in response to the environmental change in the underlying network. This new overlay network, with 6 redundant links, provides more reliability against link failures than the initial overlay network.

Figure 13 plots the maximum fitness achieved by Plato as it evolved both the initial and the reconfigured overlay network designs. We terminated an active link at generation 2500. As a result, the maximum fitness achieved at generation 2501 dropped to negative values. Within roughly 1000 generations (1 min), Plato had evolved considerably more fit overlay network topologies. Notice the relative difference in maximum fitness achieved by Plato before and after reconfiguration. The initial overlay network optimizes only with respect to operational costs, while the reconfigured overlay network optimizes primarily for reliability, but also optimizes with respect to operational costs. Since Plato doubled the coefficients for reliability in comparison to cost ($\alpha_{\text{cost}} = 1$, $\alpha_{\text{perf}} = 2$, and $\alpha_{\text{rel}} = 2$), candidate overlay networks after generation 2500 achieved a higher relative fitness value than the initial overlay network.

Figure 14 plots the average number of active links in both the initial and reconfigured overlay network designs. While the initial overlay design obtains a higher fitness by reducing the number of active links, the reconfigured overlay design obtains a higher fitness by adding several active links to improve robustness against future link failures.

Figure 15 plots the average potential data loss for a remote data mirror in both the initial and reconfigured over-

lay networks. The average potential data loss, which is a byproduct of the propagation methods, measures the amount of data, in gigabytes, that may be lost at a remote data mirror as a result of some failure. Lower average potential data loss values imply data is better protected against link failures and vice-versa. As this plot illustrates, after a link failure occurs, the reconfigured overlay network design reset *most* propagation methods to either synchronous or asynchronous propagation with a 1 or 5 minute time bound, thus improving data protection at the expense of degraded network performance.

4.4 Reconfiguration against successive link failures

The purpose of this experiment was to assess the feasibility of applying Plato to dynamically reconfigure an overlay network in real-time in response to multiple link failures. While the previous experiment assessed whether Plato could evolve suitable target reconfigurations in response to changing requirements and environmental conditions, this experiment assessed whether Plato can evolve target reconfigurations in response to more adverse conditions, where successive link failures occur throughout the network. As in the previous experiment, we first ran Plato to produce an initial overlay network design whose primary design objective was to minimize operational costs, i.e., $\alpha_{\text{cost}} = 1$, $\alpha_{\text{perf}} = 0$, and $\alpha_{\text{rel}} = 0$. Next, we randomly selected an active overlay network link and set its operational status to *faulty*, which prompts Plato to evolve new target reconfigurations in response to the changing underlying network. We repeated this process every 2,500 generations, the equivalent of one full iteration of Plato, for a maximum of ten consecutive link failures.

Figure 16 plots the maximum fitness achieved by Plato as it evolved overlay network designs. After the initial overlay network design became disconnected as a result of a single link failure, Plato automatically rescaled reconfiguration priorities to emphasize data reliability, i.e., $\alpha_{\text{cost}} = 1$, $\alpha_{\text{perf}} = 2$, and $\alpha_{\text{rel}} = 2$. This plot illustrates how Plato evolved target reconfigurations that withstood various successive link failures (depicted by the valleys) without the overlay network becoming disconnected. Specifically, on average, each generated target reconfiguration maintained a fitness value well above -400, which was the numerical penalty assigned to disconnected overlay networks. Also note that during each successive reconfiguration, the first few target reconfigurations generated by Plato were progressively lower in fitness, suggesting that as more links failed in the overlay network, it became more difficult for Plato to find promising areas in the solution space to explore further. Nonetheless, Plato evolved suitable target reconfigurations of the same overall fitness value in response to successive link failures. Moreover, Plato

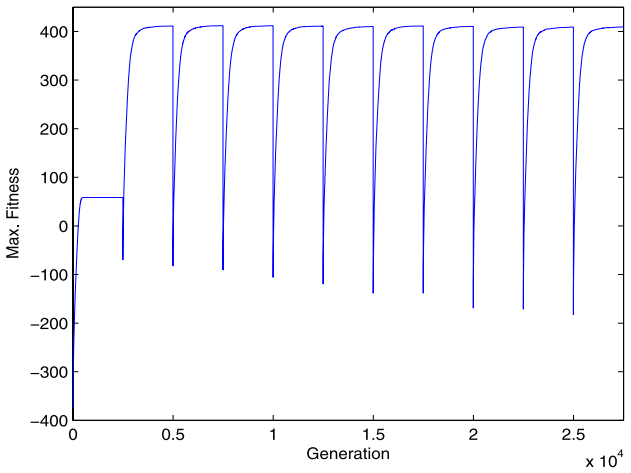

Fig. 16 Maximum fitness of overlay networks achieved throughout multiple reconfigurations

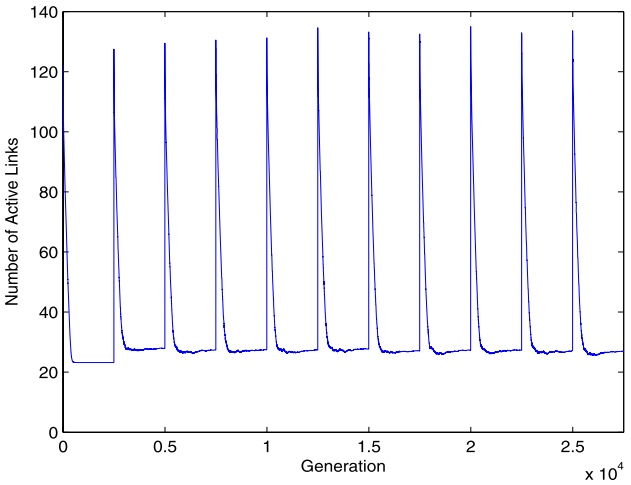

Fig. 17 Number of active links in overlay network throughout multiple reconfigurations

was able to consistently evolve viable target reconfigurations within 500 generations (approximately 30 seconds on a MacBook Pro).

The plot in Fig. 17 shows the average number of active links in the initial and reconfigured overlay network designs. In the initial overlay network design, Plato reduced the number of active links to form a spanning tree and minimize operational costs. In contrast, throughout each reconfiguration iteration, Plato increased the number of redundant active links to maximize data reliability while still attempting to minimize operational costs. As this plot illustrates, after each successive link failure, Plato generated target reconfigurations where overlay networks comprised approximately 30 active links. This plot also provides insight as to how Plato evolved suitable target reconfigurations that balanced maximizing data reliability and minimizing operational costs. Specifically, Plato activated many overlay net-

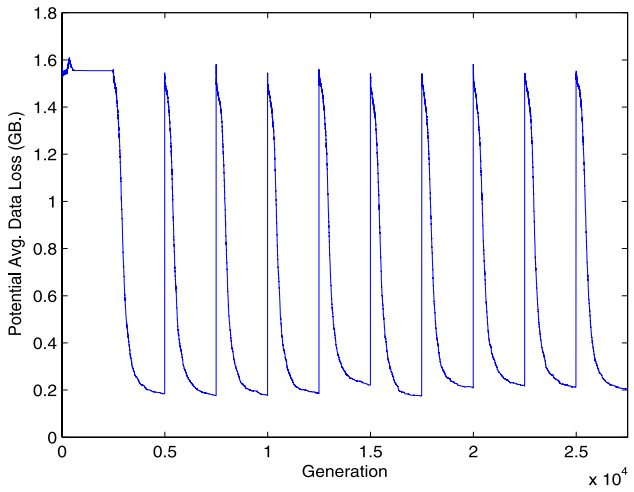

Fig. 18 Potential average data loss across overlay network throughout multiple reconfigurations

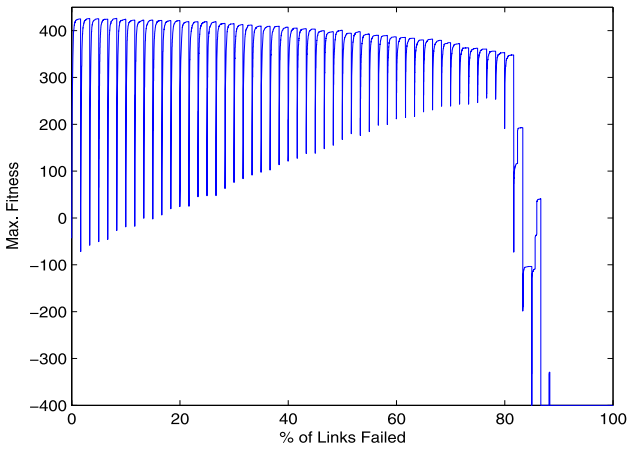

Fig. 19 Maximum fitness of overlay networks achieved throughout multiple reconfigurations until complete network failure

work links during the first few iterations of the genetic algorithm. Eventually, Plato pruned back the size of the overlay network by deactivating most redundant links in order to reduce operational costs. Do note that while some redundant links in the overlay network do increase operational costs, they also improve the robustness of the overlay network design against potential future link failures.

Finally, Fig. 18 plots the average potential data loss for a remote data mirror in the initial and reconfigured overlay network designs. The average potential data loss measures the amount of data, in gigabytes, that may be lost at a remote data mirror as a result of some type of failure. As this plot illustrates, after the initial overlay network design became disconnected, Plato rapidly evolved target reconfigurations where most propagation methods in the overlay network were set to either synchronous mode or asynchronous mode with a 1 or 5 minute time bound. These particular data propagation settings reduced the average potential data loss across the network by providing a higher level of data protection at the expense of degraded network performance.

4.5 Reconfiguration against complete network failure

We also conducted this experiment to assess the operational limits of applying Plato to dynamically reconfigure an overlay network in real-time in response to a degenerate number of link failures. Specifically, while the previous experiment assessed whether Plato could evolve suitable target reconfigurations in response to a scenario involving various link failures, this experiment seeks to assess whether Plato can evolve target reconfigurations in response to a continuously degrading network environment. As in previous experiments, we first ran Plato to produce an initial overlay network design. In particular, the objective of the initial overlay network design was to maximize data reliability while balancing performance and operational costs, i.e., $\alpha_{\text{cost}} = 1$, $\alpha_{\text{perf}} = 1$, and $\alpha_{\text{rel}} = 3$. Next, we randomly selected 5 overlay network links and set their operational status to *faulty*. Plato was restarted each time a network link failed in order to generate target reconfigurations that addressed specific changes in the environment. We repeated this process every 2,500 generations, which is the equivalent to one full iteration of Plato, for a total of 60 iterations such that by the end of the experiment all links in the overlay network were set to a faulty status.

Figure 19 plots the maximum fitness achieved by Plato as it evolved target reconfigurations in response to repeated link failures. This plot illustrates the resilience of evolved Plato reconfigurations even as the entire overlay network suffered major failures. For example, this plot shows how Plato was able to rapidly evolve suitable target reconfigurations (approximately within 30 seconds) of each link failure throughout this degenerate scenario. In addition, notice how the maximum fitness value of the generated target overlay networks remains relatively stable at around a fitness value of 400 for the majority of the experiment. This slow decay in fitness values implies that Plato was able to generate solutions at a consistent level of quality even though network conditions were severely deteriorating. A sharp decay in fitness values is evident after 80% of the overlay network links have failed, which occurs after approximately 125,000 generations. Shortly after 90% of links have failed, fitness values plummet to a fitness value of -400 as there are not enough *non-faulty* links available for Plato to maintain connectivity across the set of remote data mirrors.

The plot in Fig. 20 shows the average number of active links in the evolved target reconfigurations. In the initial overlay network design, Plato reduced the number of active links to create a spanning tree and minimize operational costs. In contrast, as link failed, Plato increased the number

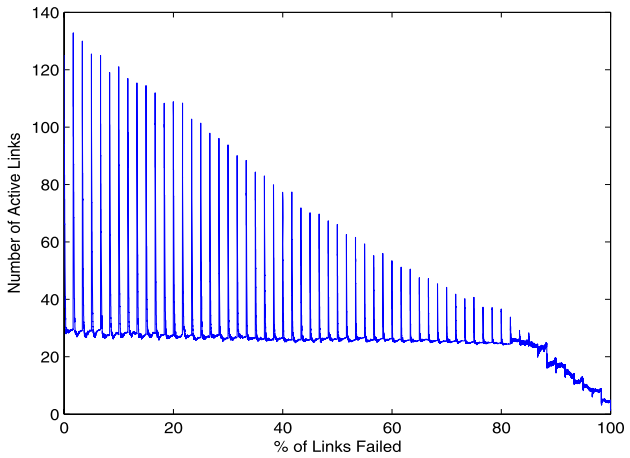

Fig. 20 Average number of active links throughout multiple reconfigurations until complete network failure

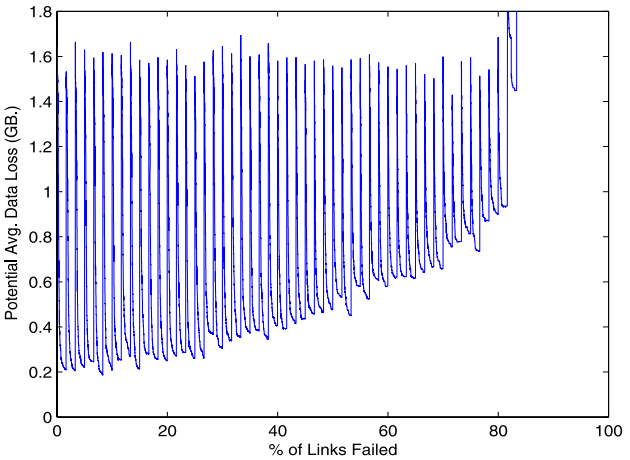

Fig. 21 Average potential data loss throughout multiple reconfigurations until complete network failure

of redundant active links to maximize data reliability while minimizing operational costs throughout each reconfiguration iteration. As Fig. 20 illustrates, throughout the majority of this experiment, Plato generated target reconfigurations where overlay networks comprised approximately 30 active links. Moreover, notice how the starting number of active links in the target reconfigurations is progressively lower during each successive reconfiguration iteration, most likely because there are fewer *non-faulty* links available for Plato to activate. This plot also highlights how Plato attempts to re-establish connectivity across the network of remote data mirrors after slightly more than 90% of the network links have failed. Unfortunately, beyond this point, it is impossible for Plato, or any other system, to generate target reconfigurations that satisfy the main functional requirement of maintaining network connectivity.

Lastly, Fig. 21 plots the average potential data loss in the initial and reconfigured overlay network designs until all links in the network fail. The average potential data loss measures the amount of data, in gigabytes, that could be lost as a result of some failure. This plot demonstrates that Plato repeatedly minimizes the potential average data loss across the network of remote data mirrors. In particular, Plato significantly reduces the average potential data loss across the network throughout the majority of the experiment. Note, however, that the average potential data loss gradually increases as the number of faulty links increases, resulting in Plato having fewer usable links to select from when constructing a target overlay network. Eventually, the average potential data loss reaches its maximum possible value when Plato is unable to build an overlay network that maintains connectivity within the set of remote data mirrors. At this point data is no longer protected against failures.

5 Discussion

The experiments described above indicate that a system such as Plato, which incorporates genetic algorithms into decision-making processes of adaptive and autonomic systems, can be executed online to support dynamic reconfiguration. In terms of execution time, Plato terminated always within 3 minutes or less, and typically converged on a solution within one minute. Moreover, viable solutions were typically found approximately within 30 seconds, well within the practical range for applications such as remote data mirroring. In terms of evaluations performed, Plato typically required between 50,000 and 100,000 evaluations of candidate overlay network designs. Whereas other complementary approaches made use of simulators to assess the effects of candidate reconfigurations [18, 25], Plato's fitness functions are computationally inexpensive calculations. As a result, Plato was able to perform many evaluations in a reasonably short amount of time.

Plato provides several advantages over more traditional approaches for decision-making in self-adaptive and autonomic systems. Specifically, Plato does not require developers to explicitly encode prescriptive reconfigurations strategies to address particular scenarios that may arise at run time. Instead, Plato exploits user-defined fitness functions to *evolve* target reconfigurations in response to changing environmental conditions. For instance, when an active link in the overlay network failed in our reconfiguration experiments, Plato did not explicitly encode how many links to activate, which links to activate, nor which propagation methods to select. Instead, by specifying the general type of acceptable solutions, Plato automatically evolved target reconfigurations that balanced these competing objectives at run time. This approach enables Plato to handle a richer set of reconfiguration scenarios than traditional prescriptive approaches.

Genetic algorithms are susceptible to changes in their configuration parameters and solution encodings. For instance, we experimented with various mutation rates during the design phase to determine which value worked best when applying Plato to the domain of remote data mirroring. Although Plato was able to evolve viable target reconfigurations with modest changes in the mutation rate, higher mutation rates typically caused Plato to require additional computational time to converge upon particular solutions as considerable variation was being introduced into the population each generation. Similarly, the encoding used to represent adaptive systems is extremely important. While more compact representations are possible for encoding a network of remote data mirrors and the propagation methods of each network link, tradeoffs must be made between reducing the search space and altering the probability of different configurations emerging. For example, a more compact genome representation in Plato might encode the operational status and propagation method of each overlay network link as a binary string 3 bits long, thus enumerating each of the possibilities from 0 (a link not used) to 7 (an active link with asynchronous propagation with a 24 hour time bound). Even though this encoding reduces the search space, the probability of deactivating a link now drops from $\frac{1}{2}$ to $\frac{1}{8}$, possibly affecting the quality of solutions evolved by Plato. As a result, it is typically best to begin with default configurations [10] and experiment how the evolutionary algorithm behaves with different parameters.

One potential drawback of Plato is that genetic algorithms are not guaranteed to find optimal, or even viable, solutions [10]. Although this problem did not arise in any of our experimental trials, it is possible for Plato to converge on sub-optimal solutions that are not suitable for particular problems and domains. Genetic-algorithm based approaches tend to be most useful when solution landscapes are vast, complex, and non-linear. As a result, Plato should not be used in autonomic systems where optimal solutions are required. Rather, Plato should be applied when acceptable solutions are viable. Finally, it may be possible to integrate Plato with traditional decision-making approaches. For instance, Plato could be leveraged in the background of a traditional decision-making approach in case a reconfiguration strategy is not available for current system conditions, thereby serving as a backup.

6 Related work

This section reviews related work in utility-based decision-making, remote mirroring, and the application of evolutionary computational techniques to the construction of dynamic overlay networks.

Utility-based decision making. Walsh et al. [33] introduced an architecture for incorporating utility functions as part of the decision-making process of an autonomic system. Utility functions were shown to be effective in handling reconfiguration decisions against multiple objectives. In the context of autonomic computing, utility functions map possible states of an entity into scalar values that quantify the desirability of a configuration as determined by user preferences. Given a utility function, the autonomic system determines the most valuable system state and the means for reaching it. In the approach proposed in [33], a utility calculator repeatedly computes the value that would be obtained from each possible configuration. Despite their advantages, utility functions may suffer from complexity issues as multiple dimensions scale depending on the evaluation method used. In contrast, although genetic algorithms use fitness functions, which are akin to utility functions, the process of natural selection efficiently guides the search process through the solution space.

Automated design of remote mirrors. The design state space of dependable data systems tends to be vast and complex due to the numerous alternatives for each configuration choice. Researchers have developed automated provisioning tools to alleviate the high complexity associated with designing data reliability systems, such as remote mirrors and tape backups [1, 15]. These automated tools rely on formal optimization approaches, such as mathematical solvers, and decomposition techniques to design restricted versions of dependable data systems [16]. These approaches have the advantage of producing optimal data dependability designs based on user's requirements. However, as the number of workloads and the complexity of the data storage solution grows, these formal approaches tend to not scale well [16]. Moreover, designs are produced according to expected usage rates and environmental conditions identified at design time, which may change once the system is deployed. For example, the methods described by Keeton et al. [15] do not address dynamic reconfiguration of the system when actual conditions differ from those assumed at design time.

Genetic algorithms for data replication. Loukopoulos et al. [21] applied genetic algorithms to the problem of file replication in data distribution. Specifically, some files are replicated at multiple sites to reduce the delay experienced by distributed system end-users. The decision as to which files to replicate and where to replicate them is an NP-complete constraint optimization problem [25]. Their initial approach [21] leveraged a genetic algorithm to solve the file replication problem when read and write demands remained static. However, this approach was not applicable when read and write demands continuously changed. Loukopoulos et al. proposed a hybrid genetic algorithm that took as input

a current copy distribution and produced a new file replication distribution using knowledge about the changing environment. This hybrid genetic algorithm is not executed at run time to dynamically solve the file replication problem, but instead incorporates possible dynamic changes into the initial design.

Genetic algorithms for dynamic networks. Genetic algorithms also have been used to design overlay multicast networks for data distribution [6, 22, 32, 34]. These overlay networks must balance the competing goals of diffusing data across the network as efficiently as possible while minimizing expenses. A common approach for integrating various objectives in a genetic algorithm is to use a cost function that linearly combines several objectives as a weighted sum [6, 21, 25]. Although most of these approaches [22, 32, 34] achieved rapid convergence rates while producing overlay networks that satisfied the given constraints, to our knowledge, the methods were not applied at run time to address dynamic changes in the network's environment.

Similarly, Cox et al. [12] applied genetic algorithms to design cost-effective backhaul networks for personal communications services (PCS) traffic. The design of these types of networks is typically complicated by the competing objectives of minimizing operational costs, ensuring network link capacity constraints are satisfied, and ensuring the optimal type of link is selected when connecting nodes and hubs. In their approach, Cox et al. represented information about hub locations and their interconnections as a two-part chromosome in a GA. Each part of the chromosome is then evolved in parallel through the processes of crossover and mutation. Although the GA representation and implementation by Cox et al. [12] differs from the one used by Plato, their genetic algorithm was able to produce high quality solutions within 30 to 35 seconds. While Cox et al. did not leverage run-time system information with their GA to dynamically reconfigure networks, their results indirectly reaffirm our conclusions that genetic algorithms can be applied in real-time to generate viable designs.

Genetic algorithms for reconfiguration. Montana et al. [25] developed a genetic algorithm to reconfigure the topology and link capacities of an operational network in response to its operating conditions. Their experimental results confirm that for small networks, comprising fewer than 20 nodes and 5 channels, the optimization algorithm would support online adaptation. However, due to computational expenses, online adaptation was not supported for larger networks. Specifically, in their approach [25], Montana et al. made repeated use of a network simulator, ns/2 [23], to evaluate each individual by accurately modeling the effects of topology reconfigurations at run time. Plato, on the other hand, makes use of computationally inexpensive fitness functions to significantly reduce the time required for convergence. Furthermore, in contrast with Plato, the approach by Montana et al. [25] did not rescale the relative importance of fitness functions to design different types of networks in response to changing system and environmental conditions. Nonetheless, given the recent advances in computing technology, it would be worthwhile to determine whether their approach would now support the online adaptation of larger networks.

7 Conclusion

Having examined the evolution of target system reconfigurations by Plato, we make several observations. First, it is possible to integrate genetic algorithms with decision-making processes of autonomic systems to dynamically evolve reconfigurations that balance competing objectives at run time. Second, decision-making processes that leverage genetic algorithms do not require prescriptive rules to address particular scenarios warranting reconfiguration. Instead, an evolutionary computation-based approach such as Plato is able to evolve target reconfigurations to address situations that were not anticipated at design time, by incorporating system and environmental monitoring information with a genetic algorithm. Third, we were able to show how Plato is resilient in evolving suitable target reconfigurations against severely deteriorating environmental conditions. Lastly, different types of target system reconfigurations can be evolved in response to changing requirements and environmental conditions by rescaling individual fitness functions.

Future directions for this work include applying Plato to other industrial-scale problems and evaluating its overall performance. Furthermore, we are interested in exploring how the quality of solutions produced by Plato compare with those generated by other traditional decision-making approaches. Lastly, we are also interested in exploring how to leverage software engineering techniques, such as formal verification and aspect-orientation, to ensure target reconfigurations satisfy system invariants and local properties [38].

Acknowledgements We gratefully acknowledge John Wilkes for introducing us to this application problem, asking insightful questions, and providing valuable feedback for this work. This work has been supported in part by NSF grants CCF-0541131, IIP-0700329, CCF-0750787,CCF-0820220, CNS-0854931, CNS-0751155, CNS-0915855, Army Research Office grant W911NF-08-1-0495, Ford Motor Company, and a Quality Fund Program grant from Michigan State University.

References

1. Alvarez, G.A., Borowsky, E., Go, S., Romer, T.H., Becker-Szendy, R., Golding, R., Merchant, A., Spasojevic, M., Veitch, A., Wilkes, J.: Minerva: an automated resource provisioning tool for large-scale storage systems. ACM Trans. Comput. Syst. **19**(4), 483–518 (2001)

2. Andersen, D., Balakrishnan, H., Kaashoek, F., Morris, R.: Resilient overlay networks. ACM SIGOPS Oper. Syst. Rev. **5**, 131–145 (2001)
3. Cheng, S.-W., Garlan, D., Schmerl, B.: Architecture-based self-adaptation in the presence of multiple objectives. In: Proceedings of the 2006 International Workshop on Self-Adaptation and Self-Managing Systems, pp. 2–8, Shanghai, China (2006). ACM
4. Deb, K.: Multi-Objective Optimization Using Evolutionary Algorithms. Wiley, New York (2001)
5. EagleRock2001: Online survey results: 2001 cost of downtime. Eagle Rock Alliance Ltd. http://contingencyplanningresearch.com/2001Survey.pdf, August 2001
6. Fabregat, R., Donoso, Y., Baran, B., Solano, F., Marzo, J.L.: Multi-objective optimization scheme for multicast flows: a survey, a model and a MOEA solution. In: Proceedings of the 3rd International IFIP/ACM Latin American Conference on Networking, pp. 73–86, New York, NY, USA (2005). ACM
7. Garlan, D., Cheng, S.-W., Huang, A.-C., Schmerl, B., Steenkiste, P.: Rainbow: architecture-based self-adaptation with reusable infrastructure. Computer **37**(10), 46–54 (2004)
8. Goldsby, H.J., Cheng, B.H.C.: Automatically generating behavioral models of adaptive systems to address uncertainty. In: Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems, pp. 568–583. Springer, Berlin (2008). (Selected as one of the Best Papers in the Conference)
9. Goldsby, H.J., Cheng, B.H.C., McKinley, P.K., Knoester, D.B., Ofria, C.A.: Digital evolution of behavioral models for autonomic systems. In: Proceedings of the Fifth IEEE International Conference on Autonomic Computing, pp. 87–96 (Best Paper Award), Chicago, Illinois (2008). IEEE Computer Society
10. Holland, J.H.: Adaptation in Natural and Artificial Systems. MIT, Cambridge (1992)
11. Ji, M., Veitch, A., Wilkes, J.: Seneca: Remote mirroring done write. In: USENIX 2003 Annual Technical Conference, pp. 253–268, Berkeley, CA, USA, June 2003. USENIX Association
12. Cox, L.A. Jr, Davis, L., Lu, L.L., Orvosh, D., Sun, X., Sirovica, D.: Reducing costs of backhaul networks for pcs networks using genetic algorithms. J. Heuristics **2**(3), 201–216 (1996)
13. Kaiser, G., Gross, P., Kc, G., Parekh, J.: An approach to autonomizing legacy systems. In: Proceedings of the First Workshop on Self-Healing, Adaptive, and Self-MANaged Systems (2002)
14. Keeton, K., Beyer, D., Brau, E., Merchant, A.: On the road to recovery: Restoring data after disasters. SIGOPS Oper. Syst. Rev. **40**(4), 235–248 (2006)
15. Keeton, K., Santos, C., Beyer, D., Chase, J., Wilkes, J.: Designing for disasters. In: Proceedings of the 3rd USENIX Conference on File and Storage Technologies, pp. 59–62, Berkeley, CA, USA (2004). USENIX Association
16. Keeton, K., Merchant, A.: Challenges in managing dependable data systems. SIGMETRICS Perform. Eval. Rev. **33**(4), 4–10 (2006)
17. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. Computer **36**(1), 41–50 (2003)
18. Khanna, R., Liu, H., Chen, H.-H.: Dynamic optimization of secure mobile sensor networks: a genetic algorithm. In: Proceedings of the IEEE International Conference on Communications, pp. 3413–3418, June 2007
19. Knoester, D.B., Ramirez, A.J., Cheng, B.H.C., McKinley, P.K.: Evolution of robust data distribution among digital organisms. In Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation (GECCO '09), pp. 137–144 (Nominated for Best Paper), Montreal, Canada, July 2009
20. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection (Complex Adaptive Systems). MIT, Cambridge (1992)
21. Loukopoulos, T., Ahmad, I.: Static and adaptive distributed data replication using genetic algorithms. J. Parallel Distributed Comput. **64**(11), 1270–1285 (2004)
22. Lu, J., Cheng, W.: A genetic-algorithm-based routing optimization scheme for overlay network. In: Proceedings of the 3rd International Conference on Natural Computation, pp. 421–425, Washington, DC, USA, 2007. IEEE Computer Society Press
23. McCanne, S., Floyd, S.: The lbnl network simulator. Software online: http://www.isi.edu/nsnam (1997)
24. McKinley, P.K., Sadjadi, S.M., Kasten, E.P., Cheng, B.H.C., Composing adaptive software. Computer **37**(7), 56–64 (2004)
25. Montana, D., Hussain, T., Saxena, T.: Adaptive reconfiguration of data networks using genetic algorithms. In: Proceedings of the Genetic and Evolutionary Computation Conference, pp. 1141–1149, San Francisco, CA, USA (2002)
26. Newman, H.B., Legrand, I.C., Galvez, P., Voicu, R., Cistoiu, C.: MonALISA: A distributed monitoring service architecture. In: Proceedings of the 2003 Conference for Computing in High Energy and Nuclear Physics, March 2003
27. Ramirez, A.J.: Design patterns for developing dynamically adaptive systems. Master's thesis, Michigan State University, East Lansing, MI 48823 (2008)
28. Ramirez, A.J., Knoester, D.B., Cheng, B.H.C., McKinley, P.K.: Applying genetic algorithms to decision making in autonomic computing systems. In: Proceedings of the Sixth International Conference on Autonomic Computing (ICAC'09), pp. 97–106 (Best Paper Award), Barcelona, Spain, June 2009
29. Sadjadi, S.M., McKinley, P.K.: ACT: an adaptive CORBA template to support unanticipated adaptation. In: Proceedings of the IEEE International Conference on Distributed Computing Systems, pp. 74–83 (2004)
30. SEC2002: Summary of "lessons learned" from events of September 11 and implications for business continuity. http://www.sec.gov/divisions/marketreg/lessonslearned.htm, February 2002
31. Tang, C., McKinley, P.K.: A distributed approach to topology-aware overlay path monitoring. In: Proceedings of the 24th International Conference on Distributed Computing, pp. 122–131, Tokyo, Japan (2004). IEEE Computer Society
32. Tseng, S.-Y., Huang, Y.-M., Lin, C.-C.: Genetic algorithm for delay- and degree-constrained multimedia broadcasting on overlay networks. Comput. Commun. **29**(17), 3625–3632 (2006)
33. Walsh, W.E., Tesauro, G., Kephart, J.O., Das, R.: Utility functions in autonomic systems. In: Proceedings of the First IEEE International Conference on Autonomic Computing, pp. 70–77, New York, NY, USA (2004). IEEE Computer Society
34. Wang, D., Gan, J., Wang, D.: Heuristic genetic algorithm for multicast overlay network link selection. In: Proceedings of the Second International Conference on Genetic and Evolutionary Computing, pp. 38–41, September 2008
35. Witty, R., Scott, D.: Disaster recovery plans and systems are essential. Technical Report FT-14-5021, Gartner Research, September 2001
36. Yang, Z., Cheng, B.H.C., Stirewalt, R.E.K., Sowell, J., Sadjadi, S.M., McKinley, P.K.: An aspect-oriented approach to dynamic adaptation. In: Proceedings of the First Workshop on Self-Healing Systems, pp. 85–92, New York, NY, USA (2002). ACM
37. Zhang, J., Cheng, B.H.C.: Model-based development of dynamically adaptive software. In: Proceedings of the 28th International Conference on Software Engineering, pp. 371–380, New York, NY, USA (2006). ACM (Distinguished Paper Award)
38. Zhang, J., Goldsby, H.J., Cheng, B.H.C.: Modular verification of dynamically adaptive systems. In: Proceedings of the Eighth International Conference on Aspect-Oriented Software Development (2009)

Andres J. Ramirez is doctoral student in the Department of Computer Science and Engineering at Michigan State University. He received his B.S. degree from the University of Illinois at Urbana-Champaign in 2006, and his M.S. degree from Michigan State University in 2008, all in Computer Science. His research interests include adaptive and autonomic systems, model-driven engineering, distributed computing systems, and evolutionary computation. He is a member of the IEEE and the ACM. He may be contacted at ramir105@cse.msu.edu; www.cse.msu.edu/~ramir105.

David B. Knoester is a doctoral student in the Department of Computer Science and Engineering at Michigan State University. His research interests include digital and biological evolution, self-organizing systems, and distributed computing systems. Knoester received an MS in computer science from Michigan State University. He is a member of the IEEE and the ACM. Contact him at dk@cse.msu.edu

Betty H.C. Cheng is a professor in the Department of Computer Science and Engineering at Michigan State University. Her research and teaching interests include automated software engineering, requirements engineering, embedded systems development, assurance patterns, adaptive systems, model-driven engineering, and applications of evolutionary computing. She collaborates with industrial partners for both her class projects and research in order to facilitate technology exchange between academia and industry. She was awarded a NASA/JPL Faculty Fellowship in 1993 to investigate the use of new software engineering techniques for a portion of the shuttle software. In 1998, she spent her sabbatical working with the Motorola Software Labs investigating automated analysis techniques of specifications of telecommunication systems. Her research has been funded by NSF, ONR, DARPA, NASA, AFRL, Army, and numerous industrial organizations. She serves on the editorial boards for Requirements Engineering Journal, and Software and Systems Modeling; she recently completed a term on the editorial board for IEEE Transactions on Software Engineering. Each year, she serves on numerous program and organizational committees for international conferences and workshops, including IEEE International Conference on Software Engineering (ICSE), IEEE Requirements Engineering Conference (RE), and IEEE MODELS.
She received her BS from Northwestern University in 1985 and her MS and Ph.D. from the University of Illinois-Urbana Champaign in 1987 and 1990, respectively, all in computer science. She may be reached at the Department of Computer Science and Engineering, Michigan State Univ., 3115 Engineering Building, East Lansing, MI 48824; chengb@cse.msu.edu; www.cse.msu.edu/~chengb.

Philip K. McKinley received the B.S. degree in mathematics and computer science from Iowa State University in 1982, the M.S. degree in computer science from Purdue University in 1983, and the Ph.D. degree in computer science from the University of Illinois at Urbana-Champaign in 1989. Dr. McKinley is currently a Professor of Computer Science and Engineering at Michigan State University, where he has been on the faculty since 1990. His current research interests include autonomic computing, self-adaptive software, evolutionary computation, artificial life, and swarm robotics. He is a member of the IEEE and ACM. He may be contacted electronically at mckinley@cse.msu.edu.