Information Reuse and Stochastic Search: Managing Uncertainty in Self-* Systems

CODY KINNEER, DAVID GARLAN, and CLAIRE LE GOUES, School of Computer Science, Carnegie Mellon University

Many software systems operate in environments of change and uncertainty. Techniques for self-adaptation allow these systems to automatically respond to environmental changes, yet they do not handle changes to the adaptive system itself, such as the addition or removal of adaptation tactics. Instead, changes in a self-adaptive system often require a human planner to redo an expensive planning process to allow the system to continue satisfying its quality requirements under different conditions; automated techniques must replan from scratch. We propose to address this problem by reusing prior planning knowledge to adapt to unexpected situations. We present a planner based on genetic programming that reuses existing plans and evaluate this planner on two case-study systems: a cloud-based web server and a team of autonomous aircraft. While reusing material in genetic algorithms has been recently applied successfully in the area of automated program repair, we find that naively reusing existing plans for self-* planning can actually result in a utility loss. Furthermore, we propose a series of techniques to lower the costs of reuse, allowing genetic techniques to leverage existing information to improve utility when replanning for unexpected changes, and we find that coarsely shaped search-spaces present profitable opportunities for reuse.

CCS Concepts: • **Computing methodologies** → *Control methods*; • **Computer systems organization** → *Cloud computing*; *Dependable and fault-tolerant systems and networks*; • **Software and its engineering** → *Software evolution*; *Search-based software engineering*;

Additional Key Words and Phrases: Plan reuse, self-* systems, planning, uncertainty, genetic programming, cloud services

ACM Reference format:
Cody Kinneer, David Garlan, and Claire Le Goues. 2021. Information Reuse and Stochastic Search: Managing Uncertainty in Self-* Systems. *ACM Trans. Auton. Adapt. Syst.* 15, 1, Article 3 (February 2021), 34 pages.
https://doi.org/10.1145/3440119

1 INTRODUCTION

Self-* systems lower the costs of operating in complex environments of change and uncertainty by autonomously adapting to change in pursuit of their quality objectives. One way these systems

This work was supported by the National Science Foundation (CCF-1618220), the Office of Naval Research (N00014172899), the NSA (H9823018D0008), and the Carnegie Mellon CyLab Security and Privacy Institute. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsoring agencies.

Authors' addresses: C. Kinneer, D. Garlan, and C. Le Goues, School of Computer Science, Carnegie Mellon University, 5000 Forbes Ave, Pittsburgh, PA 15213; emails: {ckinneer, garlan, clegoues}@cs.cmu.edu.

1556-4665/2021/02-ART3 $15.00
https://doi.org/10.1145/3440119

self-adjust is by making runtime adjustments according to an adaptation strategy or *plan*. Humans can proactively plan for various situations by hand at design time [10]. This is a form of *offline planning*, requiring a painstaking consideration of the full range of possible runtime scenarios the system may encounter. Automated techniques known as *online planners* seek to reduce planning costs by synthesizing plans at runtime [24, 40, 52, 57].

While these systems can quickly respond to the changing conditions that they were designed for, they often struggle to handle unforeseen adaptation scenarios [12]. Such "unknown unknowns" realistically include, but are not limited to (1) changes in the cost or effects of available adaptation tactics (e.g., a provider changes the pricing schedule for cloud resources), (2) changes in available adaptation tactics or options (e.g., a new type of hardware or server comes to market), or (3) unexpected changes in environmental conditions or use cases (e.g., unexpectedly surging popularity of a service, such as via the Slashdot effect [51]). Even human generated plans [10] cannot handle this challenge, requiring expensive replanning post design time in the face of unanticipated changes. As self-* systems become increasingly large, complex, and interconnected, this cost will only increase.

One potential way to respond to unanticipated adaptation needs is to automatically reuse or adapt prior knowledge to new situations. Indeed, research in artificial intelligence [1, 54] and case-based reasoning [19, 35, 42] has explored the potential of plan reuse, using knowledge contained in previously-created plans to speed the synthesis of new plans in response to unanticipated changes. However, the self-* context poses unresolved domain-specific challenges, since these systems must autonomously respond to uncertainty from a number of sources throughout the adaptation cycle.

We have previously argued [11] that this is a fruitful potential domain for the application of stochastic algorithms to self-* systems. Stochastic search techniques have been shown to be well suited for similar problems [3, 6, 14, 23, 47, 48, 50] and are effective at handling large search spaces. However, these approaches do not address the challenge of replanning for new, explicitly unforeseen contexts post-design time, which we propose to address by reusing prior plans. Intuitively, genetic algorithms should be expected to benefit from reused information, since they operate by balancing between exploring new solutions and exploiting existing solutions, in effect reusing information from previous generations. This observation has been successfully applied in other domains, such as automated program repair [21]. We investigate the extent to which reusing existing plans in self-* planning can result in an improvement in fulfilling the system's quality objectives. Surprisingly, we find that reusing plans directly is less effective than replanning from scratch. We further propose a series of techniques to make reusing existing plans more efficient, ultimately obtaining a planner that can reuse prior plans to improve the system's quality objectives.

We present a self-adaptive systems planner, built on genetic programming, that responds to unforeseen adaptation scenarios by reusing and building upon prior knowledge. We represent individuals as candidate plans, evaluating individual fitness by running them against a simulated system. Our approach explicitly takes into account the probability that individual tactics may fail, and supports reasoning about tactic latency and planning time.

Our planner reuses past information by initializing the population with individuals based on an existing plan. We present a series of techniques to support adapting to unexpected changes at runtime by lowering the costs of plan reuse during evolution and apply our approach to two case-study systems with large search spaces and different planning assumptions: a cloud-based web server and a team of autonomous aircraft. This enables an empirical study investigating the effectiveness of plan reuse for several indicative change scenarios.

The previous version of this work [29] introduced the following key contributions:

- An investigation into plan reuse in genetic algorithm planning, finding, counter-intuitively, that naïve reuse can lower planning utility.

- A set of techniques for lowering the cost of plan reuse, resulting in a self-* planner that can reuse past information to respond to unforeseen changes more effectively.
- As a sanity check, an empirical comparison of our genetic programming planner to a PRISM Markov decision processes (MDP) planner [43] that shows the genetic programming planner can produce near-optimal plans (0.05% error in the single objective case and 9.4% in the multi-objective case).
- An investigation into the time, quality, and population diversity produced by planning with reuse when adapting to unforeseen scenarios compared to planning from scratch. We find that while the improvement is often slight, effective plan reuse can result in a utility improvement.
- Results that show that the objectives emphasized in a multi-objective planner's starting plan can influence the quality and character of the planner's output.

We extend this work with the following key contributions:

- Additional content from the evaluation using the Omnet case study, including greater detail about the parameter sweep, measuring population diversity using a structural measure, and providing data from two new unexpected change scenarios.
- An evaluation using a new case-study system with a distinct domain, planning model, and assumptions, the DART team of autonomous aircraft [39], finding that coarsely shaped search spaces present opportunities for significant improvement in planning effectiveness.
- An empirical comparison of genetic programming planning and plan reuse to an exhaustive approach, a PRISM MDP planner in this new domain.
- A discussion of lessons learned from each case study, including implications for applying plan reuse in other self-* systems.

The rest of the article is as follows. Section 2 outlines necessary background. We next describe the running example we use to both illustrate and evaluate our technique (Section 3). Section 4 details our genetic programming self-* planning approach. Section 5 describes our evaluation; Section 6 outlines related work. Section 7 concludes and offers discussion.

2 BACKGROUND

This section overviews self-* planning (Section 2.1) and genetic programming (Section 2.2), focusing on the background required to understand our approach.

2.1 Self-* Planners

Self-* systems typically consist of two subsystems, a *managed* system and a *managing* system. Many self-* systems follow the well-known five-component MAPE-K architecture [26], shown in Figure 1. We focus on the planning (P) component, which produces adaptation strategies (or plans) consisting of tactics, ordered to achieve a particular goal. For our purposes, tactics are architectural changes the system can perform to respond to changes, e.g., "turn off a server at location A." While multiple planning languages exist for the self-* context [33, 52], our approach is closest to Stitch [10]. An *online* planner [52] generates a plan at runtime, which can adapt quickly at a potential cost to optimality. An *offline* planner [10] precomputes plans to handle common cases and then chooses between them at runtime. This allows for a fast, correct response to known or predicted situations, but cannot handle unanticipated adaptation needs. Online planners must adapt by performing an expensive replanning operation from scratch. For evaluation, we compare to a previous hybrid online/offline approach [43], that relies in part on MDPs, formal models that can

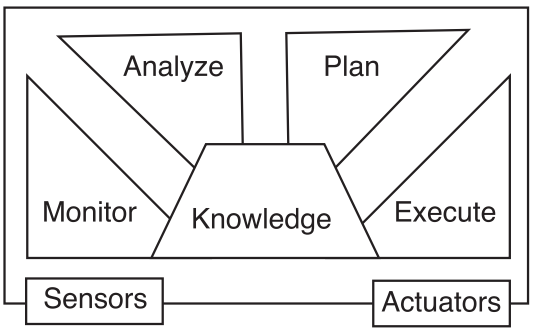


Fig. 1. MAPE-K Loop for self-* systems.

explicitly capture probabilistic behavior. Model checkers such as PRISM [33] can use exhaustive search to compute an optimal sequence of tactics to maximize one or more system objective (e.g., profit) for systems formalized as MDPs.

2.2 Genetic Programming

Genetic programming (GP) [32] is a stochastic technique modeled on the principles of biological evolution. GP is well suited for problems with poorly understood search landscapes and those for which approximate solutions are suitable [46]. Note that these conditions apply to our problem: interacting tactics or quality attributes render the search landscape complex; large search spaces may preclude the need for (or feasibility of) computing optimal plans; sub-optimal plans are often acceptable in real systems. Indeed, genetic algorithms have been successfully applied to self-* systems [6, 14, 47], although using them to explicitly leverage prior knowledge during replanning has not been investigated in self-* systems to the best of our knowledge.

At a high level, a GP evolves a population of candidate programs toward a goal over successive generations. A GP represents and manipulates individual candidate solutions as trees, which are modified and recombined using computational analogues of biological *mutation*, *crossover*, and *selection*. Mutation randomly modifies one or more subtrees in an individual, supporting search space *exploration*. Crossover randomly combines parent individuals to produce new children, supporting *exploitation* of partial solutions. A tree-based representation admits the enforcement of a type system over nodes [38], limiting exploration of some types of invalid solutions.

A problem-specific *utility*, *objective*, or *fitness function* measures how well a candidate solution satisfies the search objectives. *Fitness* typically informs the probability with which an individual is *selected* from one generation to the next for continued iteration and can inform the search stopping criterion (if an optimal value or suitable threshold is known). In contexts with multiple fitness objectives, a multi-objective search can produce a set of individuals, or *Pareto frontier*, representing the best possible tradeoffs between several objectives. We use SPEA2 [56] to implement a multi-objective search, which selects a fixed quantity of non-dominated individuals to create the next generation and has been shown to produce high-quality Pareto fronts. Our approach could also be implemented with other multi-objective evolutionary algorithms such as NSGA-II [13], although since the solution quality between the two has been shown to be similar for problems with small dimensionality [56], we do not expect the choice to have a large impact on the results. Note that while heuristic approaches like GP can handle large search spaces, finding the optimal solution is not guaranteed.

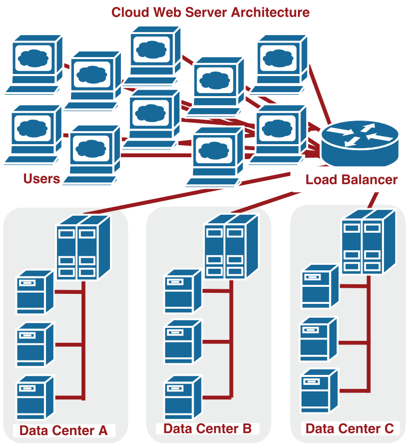


Fig. 2. Cloud web server architecture.

3 RUNNING SCENARIO

We first present one of our two case-study systems to serve as a running example: Omnet, a cloud-based web server adapted from prior work [43].

3.1 Scenario

Figure 2 shows a cloud-based self-adaptive website with an N-tiered architecture. User requests to the system are distributed by a load balancer to data centers and then to individual servers. Servers process requests and then return a response to the user. Each data center has servers of a different type, i.e., data center A will have servers of type A, each type with different attributes. In general, the more users a server can handle, the higher its cost. The website can also serve ads to increase profit, slowing response time.

3.2 Quality Objectives

The system goal is to earn profit while maintaining user satisfaction. We consider three interrelated quality objectives: (1) System *profit* as revenue generated by current users, minus operating cost (corresponding to the number and cost of the servers); (2) *User latency*, or the number of users that have to wait for a system response (related to the number and quality of the running servers); and (3) *User-perceived quality*, the percentage of users viewing ads. These goals are in tension. For example, while the system uses ads for revenue, they increase latency. The system can remove ads using a throttling mechanism to improve user experience and server load, while decreasing profit.

The profit P of the case-study system at a particular state is given by the following equation:

$$P = R_O \cdot x_O + R_M \cdot x_M - \sum_{i=1}^{n} (C_i \cdot S_i),$$

where R_O and R_M is the revenue generated from requests that are unthrottled and throttled respectively, and x_O and x_M denote the number of requests that the system handles, unthrottled and throttled, respectively. Throttled requests are those requests where optional content such as ads are not shown to the user to reduce server load. The summation provides the cost of operation, which is subtracted from the revenue to yield profit. The summations add the costs at each data center i, which is given by the operating costs of the server type at the data center C_i, multiplied by the number of servers that have been started at that data center, denoted by S_i.

Table 1. Left: The Tactic Failure Rates and Tactic Execution Times for the Six Tactics in the Web Server Scenario. Right: The Starting Values for Data Center Attributes

Tactic	Fail Rate (%)	Time (seconds)
StartServer	10	120
ShutdownServer	10	30
IncreaseDimmer	5	1
DecreaseDimmer	5	1
IncreaseTraffic	5	5
DecreaseTraffic	1	5

Data Center Attributes	Starting Value
Running Servers	1
Dimmer Value	0%
Traffic Value	4

When evaluating latency, rather than simulating response times, we report the number of users who experience delays due to the system being overloaded, which is given by the difference between the number of requests the system can support in its configuration versus the total number of incoming requests, denoted x_T,

$$L = x_T - x_O - x_M.$$

Users are allocated between data centers proportionally according to a traffic value parameter $t_i \in [1, 5]$ that is set by an adaptation tactic. The number of unthrottled and throttled requests is determined by the total number of requests, the capacity of each server type for full requests O_i, throttled requests M_i, the number of running servers, and the dimmer value $d_i \in [1, 5]$, which is set by an adaptation tactic. For each data center, requests are allocated to either be unthrottled or throttled proportionally to the dimmer value setting.

3.3 Adaptation Tactics

Multiple tactics can adjust the system in pursuit of its quality objectives. These tactics can turn on and off different types of servers, up to a maximum of five per type. Each server type has an associated operating cost per second and a number of users it can support per second, with or without ads. The system's load balancer distributes requests among data centers according to a traffic value; there are five traffic levels per data center, and traffic is distributed proportionally. The system can modify *dimmer* settings on each server type, which controls the percentage of users who receive ads (using a brownout mechanism [31] on a per-data center basis). This tactic allows the system to reduce demand by decreasing the amount of content that needs to be served, at the cost of reducing the system's advertising revenue. The dimmer level can be changed by 25% increments. At runtime, each of these adaptation tactics may fail. The failure rates for the adaptation tactics are provided in the left of Table 1, and starting values for the number of running servers, dimmer value, and traffic value, are are provided on the right. Starting and shutting down servers fails 10% of the time, modifying the dimmer level and increasing the traffic level fails 5% of the time, and decreasing the traffic level fails 1% of the time. Additionally, each adaptation tactic requires time before its effects are felt. For example, a server requires time to boot up and initialize its state before being available to server requests. The tactic execution times used in the case study are shown on the left side of Table 1.

3.4 Post-design-time Adaptation

Although synthetic, this case study illustrates a number of ways that a self-* adaptation problem can change post-design. Quality priorities may change, e.g., the system owner might sell it to a charitable organization that cares more about user satisfaction than profit. The effects of existing tactics may change, e.g., the cost of adding a new server may increase or decrease based on a cloud service provider's fee schedule. New tactics may become available, via new data centers, server types, or even hardware. The use case or environment may also unexpectedly change. In the baseline scenario, the system starts with one running server in each of the three data centers and 1,000 incoming requests. The considered change scenarios are as follows:

- **Increased Costs.** All server operating costs increase uniformly by a factor of 100, a *system-wide change.*
- **Failing Data Center.** The probability of `StartServer C` failing increases to 100%, a change in the *effect of an existing tactic.*
- **Request Spike.** The system experiences a major spike in traffic, an *environmental* change.
- **New Data Center.** The system gains access to a new server location. This location (D) contains servers that are strictly less efficient than those at location A (i.e., they have the same operating cost, but lower capacity) but would be useful if there were more requests than could be served by location A. This change is an addition of a *new tactic.*
- **Request Spike + New Data Center.** This adaptation scenario is a combination of the `Request Spike` and `New Data Center` scenarios. This corresponds primarily to an *environmental* change, along with the addition of a *new tactic.*
- **Network Unreliability** The failure probability for all tactics increases to 67%, a change in the *effect of an existing tactic.*

To the best of our knowledge, existing self-* planning technology must always replan from scratch in the face of such adaptation changes.

4 APPROACH

We present a planner that reuses previously known information using GP to efficiently produce nearly optimal results in a large, uncertain search space in response to unforeseen adaptation scenarios. Our approach reuses past knowledge by seeding the starting population with prior plans. These plans satisfied the system's objectives in the past but are currently sub-optimal due to "unknown unknowns," unexpected changes to the system or its environment that the past plans did not address. After an unexpected change occurs, the system model must be updated to reflect the new behavior after the unexpected change, and this update triggers adaptation. The mechanism for synchronizing the system model with the actual world is outside the scope of this article; this may be done manually (likely with less effort than replanning), or automatically [27, 55]. The approach is agnostic to the representation of the system model and relevant changes, as long as the provided model can be used to evaluate the utility of candidate plans. Section 4.4 explains how our approach reuses prior plans, while Sections 4.1–4.3 provide the necessary technical details on the GP implementation. We explain our approach in terms of the running example introduced in Section 3. Section 5.2.1 explains how we modify the approach for the DART case study.

Algorithm 1 shows how the GP planner works at a high level. First, on line 1, a population of candidate solutions called individuals are initialized. The pop_size parameter determines how many individuals will be in the population. The while loop on line 2 iteratively performs reproduction on the population, resulting in a new population of individuals. This is repeated based on the num_generations parameter. Finally, after the designated number of generations, the individual in the population with the highest fitness or utility is returned. The scratch_ratio, trimmer, and kill_ratio parameters improve the efficiency of plan reuse and are explained in Section 4.4.

A new GP application is defined by how individuals are represented (Section 4.1), how they are manipulated during reproduction through mutation and crossover (Section 4.2), and how the fitness of candidate solutions is calculated (Section 4.3).

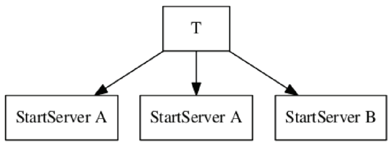
$\langle plan \rangle$ ::= '(' $\langle operator \rangle$ ')' | '(' $\langle tactic \rangle$ ')'

$\langle operator \rangle$ ::= 'F' $\langle int \rangle$ $\langle plan \rangle$ (For loop)
| 'T' $\langle plan \rangle$ $\langle plan \rangle$ $\langle plan \rangle$ (Try-catch)
| ';' $\langle plan \rangle$ $\langle plan \rangle$ (Sequence)

$\langle tactic \rangle$ ::= '`StartServer`' $\langle srv \rangle$ | '`ShutdownServer`' $\langle srv \rangle$
| '`IncreaseTraffic`' $\langle srv \rangle$ | '`DecreaseTraffic`' $\langle srv \rangle$
| '`IncreaseDimmer`' $\langle srv \rangle$ | '`DecreaseDimmer`' $\langle srv \rangle$

Fig. 3. Grammar for specifying plans for the Omnet running example. Servers (*srv*) can be of types A, B, C, or D; For loops can iterate up to 10 times.

```
( T (StartServer A) (StartServer A) (StartServer B) )
```



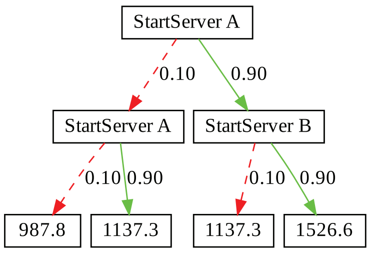


Fig. 4. Top: An example plan. Bottom: This plan's system state tree. Dashed red arrows denote tactic failure, while solid green denotes success.

ALGORITHM 1: Genetic programming planner and reuse enabling approaches parameters.

```
1: p = initialize(pop_size, scratch_ratio, trimmer)
2: while i < num_generations do
3:     p = reproduction(p, kill_ratio)
4: end while
       return best_ind(p)
```

4.1 Representation

Individuals in the population are plans represented as trees. Figure 3 gives a Backus-Naur grammar for our plans. Each plan consists of either (a) one of six available *tactics* (described in Section 3) or (b) one of three *operators* containing subplans. The `for` operator repeats the given subplan for 2–10 iterations; the `sequence` operator consecutively performs two subplans. The `try-catch` operator tries the first subplan. If the last tactic in that subplan fails, then it executes the second subplan; otherwise, it executes the third subplan. The example plan at the top of Figure 4 uses a `try-catch` operator, first attempting to start a new server at data center A. If successful, then it attempts to start a server at data center B; if not, it retries the `StartServer A` tactic.

This planning language is a simplified variant of other languages such as Stitch [10]. Unlike Stitch, our language does not consider plan applicability (guards that test state to determine when a plan can be used). We assume that applicability will be determined by choosing the plan with the highest expected utility, making explicit guards in the planning language unnecessary. We leave investigating the use of applicability guards for reducing the fitness evaluation time to future work. Note that any plan expressible in our language could be expressed with only the `try-catch` operator, and that our language can represent any PRISM MDP [33] plan (or policy) as a tree of `try-catch` operators with depth 2^h, where h is the planning horizon.

4.2 Mutation and Crossover

Mutation may either replace a randomly selected subtree with another randomly generated subtree, or copy an individual unmodified to the next generation. The distribution between these choices is a tunable parameter. Mutation imposes both size and type limitations on generated subtrees, which can range from a single tactic to a tree of depth ten. The crossover operator [53] selects a subtree in each of two parent plans (selected via tournament selection [32]) and swaps them to create two new plans. We enforce syntax rules on both operators (e.g., requiring swapped or generated nodes to have the correct number of children of the correct type). However, it is still possible for the planner to generate plans that lead the system to an invalid state, e.g., a plan that tries to add more servers than are available is syntactically correct, but invalid. We do not prevent this behavior, instead penalizing such plans, to allow the search to break out of local optima.

4.3 Fitness

We evaluate candidate fitness by simulating the plan to measure the expected utility of the resulting system. Since utility may differ between applications, the fitness function is application specific. Because tactics might fail, we must combine multiple eventualities. Thus, conceptually, fitness is computed via a depth-first search of all possible states that a system might reach given a plan, captured in a *system state tree*. Tree nodes represent possible system states; connecting edges represent tactic application attempts, labeled by their probability (the tactic success/failure probability). Every path from the root (the initial system state) to a leaf represents a possible plan outcome. Overall plan fitness is the weighted average of all possible paths through the state tree. Path fitness is the quality of the leaf node system state, measured as one or more of *profit*, *latency*, and *user perceived quality* (Section 3). Each final system state contributes to overall plan fitness, weighted by the probability that that state is reached, which is the product of the edge probabilities from the root to the final system state. In this example, the utility values at the leaf system states are calculated according to the profit equation in Section 3.2, with an unthrottled revenue of 3 and throttled revenue of 1, for 1,000 incoming requests.

To illustrate, Figure 4 shows a plan and its corresponding state tree. Leaf nodes are labeled with their state fitness (profit, in this example); edges with their probability. Left transitions correspond to tactic failure; right-transitions, tactic success. Following the right-hand transitions shows that if all tactics succeed, profit will be 1526.6, with an 81% probability. Following the left-hand transitions shows the expected system state if all tactics fail (1% probability). The weighted sum over all paths (overall fitness) is 1451.14.

The simulator takes into account planning time and tactic latency [41]. Each leaf in the state tree represents a timeline of events (parent tactics succeeding or failing). This timeline is simulated to obtain the utility accrued while the plan was executing, as well as the utility state of the system after the plan terminates. To support reasoning about the opportunity cost of planning time, the fitness function takes as input a *window size* parameter that specifies how long the system is expected to continue accruing the utility resulting from the provided plan. If the system will remain in a state for a long period of time, then it may be worthwhile to spend more time planning, since the system has more time to realize gains from the planning effort. However, if the system is expected to need to replan quickly, spending time optimising for the current state may be wasted, since this effort will need to be repeated before gains are realized. The final expected utility then is equal to $s * p + d + a * (w - (t + p))$, where s is the system's initial utility, p is the planning time, d is the utility accrued during plan execution, a is the utility value after the plan is executed, w is the *window size*, and t is the time plan's execution time. The fitness at each time is computed based on the profit and latency equations in Section 3.2. Planning time is measured as the amount of time in seconds that the GP planner required to generate the plan. The plan execution time is determined based on the tactic executions times in Section 3.3.

In Section 5, we investigate several additional heuristic modifications to fitness computation to manage invalid actions and plan size. These include an *invalid action penalty* per invalid tactic, a *verboseness penalty*, which penalizes a plan proportional to its size; a *parsimony pressure kill ratio*, which assigns a fitness of zero to a random proportion of individuals larger than the average population size; and a *branch pruning threshold*, which assigns a fitness of zero to branches of the system state tree when the maximum possible contribution of that branch to the system's overall fitness is below the threshold. In reporting final plan utility values, we report the aggregate expected utility, without any internal heuristic fitness penalties used during the GP search (i.e., the verboseness penalty).

4.4 Plan Reuse

Our approach reuses past knowledge by seeding the starting population with prior plans. After the system model is updated to reflect the unexpected change, a starting population of plans is created. These plans are iteratively improved by random changes via mutation and crossover, with the most effective plans being more likely to pass into the next generation, resulting in utility increasing over time (although this is not guaranteed). Seeding previously useful plans into the population allows for useful pieces of planning knowledge to spread to other plans during crossover. Note that in this work, we do not address the important issue of how to select plans for reuse, and focus attention on reusing a single plan at a time. We consider how a self-* planner can reuse a repertoire of saved plans that are amenable to reuse and likely to generalize to future situations in other work [30]. To generate the starting plan, we generate a plan for the baseline scenario using the GP planner initialized with a population of randomly generated plans, rather than seeding the population with prior plans.

As we detail in Section 5, preliminary results show that initializing the search by naïvely copying existing plans did not result in efficient planning, and in most cases was inferior to replanning from scratch with a randomly generated starting population. In these preliminary experiments, the cost of evaluating even the first generation of individuals was very high, to the point that planning from scratch could sometimes converge to a good solution before the first generation was done evaluating. One key reason for this is the high cost of calculating the fitness values of long starting plans. Specifically, because fitness evaluation must consider the possibility that every tactic in the plan may succeed or fail, the evaluation time is exponential with respect to the plan size. To realize the benefits of reuse, we introduce several strategies for lowering this cost, including seeding the initial population with a fraction of randomly generated plans in addition to previous plans, prematurely terminating the evaluations of long running plans, and reducing the size of starting plans by randomly splitting these plans into smaller plan trimmings. Table 2 shows a summary of these approaches.

To reduce the number of long starting plans that the planner needs to evaluate, we initialize a *scratch_ratio* percentage of the starting population with short (a maximum depth of ten) randomly

Table 2. A Summary of the Reuse Enabling Approaches

Approach	Technique	Rationale
scratch_ratio	Generate some percentage of plans from scratch rather than all reused.	Short plans generated from scratch are much faster to evaluate, reducing the overall evaluation time.
kill_ratio	Prematurely terminate some percentage of the longest evaluating individuals.	A few very large plans can take significantly longer to evaluate than the rest of the population.
trimmer	Reuse randomly chosen plan trimmings rather than entire plans.	Plan trimmings contain the information from the initial plan, but shorter plans are much faster to evaluate.

generated plans, and only seed the remaining $1 - scratch\_ratio$ individuals with reused plans. This reduces the amount of time spent evaluating the fitness of the starting plan in the new situation while still allowing for the reusable parts of the existing plan to bootstrap the search.

Since the evaluation time is exponential with respect to the plan size, a few of the longest plans can take significantly longer to evaluate than the rest of the population. To prevent wasting search resources on excessively long plans, we introduce a *kill_ratio* parameter that terminates the evaluation of overly long plans and assigns them a fitness of zero. When *kill_ratio* percentage of individuals have been evaluated, evaluation stops and all outstanding plans receive a fitness of zero. This approach leverages the parallelizability of GP to avoid hard-coding hardware and planning problem-dependent maximum evaluation times but requires planning on hardware with multiple cores.

Last, to further reduce the cost of reuse, rather than completely copying large starting plans, we initialize the search with small plan "trimmings" from the initial plan. Our planner generates trimmings by randomly choosing a node in the starting plan using Koza's node selector [32] (an approach that assigns a probability of selection to every node in a tree) that can serve as the root of a new tree. This subtree is then added to the starting population. The process is repeated until the desired number of reused individuals is obtained.

Like the many other parameters that must be set in stochastic search algorithms, the settings for the *scratch_ratio* and *kill_ratio* are domain specific. In this work, we set them by performing a parameter sweep to determine what ratios perform well and leave the interesting question of discerning a fundamental principle for how to choose these parameters to future work.

5 EVALUATION

We built the genetic programming planner described in Section 4 on ECJ, a framework for evolutionary computation in Java.[1] This section describes our evaluation, which investigates whether our approach for reuse can improve the effectiveness of self-* systems in responding to unexpected changes compared to planning from scratch, as well as compared to an exhaustive planning approach. We evaluated our approach for plan reuse using two case-study systems featuring different domains and planning assumptions. The first is Omnet, a cloud-based web server described in Section 3. The second case study, DART, is a team of autonomous drones that must detect targets in a hostile environment while avoiding threats, described in Section 5.2.1. This second case study

[1]ECJ is available at https://cs.gmu.edu/~eclab/projects/ecj/. The source code for our planner is available at https://github.com/ZackC/AdaptiveSystemsGeneticProgrammingPlanner.

highlights reuse in a domain where planning occurs every timestep and features a different search landscape. Section 5.1 describes the experimental setup and reports results for the Omnet case study. Section 5.2 does the same for the DART system. Finally, Section 5.3 provides a discussion of our results and their implications for evolving self-* systems.

5.1 Omnet Evaluation

For the Omnet case study, we investigated the following research questions:

(1) As a sanity check, how do the GP planner's efficiency and effectiveness compare to an exhaustive planner?
(2) Can plan reuse improve planning utility in response to unforeseen adaptation scenarios?
(3) Does our planner's techniques for facilitating reuse improve planning effectiveness?
(4) How does plan reuse impact population diversity?

In all experiments using Omnet, we evaluate various scenarios based on the system shown in Figure 2 and described in Section 3. The evaluation is performed on a simulator implemented in Java based on the description of the case-study system in Section 3, and implements the fitness function described in Section 4.3. The system begins each scenario with one server of each type, a default traffic setting of 4, and all dimmers set to 0. The experimental server ran 64-bit Ubuntu 14.04.5 LTS with a 16 core 2.30-GHz CPU and 32 GB of RAM, but was set to limit the planners to 10 GB of RAM. The GP used 8 of the available CPU cores. PRISM experiments use version 4.3.1 and the sparse engine. We set the planning horizon to 20 for PRISM, and the maximum plan tree depth to 20 for the GP planner. Since the GP planner incorporates randomness and we measure planning time, planner executions are repeated 10 times and the median values are reported. Where statistical tests are used to assess significance, we use the Wilcoxon rank-sum test, a non-parametric test that does not require the samples to follow a normal distribution, and is appropriate for small sample sizes. When $P < 0.05$, we reject the null hypothesis that the samples arise from the same population. In the multi-objective context, we compute a SPEA2-defined Pareto optimal front optimizing for two or more of the given utility objectives. Selecting a particular plan from the Pareto front might be done by a human in the case of offline planning, or automatically during online planning. The selection strategy is out of scope for this work but could be accomplished easily by random selection, since each solution is non-dominated with respect to each other. We set the SPEA2 algorithm elite set to 50. In experiments that we compare to PRISM, we disable reasoning about tactic latency since this is not easily achieved in PRISM. Where tactic latency is considered, we set the window size to be 10,000 s. Where we compare to searches from "scratch," we use Koza's ramped half-and-half [32] algorithm for constructing random trees to initialize the population.

5.1.1 Comparative Study. **Efficiency.** As a sanity check to establish that our stochastic planner achieves reasonable results, we first tuned and compared it to an exhaustive planner from previous work [43], an MDP planner written in PRISM.[2] We configured the planner with the same settings as in the previous work, adding path probability to the system specification and planning for a single environment state. For this experiment, we disabled reasoning about tactic latency in the GP planner, since this is not supported by the PRISM model.

As with many optimization techniques, a GP typically includes many tunable parameters that require adjustment. We thus performed a parameter sweep to heuristically tune the reproductive strategy (which determines how individuals in the next generation are produced, a ratio of

[2]Because the Pandey et al. approach [43] was not named, and we assess the limitations of PRISM rather than the hybrid element, we refer to this as the PRISM planner for the remainder of the article.

Table 3. The Parameter Settings in the Parameter Sweep

Parameter Name	Tested Values
Generations	10, 30, 100
Population Size	10, 100, 1000
Crossover	0.9, 0.8, 0.7, 0.6, 0
Mutation	1, 0.4, 0.3, 0.2, 0.1
Reproduction	1, 0.4, 0.3, 0.2, 0.1
Parsimony Pressure Kill Ratio	0.2, 0.1, 0
Verboseness Penalty	10, 1, 0, 0.1, 0.01, 0.001
Invalid Action Penalty	10, 1, 0, 0.1, 0.01
Branch Pruning Threshold	10, 1, 0, 0.1, 0.01, 0.001

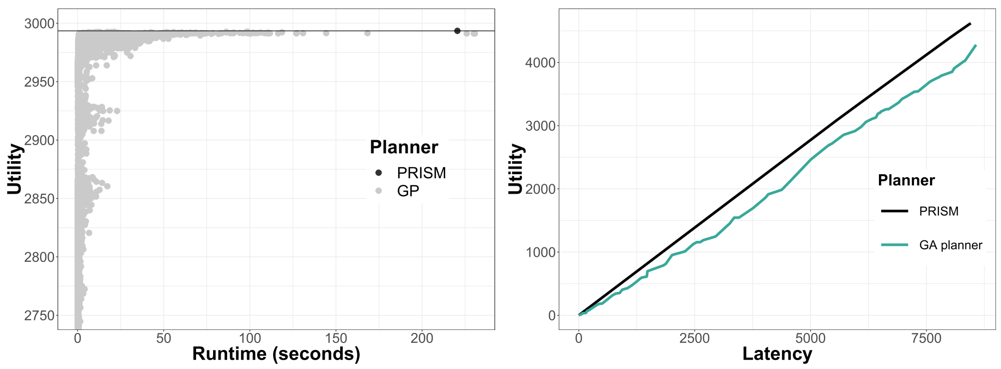


Fig. 5. Left: Utility versus planning time for GP parameter configurations. Many configurations produce similar utility results to PRISM, significantly faster. Right: Pareto fronts for utility (higher is better) and latency (lower is better) from both planners.

crossover, mutation, and reproduction/copying) and number of generations, population size, and all penalty thresholds (Section 4.3). We generated plans for the system's initial configuration (Section 3), and started each search from a hand constructed minimal plan of four tactics that does not affect utility. This starting plan consists of four tactics that attempt to change the systems traffic and dimmer values outside of the allowed range, and are thus discarded. Table 3 shows the parameter values covered in the sweep.

The dark point at the top of Figure 5 shows the optimal system profit (fitness) and planning time (200 s) of the PRISM planner. Each gray point corresponds to a different parameter configuration of the GP planner. Many parameter configurations allowed the GP planner to find plans that were within 0.05% of optimal, but in a fraction of the time (under 1 second in some cases). An example plan that achieved close to the optimal utility is shown in Figure 6. The best configuration that produced plans in 0.50 s resulted in only 0.29% error, which demonstrates that the planner has the potential to be used as an online planner that reacts to change in real time. This top configuration used 30 generations each containing 1,000 individuals; the next generation is produced 60% by crossover, 20% by mutation, and 20% reproduction; applied 0 parsimony pressure and 0.01 verboseness penalty (i.e., a small penalty for large plans); and an invalid action penalty of 0. We use these values in subsequent experiments unless otherwise indicated.

Search space. Next, we evaluate and compare the planners' search space limitations. We varied the search space size by adjusting the number of available server types (t) in our scenario,

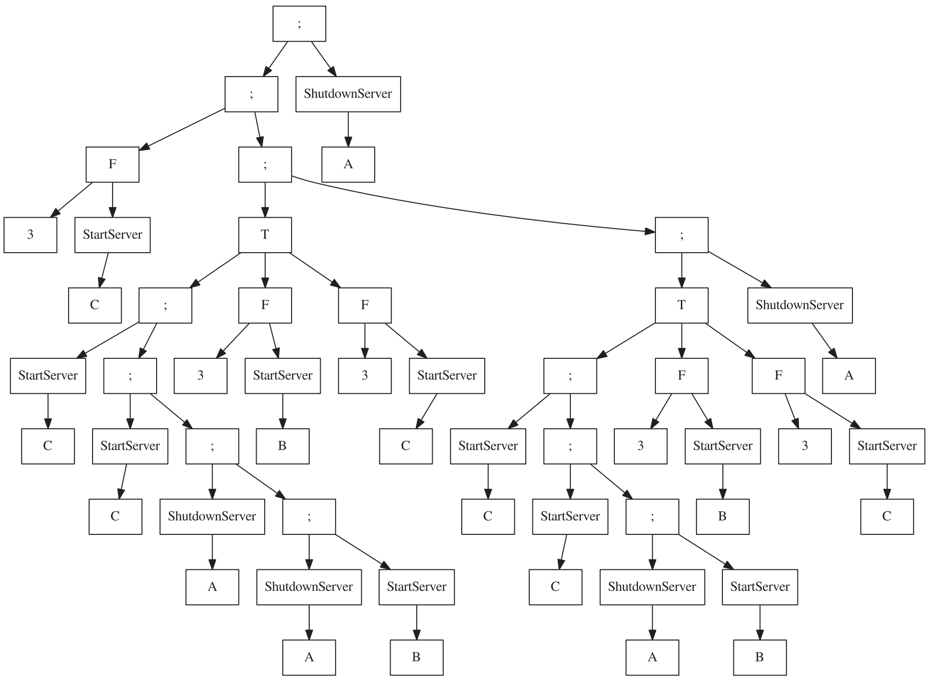


Fig. 6. An example plan generated for the cloud web server case study.

which caused the model states to grow exponentially following the equation $(6servers\_per\_type \cdot 5possible\_dimmer\_values \cdot 5possible\_traffic\_values)^t$.

We found that PRISM can plan to maximize profit for three server types, with a maximum plan size of 20 tactics. However, PRISM runs out of memory and produces no plans when given four server types to consider, even when searching for only a single tactic. Using the explicit engine, which requires less memory but more runtime, PRISM could produce a plan for four server types for a plan length of up to seven. By contrast, our GP planner succeeded on the four server type case, increasing profit from 988 in the initial state to 2993. Finally, we increased the number of data centers from 4 to 16, a state space on the order of 10^{37}, and successfully generated a plan after about 9 minutes. These tests demonstrate that the GP planner can handle a very large search space, outperforming an exhaustive planner, and provides evidence that the planner works correctly to build confidence in our core experiments investigating plan reuse.

Multi-objective search. The GP planner can create a Pareto frontier of plans to tradeoff between multiple quality attributes, allowing system maintainers to evaluate the best possible combinations. PRISM can also generate a Pareto frontier for two objectives. The right of Figure 5 shows the Pareto fronts as lines for the profit (higher is better) and latency (lower is better) objectives produced by PRISM and the GP for the Request Spike scenario. For this experiment, we set the planning horizon for both planners to 10. PRISM found 30 points along the curve; the GP planner produced 89, after removing duplicates. PRISM took 1177 s; the GP planner took 751 seconds. The front produced by the genetic planner roughly approximates the front produced by PRISM, with 9.4% average error.

We also generated three-dimensional Pareto fronts for all three quality objectives with the GP planner. PRISM cannot produce fronts in this case, and the graphs are difficult to display, but we observe that the starting plan influenced the shape of the resulting front. If we begin with

Table 4. Improvement Obtained by Reuse Enabling Techniques

Planning Technique	Starting Population	Utility	*P* Value
Scratch	0% seeded	1.000	
Scratch & *kill_ratio*	0% seeded	1.044	<0.01
Reuse	100% seeded	0.962	0.06
Reuse & *kill_ratio*	100% seeded	1.072	<0.01
Reuse & *kill_ratio* & *scratch_ratio*	10% seeded	1.077	0.63
Reuse & *kill_ratio* & *scratch_ratio* & *trimmer*	10% seeded	1.112	<0.01

plans previously optimized for profit, then we find Pareto fronts with more high-profit individuals. Starting from a lower-quality plan, or planning from scratch, produced a broader front of lower latency individuals. In effect, these starting plans led the search to explore more of the tradeoffs between latency and quality. We explore the tradeoffs of plan reuse more directly in the next set of experiments.

5.1.2 Reuse-Enabling Techniques. While the previous results inspire confidence that the planner can be competitive with an optimal planner, our primary goal is to use the GP planner to realize increased planning ability in response to unexpected changes through reusing prior plans. Since preliminary results showed naïvely reusing entire plans in the starting population resulted in poor planning performance, recall we explore several techniques for lowering the cost of reuse (Section 4.4), the *kill_ratio*, *scratch_ratio*, and plan *trimmer*.

To demonstrate the usefulness of these features, we performed planning for the Request Spike + New Data Center scenario with a planning window of 10,000, incrementally enabling the proposed reuse enabling techniques to show the improvement obtained from each feature. For comparison we also plan from scratch both with and without using *kill_ratio*. When used, the values chosen were *kill_ratio* = 0.75 and *scratch_ratio* = 0.5. These values were selected based on a parameter sweep.

Table 4 shows the results, normalized to the utility of planning from scratch without the *kill_ratio*, such that this utility is 1 (i.e., 1 would be the same as planning from scratch, 2 would be twice the utility, 0.5 would be half the utility, etc.). Using the *kill_ratio* without reuse improved utility to 1.044. This makes sense, because even though the plans start out shorter, eventually the plans become large and the largest plans require a disproportionate amount of time to evaluate due to the exponential relationship between plan size and evaluation time. However, we expect *scratch_ratio* to be more useful when replanning with reuse, since the reused plans are often large from the start of the search. Plan reuse without any reuse-enabling techniques resulted in a utility of 0.962, underperforming compared to planning from scratch. Enabling the *kill_ratio* feature improved the utility obtained by reusing plans to a level slightly better than planning from scratch while using the *kill_ratio*. Adding the *scratch_ratio* resulted in a slight improvement of 0.005, and trimming the reused plans resulted in a further improvement of 0.035. The *scratch_ratio* did not show a statistically significant improvement for this scenario, but did for the `Increased Costs` scenario at the 0.05 level. Trimming plans and the *kill_ratio* both showed statistically significant improvements.

These results demonstrate that while the costs of evaluating the fitness of prior plans make improving planning utility through reuse nontrivial, the proposed enhancements to GP planning can reduce this cost and achieve higher utility than planning from scratch.

5.1.3 Unforeseen Adaptation Scenarios. We investigate the GP planner's ability to address unforeseen adaptation needs with plan reuse. We do this by constructing unexpected change scenarios that cover different types of adaptation needs based on different sources of uncertainty, and

Table 5. Percentage Change Reusing Plans Instead of Planning from Scratch

Scenario	1K	10k
`Increased Costs`	0.02	0.81
`Network Unreliability`	0.01	**0.10**
`Failing Data Center`	−0.02	**0.14**
`Request Spike`	−0.14	−0.01
`New Data Center`	−0.63	0.28
`Request Spike + New Data Center`	−0.47	**1.54**

Statistically significant results ($P < 0.05$) are shown in bold.

assessing the planner's ability to respond when planning with reuse compared to planning from scratch. The considered scenarios are described in Section 3.4, and include: *Increased Costs*, *Failing Data Center*, *Request Spike*, *New Data Center*, *Request Spike + New Data Center*, and *Network Unreliability.*

For each change scenario, we modified the simulator to behave according to the change relative to the initial scenario (Section 3). We maximize profit in all experiments; box and whisker plots show the best individual in the population each generation over 10 planner executions. We show convergence in terms of the quality (profit) of the produced plans over GP iterations, a machine- and problem-independent proxy for evaluation time. When performing reuse, all three reuse-enabling techniques are used.

Table 5 shows the percentage change between planning from scratch and plan reuse for each scenario and for two window sizes. Positive values indicate the reuse resulted in an improvement, negative values indicate a decrease in utility compared to planning from scratch. Most values showed a small difference that was not statistically significant. For the smaller window size, no values were statistically significant, indicating that there is no statistical difference between plan reuse and planning from scratch. For the larger window size, half of the scenarios showed statistically significant improvements from planning from scratch, with the complex Request Spike + New Data Center scenario showing the largest improvement. Since a larger window size means that the system has more time to realize the benefits of a higher-quality plan, this result is intuitive. Additionally, since a more complex change scenario is more difficult to plan for, it follows that plan reuse results in a greater improvement for these scenarios. While in most cases the differences are small, these results show that our approach using plan reuse can result in utility improvements.

Generations of Planning. Figure 7 shows the utility over generation produced by the GP for each of the considered scenarios for the first 20 generations of planning. A common pattern is that for the early generations of planning, plan reuse outperforms planning from scratch, with the two eventually converging to the same fitness after generation 20, although some scenarios converged more quickly. Intuitively this makes sense, since having access to useful planning knowledge through reuse gives the search a head start in the early generations of planning, and given enough planning time eventually both approaches converge near an optimal solution. Plan reuse performed the best for the `Request Spike + New Data Center` scenario, in which the system replans for a large increase in the number of system requests handled by previous plans (e.g., the Slashdot effect [51]). We also provide the system with a new data center, D, to possibly use to address this issue. Plan reuse also performed well in the `New Data Center and Request Spike` scenarios individually, but less prominently than in the hybrid scenario.

Wall-clock time. Because fitness evaluation time varies by plan size, the amount of time needed to evaluate the fitness of each generation is variable, making the number of generations an

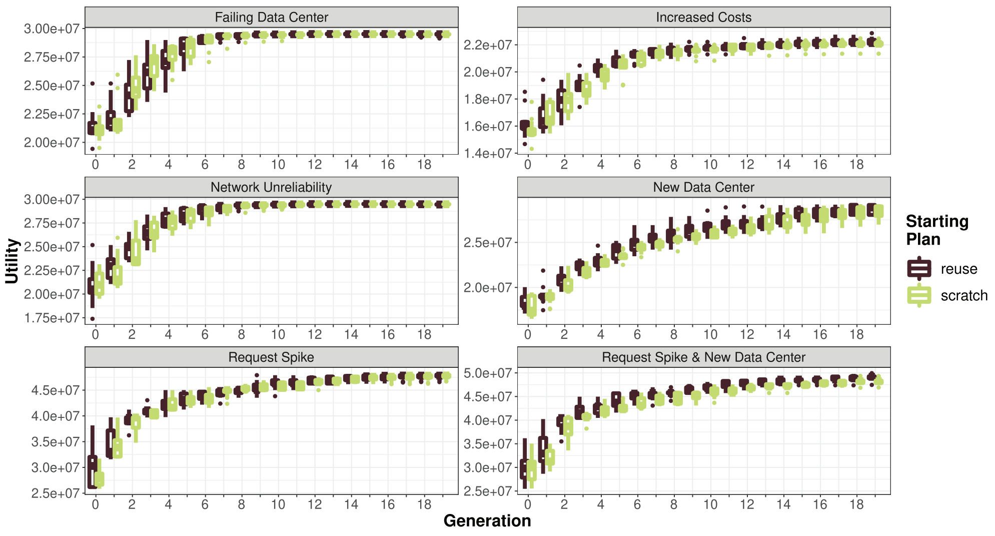


Fig. 7. Utility versus generation for all six scenarios.

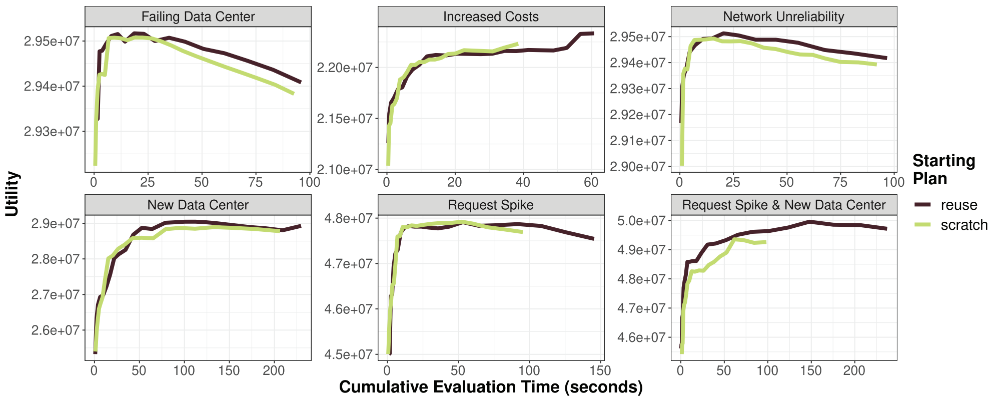


Fig. 8. Utility versus cumulative runtime for all six scenarios.

imperfect proxy for runtime. Thus, Figure 8 shows results in terms of wall-clock time for each scenario. Note that utility can decrease over time, since time spent planning means that the system remains in a lower utility state for longer. The `Network Unreliability`, `Increased Costs` and `Failing Data Center` scenarios showed similar behavior, with only a very small benefit from reusing plans. The `New Data Center` and `Request Spike + New Data Center` scenarios showed greater differences, in particular the `Request Spike + New Data Center` scenario showed a clear advantage to reusing existing plans.

5.1.4 Diversity. Genetic programming balances search space *exploration*, to avoid local optima, and *exploitation* of promising partial solutions. Solution diversity is necessary to support exploration of good partial solutions; however, it typically decreases as the search *converges* [36], assuming that the population is sufficiently diverse. To gain additional insight into plan evolvability given different scenarios, we measured the syntactic population diversity over a search by computing the average pairwise tree edit distance of the individuals, using the APTED

Table 6. A Comparison of the Omnet and DART Systems

	Omnet	DART
Tactics can fail	Yes	No
Planner executions	Once	After every timestep
Plan execution	Entire plan	First tactic only
Changing environment	Not after planning	After every timestep

algorithm [45] (a state of the art approach that uses dynamic programming to obtain polynomial runtime).

Figure 9 shows population diversity across the scenarios. Diversity values from planning from scratch, as well as reusing plans both with and without trimming (the mutator starting plan) are shown. The lower plot shows diversity computed by structure only, that is, the labels of nodes in the tree are assumed to be identical, allowing computation of the difference in the structure of trees only. Either way diversity is measured, planning from scratch produces a highly diverse starting population that gradually becomes less diverse as it converges toward a high-quality solution.

As shown in Figure 9, reusing existing plans without first trimming them results in a less diverse population initially. Rather than a gradual decrease in diversity as would be expected, in some situations (such as generations 2–8 for the `New Data Center`) the diversity actually increases as the population explores new plans before continuing to converge on a good solution. However, when using the *trimmer*, the diversity values start high and smoothly decrease. This observation helps to explain why trimming existing plans resulted in a more significant improvement than the *scratch_ratio* alone, since the presence of smaller plan trimmings facilitates a smoother exploration and exploitation tradeoff as the population evolves.

5.2 DART Evaluation

The first case-study system, Omnet, provided an example self-* system modeled on a cloud-based web server. To investigate the usefulness of plan reuse in a different domain, we apply our approach to a simulated team of autonomous aircraft called DART, implemented by modifying the DARTSim exemplar [39] to accommodate the unexpected change scenarios introduced in our study and to integrate with the GP planner. Table 6 outlines the key differences between the case-study systems from a planning perspective. While the challenge of planning for tactic failure is relaxed in this system, the planner must replan after every timestep. Additionally, only the first tactic in the plan is executed at each timestep. Last, as the team moves, the system must respond to changes in its environment.

In this evaluation, we address the following research questions for the DART system:

(1) As a sanity check, how does the GP planner's efficiency and effectiveness compare to an exhaustive planner?
(2) Can plan reuse improve planning effectiveness in response to unforeseen adaptation scenarios?

Since tactics cannot fail in this case study, effective plans tend to be shorter as contingencies for tactic failure do not need to be built in. This makes the `kill_ratio` and `trimmer` less applicable to this case study, and, as a result, research questions involving the reuse enabling approaches are not evaluated.

Section 5.2.1 describes the DART system, including how we adapt our approach from Section 4 to handle the new case. Section 5.2.2 describes the results of the experiments involving DART.

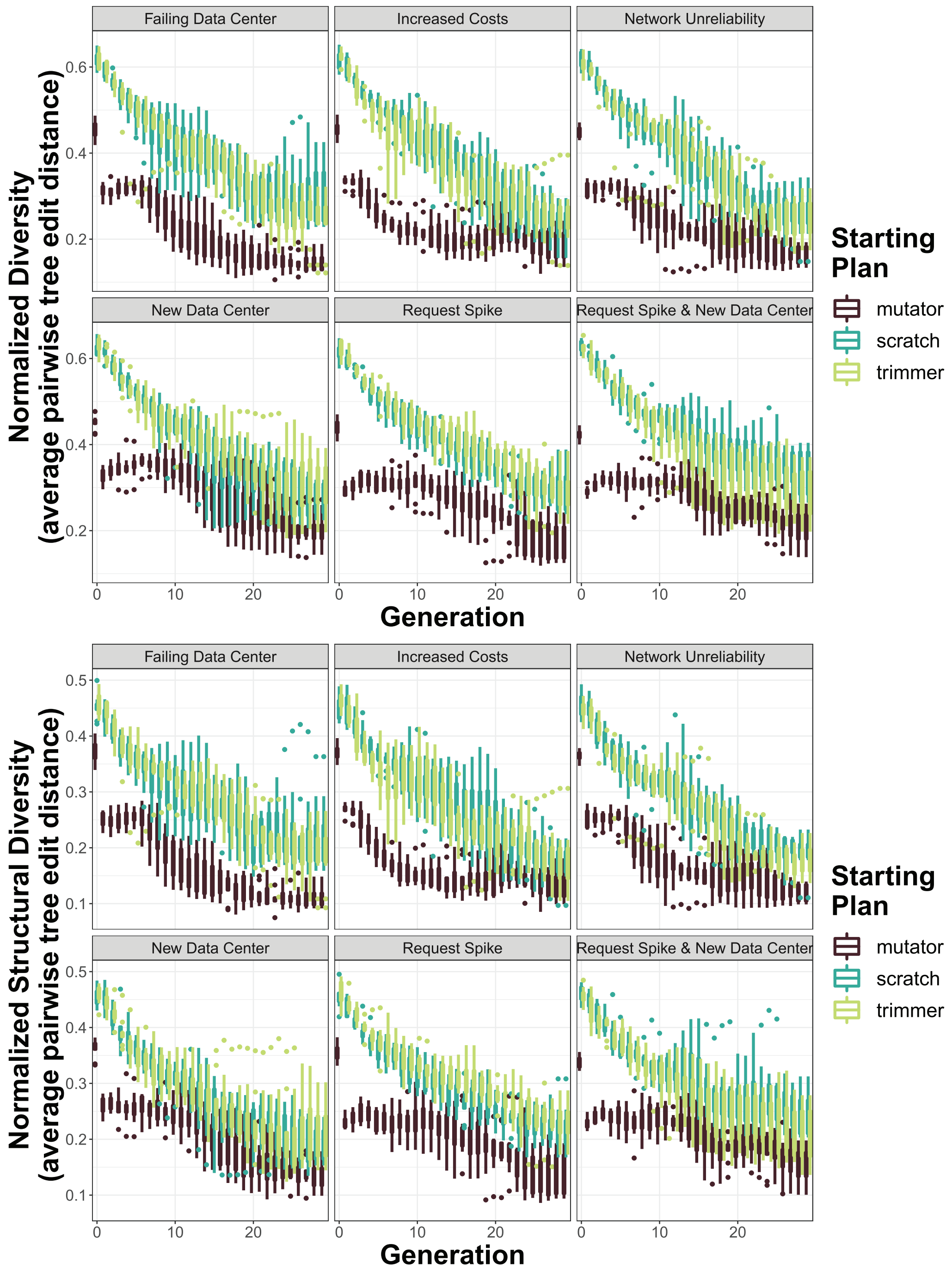


Fig. 9. Diversity versus generation for all six scenarios.

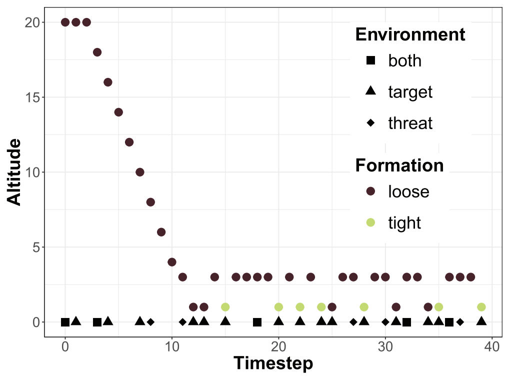


Fig. 10. An example trace of the DART team moving through an environment.

5.2.1 DART System Description. We also evaluate our approach for plan reuse in a second case-study system inspired by a scenario from the DART systems project [22], the DARTSim [39] exemplar. In this case study, the system is a team of autonomous aerial vehicles or drones. The team flies together in formation, and a central leader drone commands the rest of the team autonomously. The team's mission is to fly over a predetermined path in hostile territory, detecting targets while avoiding threats.[3] The team's path is divided into discrete locations, and the team moves at a constant speed, traversing one location per timestep. The team is equipped with noisy sensors that allow the drones to estimate the probability that a threat or target lies in each location in their look-ahead horizon, with the accuracy of these estimates improving with each timestep that the location is sensed. The team's configuration influences whether the team detects a target or is destroyed by a threat when encountered. This configuration includes the altitude of the team, with higher altitudes offering greater protection against threats, but also reducing the ability of the team to detect targets. The team can also change the tightness of its formation, with a tight formation offering reduced exposure to threats, but also less sensor coverage to detect targets. Last, the team can enable electric counter measures (ECM), which also decrease the chance that a threat destroys the team, but at the cost of reducing the effectiveness of target detection. In this work, the team starts at a high altitude of 20, and must descend 16 levels before utility gain can occur. This situation could arise if the team was retasked from another mission, or must arrive at the mission area due to air traffic restrictions or to avoid other threats. More generally, this aspect of the case study is indicative of self-* systems that require a specific initialization before utility can be affected by adaptation. Figure 10 shows an example simulation of the DART team moving through an environment. Each dot indicates the position and formation of the DART team at a particular timestep. Black shapes at the bottom of the figure indicate the positions of threats and targets in the environment.

Objectives. The team's goal is to detect targets while avoiding threats, without knowing the number of location of targets or threats beforehand. When the team occupies the same location as a target, the team detects the target with some probability, which is based on the team's altitude and configuration. Likewise, when the team occupies the same location as a threat, the team is destroyed with some probability based on the team's state.

[3]While described in terms of maneuvering through a hostile environment, the DART case study is representative of a general domain of autonomous systems that need to explore an environment to locate targets of interest, for example, disaster victim rescue or surveying wildlife populations.

The team's utility U is the expected number of targets detected over the course of the mission, plus the team's probability of survival. This can be found according to the following equation:

$$U = \sum_{t=1}^{T} \left(\left(\Pi_{i=1}^{t} (1 - d^i) \right) \cdot g^t \right) + \Pi_{i=1}^{t} (1 - d^i).$$

This expression sums, over each timestep t, the product of the probability that the team survives until each timestep with the probability that the team observes a target. Here, d^i denotes the probability that the team is destroyed at timestep i. Since the team's chance to survive until the current timestep depends on the team surviving the previous timesteps, the first inner term of the summation provides the multiplicative probability that the team survives until timestep t. The second term g^t denotes the probability that the team observes a target at timestep t. The team's probability of surviving the mission is added to the sum to discourage the team from sacrificing itself at the end of the route (which the team might do to get a better chance of observing the final target if survivability is not a concern).

Both values d and g depend on the team's configuration (and therefore the chosen plan) and the positioning of threats and targets in the environment. The team's probability of destruction is higher when the team is lower in altitude, looser in formation, and the ECM state is off. These variables have the opposite effect on the team's probability to observe targets. A complete treatment of how d and g are computed is provided in prior work [39].

Adaptation Tactics. The team has eight adaptation tactics available. The team can ascend or descend in altitude. Since it takes time to change altitude, a timestep is necessary before the effects of these tactics are felt. Airspace is divided into twenty levels, and an `IncAlt` or `DecAlt` tactic results in the team moving up or down one level in the next timestep. An additional two tactics, `IncAlt2` and `DecAlt2`, allow the team to traverse two altitude levels instead of one. The team can be in either a loose or tight formation, toggled using the `GoLoose` and `GoTight` adaptation tactics. Lastly, the team's ECM state can be toggled by the `EcmOn` and `EcmOff` tactics. Changes to the team's formation and ECM state occur the same timestep as the tactic is used.

Post-design-time Adaption. We examine the usefulness of plan reuse to three types of change scenarios, changes to the positions of the threats and targets present in the environment, changes in the available adaptation tactics, and changing the desired utility tradeoff between the survivability of the team and the expected number of targets detected. In each of these change scenarios, the team begins in a different state and after the change scenario arrives at a common state, allowing direct comparison of utility values between change scenarios.

Integration with GA Planner. The approach for using the GA planner is largely the same as described in Section 4 but with a few modifications to accommodate the new case study and its assumptions. Unlike the cloud-based server case study where tactics may fail, in this scenario, tactics are guaranteed to succeed as long as the team has not been destroyed. However, the DART case study involves the system moving through a changing environment, replanning after each timestep, while the server case study was restricted to generating a single strategy that is committed to after a single execution of the planner. We simplify the planning language to include only the sequence operator and a terminal for each adaptation tactic. To evaluate fitness, we compute the expected utility of the team, which is the sum of the expected number of targets detected and chance of survival. Due to these differences, the *kill_ratio* and *trimmer* reuse enabling approaches are less applicable and are not used. The *scratch_ratio* reuse enabling approach is used with a value of 0.9. Figure 11 shows an example plan generated for the DART case study.

5.2.2 Parameter Sweep. To choose parameters for the GP planner for this case study, as well as to provide a sanity comparison to an exhaustive planner, we performed a parameter sweep of the

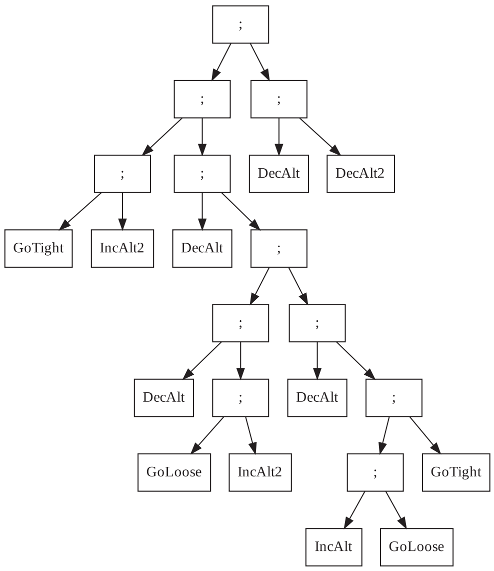


Fig. 11. An example plan generated for the DART case study.

Table 7. The Parameter Settings in the Parameter Sweep

Parameter Name	Tested Values
Generations	1, 10, 25, 75, 100
Population Size	1, 10, 100, 1000, 10000

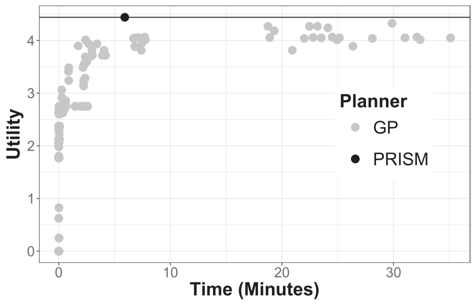


Fig. 12. Utility versus planning time for GP configurations.

population size and number of generations of evolution. The remaining parameters were kept from the Omnet case study. To compare to an exhaustive planner, we compare to the probabilistic model checking approach presented in related work [40]. This approach models the problem as an MDP and uses the PRISM probabilistic model checker to generate an optimal plan at each timestep. To reduce the computing resources required by the sweep, we record results from planning for a single timestep only in this experiment. Table 7 shows the parameter values used in the sweep. Figure 12 shows the results of the parameter sweep. Each grey dot represents the utility and planning time for a single combination of parameters. The black dot and horizontal line shows the utility and total planning time obtained by model checking using PRISM. Since PRISM finds the optimal plan, we expect it to result in the highest utility. We chose parameter values near the knuckle point of this figure to strike a balance between plan utility and speed, settling on a population size of 1,000 individuals evolved for 30 generations.

However, since our sweep focused on a single timestep only rather than a complete simulation, we found that these parameter values could not be used to generate plans from scratch. This is because the search space in the first few timesteps of the simulation is very coarse, with almost all plans resulting in a utility of zero. This occurs because the team always starts at the highest altitude level, but cannot detect targets until the team is close to the lowest level. Thus, the team receives an expected utility of zero unless a very specific sequence of tactics (many consecutive commands to descend and few commands to ascend), which is unlikely to be generated at random with only a population size of 1k. We discovered in preliminary experiments that a population size of 10k provides enough sampling to allow the planner to converge to a good solution. Therefore, when planning from scratch, either for purposes of comparison or finding starting plans to reuse, we use a population size of 10k. When we reuse existing plans, however, we use a population size of 1k, since reuse allows the search to immediately start converging to a good solution. A comparison between both planning approaches set to 1k would result in planning from scratch failing to produce a useful plan, instead, changing the parameter to 10k allows us to compare how long it takes to generate a useful plan with reuse versus from scratch.

5.2.3 Plan Reuse. To evaluate the benefit of plan reuse in DART, we investigate three unexpected change scenarios.

- **Environment Only.** The location of threats and targets in the teams path is changed, an *environmental* change.
- **Environment + No Survivability.** Utility is determined solely by the expected number of targets detected without adding the survivability likelihood, a change in the *system objectives* in addition to an *environmental* change.
- **Environment + Slow Descend.** The system encounters different configuration of threats and targets, as well as not having access to the `DecAlt2` tactic. An *environmental* change and a change in *available tactics*.

For each change scenario, the team begins in a different state and after the change scenario arrives at a common state, allowing direct comparison of utility values between change scenarios. We performed 10 simulations for each change scenario and measured the utility and planning time. Each simulation uses a different random seed, resulting in a different randomly generated environment. We use the same 10 random seeds for each scenario, permitting easy comparison. Each simulation has a path length of 40 and thus runs for 40 timesteps. At each timestep, the planner is executed for the current system and environment state, using the initial population dictated by the scenario. The planner generates a plan with a length equal to the lookahead horizon, which is 20 for this case study. The system then executes the first tactic in the plan returned by the planner, and the simulation continues to the next timestep, until the team is destroyed or until the team reaches the end of the path. Although only one tactic is executed at a time, generating a plan for the entire lookahead horizon allows the planner to take into account how the action in the present timestep affects the system's utility in the future (for example, descending now if there are threats ahead may result in the team having a higher chance of destruction).

For the `Environment Only` and `Environment + No Survivability Requirement` scenarios, initial plans are collected by running a simulation with the scenario conditions, and saving the best plan from all 40 timesteps. When reusing these plans, the starting population is created by generating a new plan from scratch 90% of the time, and using a randomly chosen plan from the

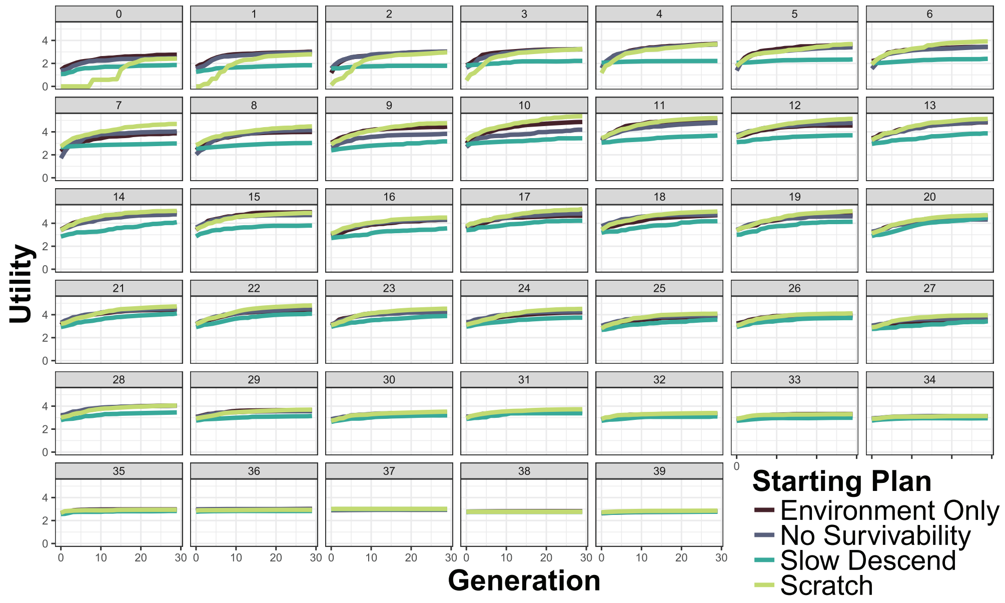


Fig. 13. Utility versus generation by timestep.

repertoire otherwise. The initial population for the `Environment and Slow Descend` scenario could not be generated in this way, because the lack of the `DecAlt2` tactic means that it is much more difficult for the planner to discover a plan to descend enough to be in range of the targets, resulting in planning from scratch to fail to produce a plan with better than zero utility, even when we increase the population size to 100k. Instead, we manually created a plan to guide the system toward the ground, consisting of 16 consecutive `DecAlt` tactics. When generating an initial population for this scenario, a plan is generated from scratch 90% of the time, and the manually created starting plan is mutated and added to the initial population for the other 10%.

Utility by Timestep. Figures 13 and 14 show the average utility of the best available plan in the population during the execution of the planner, for each timestep, and for each change scenario. Note that the utility values for the last timestep are transformed by shifting them down by 4.5 to avoid increasing the vertical axis for this timestep (necessary due to an implementation quirk where the simulator will estimate utility as if the team will continue along a new route at the final timestep). Figure 13 shows how the utility changes between each generation of evolution. For the first few timesteps, planning from scratch significantly underperforms in all three change scenarios for the first five generations of planning. This is not true for the remaining timesteps, however, and the most notable pattern for the remaining timesteps is that reusing plans from the `Environment and Slow Descend` change scenario tends to underperform compared to the other starting plans.

Figure 14 shows utility versus wall clock time. Again, there is a wide gap between planning from scratch and all three change scenarios for the first three timesteps. Overall, the `Environment Only` and `Environment + No Survivability Requirement` scenarios perform better than planning from scratch for the first 30 s of planning. The `Environment + Slow Descend` scenario sometimes outperformed planning from scratch in the first 30 s of planning but usually underperformed compared to the other reuse approaches.

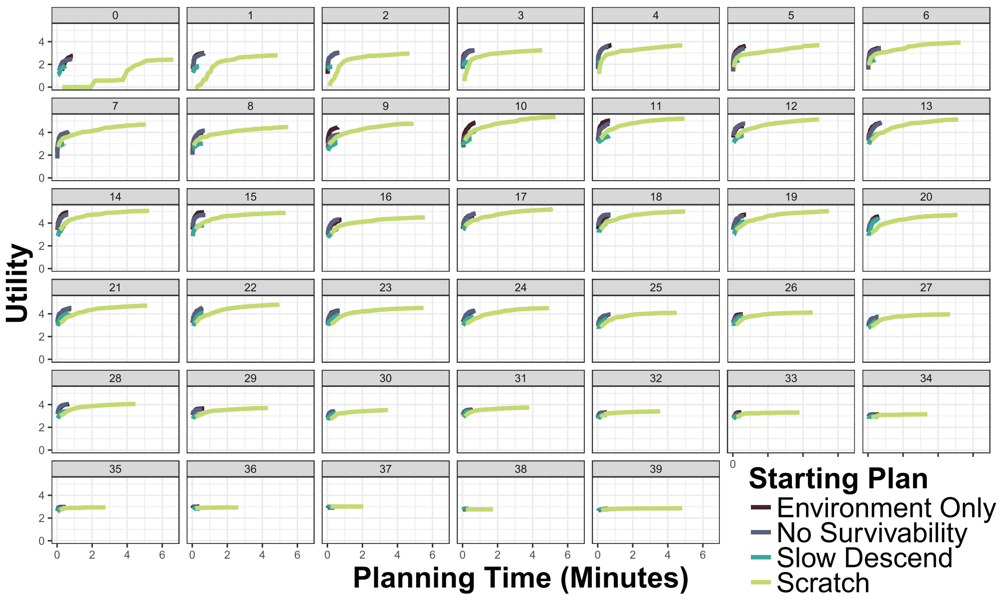


Fig. 14. Utility versus runtime by timestep.

These results show that plan reuse is most beneficial in the first few timesteps of planning. This is due to the coarse shape of the search space at the start of the scenario. Since the team always starts at the highest level of altitude at the start of the simulation, and targets can only be detected when the team is close to the bottom, the planner must discover a very specific sequence of tactics to maneuver the team down (many descend tactics with few ascend tactics), before any plan will have a non-zero utility. Once the planner has a plan reaching such an altitude, any further mutation to the plan results in a small utility delta, enabling the planner to improve the utility of successive generations.

Analysis. In the DART case study, we found a case where plan reuse resulted in significant improvement compared to planning from scratch due to a coarse region of the search space, that is, a region where mutation does not result in a measurable change in fitness. Here, we attempt to quantify the improvement that should be expected from plan reuse versus planning from scratch. In order for the GP to make progress, the search must find a mutation that results in a fitness improvement. If the probability of finding such an improvement is too low, then the search will fail to make progress and will not find a high-quality solution.

The team starts at an altitude of 20, but can only interact with the environment at an altitude of 4 or lower. This means that, any plans that do not result in the team descending 16 levels will have identical fitness. Since the team has the same number of tactics to increase and decrease altitude, and these tactics have mirrored effects (ascending or descending one or two levels), a randomly chosen strategy is expected to have a net altitude change of zero. If these identically ineffectual plans make up a large percentage of the search space, then the planner may become trapped in this area of the search space. This will occur if all of the plans in the existing population are in the coarse region of the space. If the initial population is generated uniformly at random, such as when planning from scratch, then we can determine the probability that this occurs based on the percentage of coarse individuals in the search space.

A plan consists of a sequence of length 20 of 8 possible tactics. Since there are 8 choices for each tactic, the number of possible plans is $8^{20} = 1.15 \times 10^{18}$. To determine the number of plans outside of the coarse region, we can count the number of plans that have a net altitude decrease of 16 or more.[4]

Before showing how these plans may be counted, we will first analyze a simpler case where the team does not have access to the `DecAlt2` and `IncAlt2` tactics. We enumerate the plans that descend 16 levels, we count the number of ways that each combination of tactics that result in altitude change can occur,

$$N_\delta^h = \sum_{c=\delta}^{h} \sum_{d=0}^{\max(h-c,\delta-c)} \binom{h}{c} \binom{h-c}{d} 4^{h-c-d}.$$

In this quantity, h is the number of tactics in the plan and δ is the number of net levels the team must descend to reach the smooth region of the search space. The first sum enumerates every possible number of `DecAlt` tactics that can appear in a promising plan. There must be at least δ if δ levels are descended. The second sum enumerates all possibilities for the number of `IncAlt` tactics, the upper bound ensures that the team keeps a net descent of δ levels. For each possible number of altitude change tactics, the `DecAlt` tactics can be arranged within the h length plan in $\binom{h}{c}$ ways, and for each of these arrangements the `IncAlt` tactics can be placed in $\binom{h-c}{d}$ ways. Then there are $h - c - d$ tactics left, which can be any of the four remaining tactics, so there are 4^{h-c-d} possible choices for this.

When all tactics are available, the following expression counts the number of promising plans:

$$N_\delta^h = \sum_{a=0}^{h} \sum_{b=\max(\delta-2a,0)}^{h-a} \sum_{c=0}^{X} \sum_{d=0}^{Y} \binom{h}{a} \binom{h-a}{b} \binom{h-a-b}{c} \binom{h-a-b-c}{d} 4^{h-a-b-c-d},$$

where X and Y are

$$X = \min\left(\left\lfloor \frac{2a+b-16}{2} \right\rfloor, h-a-b\right)$$

$$Y = \min(2a + b - 2c - \delta, h - a - b - c).$$

Note that since ascending and descending actions have latency, it takes a turn before their effects are felt. Thus, although the planner considers a horizon of 20, an action that changes the altitude on the last timestep would not change the altitude until timestep 21, so only the first 19 tactics can affect the altitude during the planning horizon of 20. The number of plans in the smooth region would then be $8N_{16}^{19}$ (multiplying by 8 is necessary, since there are 8 ways to choose the 20th tactic). To compute the probability that a promising tactic is selected, we must still take into account that the starting population is generated using Koza's grow builder, which generates individuals with randomly selected sizes rather than always generating individuals of the maximum size. The size is chosen randomly between 1 − 20, so then the probability that a randomly selected individual is in the smooth region is $p(\sigma) = \frac{(\sum_{i=1}^{19} N_{16}^{i}) + 8N_{16}^{20}}{20 \cdot 8^{20}} = 0.0036\%$. The probability that the initial population does not get stuck $p(e)$ is when at least one of the P individuals is in the smooth region, given by $p(e) = 1 - (1 - p(\sigma))^P$, which for the DART case-study system is 3.58% when $p = 1000$ and 30.53% when $p = 10{,}000$.

[4]Note that this is an under approximation, since there are plans that reach an altitude of 4 and then increase their altitude. However, the majority of the promising region is captured, since there are more ways for the team to maintain altitude or descend further than there are for the team to ascend back up.

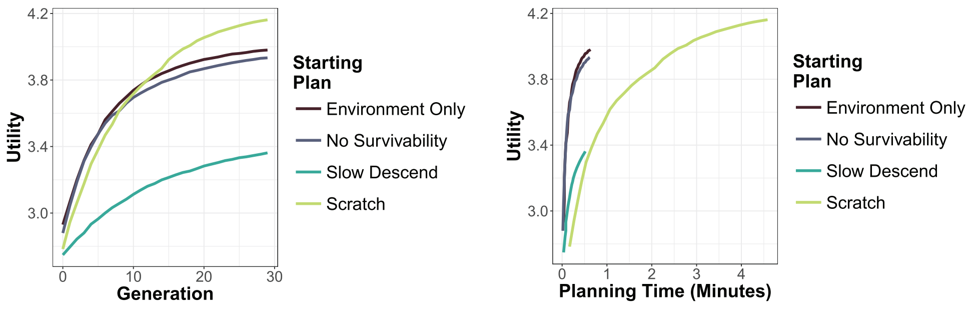


Fig. 15. Left: Utility versus generation. Right: Utility versus planning time.

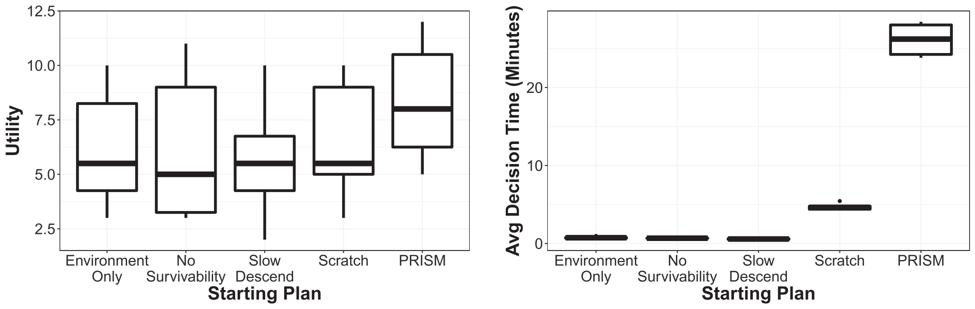


Fig. 16. Left: Utility versus generation. Right: Utility versus planning time.

If we desire the probability of escape to be higher than $\tau\%$, then we can find the number individuals necessary $p(e) > \tau\%$ when $P > \log_{1-p(s)} \tau$. Fixing the number of individuals, we can also quantify how sparse the search space can be while still being tractable by finding the minimum value of $p(\sigma)$ where the search is unlikely to stall, $p(e) > \tau$ when $p(s) > 1 - 10^{\frac{\log \tau}{P}}$.

Expected Utility. Figure 15 shows the average utility during planning over all time steps, giving an overall picture of how various change scenarios compare to planning from scratch. When considering the number of generations the `Environment + Slow Descend` performs the worst, with the other three approaches being fairly close to one another. From the wall clock time perspective however, all three reuse approaches outperform planning from scratch for as long as they are running. The `Environment + No Survivability Requirement` scenario performed about the same as the `Environment Only` scenario. The `Environment + Slow Descend` was the least effective change scenario, but still outperformed planning from scratch during its execution.

Actual Utility. Figure 16 shows the distribution of final results of the simulations for each planning approach (as opposed to the expected utility), as well as a planner using PRISM. This utility may differ from the expected utility because of the stochastic properties of the case study, such as imperfect sensors, and internal modifiers to the fitness function (such as verboseness penalty) and represents the actual utility of running the simulation rather than the system's internal fitness function. The first boxplot shows achieved utility. While the reuse approaches and planning using the GP from scratch all resulted in mostly similar distributions, we see that using an exhaustive planner results in about 2.5 more targets detected on average compared to the other approaches. The right plot shows the average decision time the planner took on each timestep to produce a plan. PRISM took around 25 minutes per timestep. Planning from scratch using the GP required about five minutes. The fastest approaches were the three approaches using plan reuse, which terminate in under one minute. In systems where near real-time adaptation is required, planning for five or 25 minutes is likely unacceptable, while plan reuse produced high-quality plans in the first 30 s.

5.3 Discussion

Our evaluation of two case-study systems revealed several insights for the design of self-* planners resilient to unexpected changes. Reusing plans for the Omnet system revealed that naïve reuse can result in less effective planning than replanning from scratch. In this system, the loss of planning effectiveness stemmed from the long evaluation time necessary to compute the fitness of large plans when we consider that tactics may fail. In this case, applying reuse enabling approaches, *scratch_ratio*, *kill_ratio*, and *trimmer*, to manage the size of candidate plans while maintaining existing planning information can mitigate this problem and improve on planning from scratch. While these techniques resulted in a statistically significant improvement in three of the six change scenarios for the Omnet case study, the magnitude of the improvements were small (under 2%). While the improvements were small, these results represent an important step from naïve reuse that could result in a loss of utility, and shows the promise of plan reuse as an approach for more effectively responding to unexpected changes.

The DART system, however, which does not consider the possibility of tactic failure, significantly reduces the amount of time required for fitness evaluation. Plans in this system are short and simple compared to the Omnet system that requires complex branching plans to manage responding to the success or failure of adaptation tactics. A challenge to using GP for planning for DART is the character of its search space. The Omnet system, while requiring large and complex branching plans to accommodate uncertainty in tactic success, seems to benefit from a smooth, easy to explore search space. Any tactic tends to result in a change to the system's utility, allowing the GP to make steady progress toward convergence on a high-quality solution. The DART case study, however, exhibits a search space with a large coarse region where no sequences of tactics under a certain length results in any change to fitness. However, after the team discovers a plan to maneuver close enough to the ground to potentially detect targets or be destroyed by threats, the search space smooths and plan improvement becomes possible. This property of the problem reveals an opportunity where plan reuse can significantly improve the effectiveness of planning. When an existing plan can guide the system from the coarse to the smooth part of the search space, significant reduction in the planning time is possible. Search spaces with this property may be present in systems with complex initialization procedures or when many steps are necessary to change the system's state. Systems requiring several low level tactics to be used together to result in a meaningful reconfiguration may also exhibit this property and particularly benefit from reuse. Unlike the Omnet case where the improvement obtained by plan reuse was small, in the DART case study reuse resulted in plans with slightly lower utility, but generated them four times as quickly as planning from scratch.

While these results show the promise of plan reuse and stochastic search for improving planning utility after unexpected changes, there remain several limitations to our approach and evaluation. Our approach does not address the problem of updating the self-* system's models after an unexpected change occurs, and our approach depends on having access to these updated models. While this is a limitation for applying the approach in practice, it is not exclusive to our approach and applies generally to other model-based planners. The planning language utilized by the approach is simplistic, and investigating a more fully featured planning language is left to future work. While we selected two case studies from different domains and with different assumptions, and examined several different types of unexpected change scenarios for each case study, it is possible that our results will not generalize to other systems or change scenarios.

6 RELATED WORK

Planning: The artificial intelligence community has produced a considerable body of research in planning, including but not limited to probabilistic approaches like MDP [25, 37]. Some of this work demonstrates the benefits of plan reuse, such as by reusing parts of existing plans that target particular goals in new situations that share those goals [54]; concurrently executing and dynamically switching between plans designed to handle contingencies [4]; modifying plans produced under assumed optimal conditions to handle common problems found in simulation [34]; or iteratively transforming simple plans to produce complex plans [1]. Case-based plan adaptation [42] explicitly reuses past plans in new contexts, in which context GAs have been explored directly [19, 35], e.g., by injecting solutions to previous problems into a GA population to speed the solution of new problems. Although the mechanism is similar, our approach is importantly novel in that it addresses a broader class of uncertainty.

Reinforcement learning has been applied to self-* systems to learn at runtime using Q-learning [24, 28]. Like our approach, this technique could be used to aide in adapting to unexpected changes, and like GP, this technique balances exploration of possible alternatives and exploiting solutions that achieve the best results. Our approach is different from reinforcement learning, because our approach utilizes a model of the system and environment. While reinforcement learning has the advantage of learning online without needing to synchronize a model with the environment, the system will likely perform suboptimally while it performs random actions to learn. This might often be undesirable in many production systems, especially if such actions could result in irreparable damage to the system or others.

A number of works address the problem of updating system models when they become out of sync with reality, such as focusing on architectural evolution [2], identifying when unexpected changes occur for to assist humans in evolving the system [49], or evolving models at runtime [55].

Self-* planning: Mutliple PRISM MDP planners appear in the self-* literature [5, 40, 43]. These techniques are typically offline, as is manual human planning [10], and produce good plans but have issues with problem size limitations. Much of the other work in this space focuses on online planning, e.g., reactively regenerating failing parts of a plan [52] or using hill-climbing [57] (another stochastic technique) to generate plans quickly in exchange for a moderate loss of optimality. Our work focuses in particular on plan reuse to handle uncertainty, and is not reactive.

Plato and Hermes [47] use genetic algorithms to reconfigure software systems (in the domain of remote data mirroring) to respond to unexpected failures or optimize for particular quality objectives. The search problems (representation, operators, and fitness function) differ from ours, commensurate with the different domain. However, the key distinction is our focus on information reuse to handle uncertainty. That is, although both Hermes and our approach are initialized with existing plans, we focus explicitly on the effectiveness of reusing alternative starting plans in the face of unanticipated scenarios. We also compare a GP planner to an optimal planner.

Fredericks et al. [15] presented an approach to planning that can reuse past information and utilizes evolutionary computation. This work reuses prior planning knowledge by maintaining a knowledge base of previously encountered situations. These situations are identified by performing clustering to find environmental parameters that have similar effects on the system, and the planner can retrieve solutions to situations that have already been planned for. This is a different approach to reuse than our work, since we investigate seeding the search with prior plans rather than attempting to retrieve strategies that are relevant to system's current situation.

EvoChecker [17] is an approach using evolutionary algorithms for generating probabilistic models under multiple quality of service objectives. This approach extends the modeling language used by PRISM to support specifying a range of possible models for an evolutionary search. Like our approach, EvoChecker can be used to reconfigure self-* systems at runtime, and supports speeding up the search by reusing information through maintaining an archive of effective prior solutions. Our approach differs by focusing on the planning component of self-* systems and specifically investigates evolving planning languages represented as ASTs rather than the PRISM modeling language. The reuse strategy used by EvoChecker is complementary to our approach, as we investigate approaches for reducing the evaluation time of plans to make reuse more effective, while not investigating how a self-* system should maintain its archive of prior solutions, several such strategies are investigated by EvoChecker.

FEMOSAA [7] also leverages genetic algorithms for runtime adaptation in self-* systems. FEMOSAA is a framework that supports translating the specification of a self-* system from a feature model into an encoding amenable to evolutionary computation, which can then be optimized at runtime. It also provides an approach for automatically identifying knee solutions in domains with multiple objectives. The key difference in our work is that we focus on how self-* planners can replan more effectively by reusing existing knowledge.

Chen et al. [8] investigate seeding evolutionary algorithms with prior solutions in the domain of software service composition. This work investigates four seeding strategies including strategies that reuse historical solutions and solutions generated during a pre-optimization step. Our work is different in that we evolve plans represented as ASTs rather than service composition plans. Additionally, we present several reuse enabling approaches to address the unique challenges posed by seeding evolutionary algorithms for self-* planners, particularly the high cost of evaluating large solutions. The positive results for our reuse enabling approaches, particularly the scratch ratio, which reduces the percentage of individuals that are reused, reinforce the result of Chen et al. that found that the number of seeds is less important than their quality.

GAs have also been used to optimize across multiple quality objectives in quality of service composition [6], and to optimize architectures and associated service providers in self-architecting systems [14]. This prior work broadly substantiates the usefulness of stochastic algorithms in an self-* planning context, but otherwise focuses on very different domains than we do.

In our own previous work [29], we investigated the potential of plan reuse to improve replanning effectiveness in repose to unexpected changes. In this work, we extend our prior article with a new case-study system with a different planning model and assumptions, the DART [22] team of autonomous aircraft. Using this case study, we identify properties of the search space particularly amenable for improvement via plan reuse.

Our initial position [11] identified key research questions that we address in this work, further expanding upon our prior concept in several ways: a more expressive individual representation inspired by true planning languages [10], a significantly more efficient fitness evaluation strategy, a series of techniques for supporting plan reuse by reducing the search cost, a comparative evaluation to an exhaustive planner [43], an experimental study investigating the tradeoffs of plan reuse when adapting to unforeseen scenarios, and support for multi-objective search [56].

In our other work [30], we apply plan reuse to repertoires of plans, while in this work we focus on reusing a single plan at a time. Reusing repertoires of plans raises additional challenges to successful plan reuse, since reusing multiple large preexisting plans introduces even more evaluation overhead. In our other work, we present an approach for proactively generating repertoires that contain a diverse base of planning knowledge to utilize during replanning, as well as analyses for extracting reusable planning components that are likely to generalize to future situations and are cost effective to evaluate.

Search-based software engineering: The field of Search-Based Software Engineering (SBSE) uses meta-heuristic and stochastic search to solve multiple software engineering problems [20].

There has been considerable recent success in reusing and improving existing programs [3], similar to the way we reuse and improve existing plans. Such methods have been applied to self-adaptive systems and architectural design and evolution [9, 44], set configuration parameters for systems with strict quality requirements [18], and improving self-adaptive system test cases [16]. However, such work does not directly tackle the problem of self-adaptive planning, or improve on previous plans, and the research space of SBSE as applied to such systems remains underinvestigated, despite its potential [21].

7 CONCLUSION

Future generation systems will operate in complex environments of change and uncertainty. Self-* systems lower the costs of operating in such conditions by autonomously adapting to change in pursuit of their quality objectives, and planning is an important component of these approaches. We propose to use stochastic search to deal with unexpected change scenarios, specifically by reusing or building upon prior knowledge. Our GP planner can handle a very large search space (over 10^{37} possible states), can produce plans to within 0.05% of optimal, and can effectively plan with respect to multiple system quality objectives.

Our core results demonstrate the feasibility of reusing past knowledge in unexpected adaptation scenarios, and that the nature of both the scenario and of that prior knowledge influences its effectiveness. While naïvely reusing existing plans can actually result in worse performance than planning from scratch, effectively utilizing prior plans can reduce the number of generations required to reach a good plan for various scenarios, but whether this translates into runtime savings depends on both the size of the starting plan and its relationship to the new scenario. Our diversity analysis corroborates these results. The DART case study also shows how search spaces with large coarse regions particularly stand to benefit from plan reuse.

There exist several limitations and threats to the validity of our results. First, our parameter tuning is heuristic, performing a coarse test of one set of parameters before a finer-grained sweep; it is possible that the best configuration is located in an area that seemed less promising initially. Additionally, while we took care in our implementation of the PRISM MDP planner, mistakes in our implementation could affect our results. We mitigate the risks of bias in our Java GP planner representation by releasing it publicly for review and replication.

Additionally, our results may not generalize to other systems, or to other unexpected adaptation scenarios. We mitigate this risk by building our evaluation on two case-study systems from different domains, a cloud system used to assess prior work [43], and a team of autonomous aerial vehicles [22], both designed to approximate real-world systems. We constructed our evaluation scenarios to sample as much of the space of possible unanticipated adaptation needs as possible, and leave the investigation (or even a taxonomization) of additional such scenarios to future work. Other future research directions include autonomously determining when to replan, and investigating how the system model can be kept up to date with reality.

As systems become larger and more complex, the difficulty of planning for the unexpected will only increase. We show that knowledge reuse is a promising tool for addressing unexpected changes, while being underexplored in the self-* context. Future work in plan reuse in self-* systems has the potential to enable the next generation of autonomous systems to quickly respond to changes unforeseen at design time.

ACKNOWLEDGMENTS

We acknowledge Zack Coker and Jiacheng Wang for their contributions to the work that this journal article extends.

REFERENCES

[1] José Luis Ambite and Craig A. Knoblock. 2001. Planning by rewriting. *J. Artif. Int. Res.* 15, 1 (2001), 207–261.

[2] Jeffrey M. Barnes, David Garlan, and Bradley Schmerl. 2014. Evolution styles: Foundations and models for software architecture evolution. *Softw. Syst. Model.* 13, 2 (May 2014), 649–678. DOI: https://doi.org/10.1007/s10270-012-0301-9

[3] Earl T. Barr, Mark Harman, Yue Jia, Alexandru Marginean, and Justyna Petke. 2015. Automated software transplantation. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'15)*. 257–269. DOI: https://doi.org/10.1145/2771783.2771796

[4] Micheal Beetz and Drew McDermott. 1994. Improving robot plans during their execution. In *Proceedings of the International Conference on AI Planning and Scheduling (AIPS'94)*. 7–12.

[5] Javier Cámara, David Garlan, Bradley Schmerl, and Ashutosh Pandey. 2015. Optimal planning for architecture-based self-adaptation via model checking of stochastic games. In *Proceedings of the Symposium on Applied Computing (SAC'15)*. 428–435.

[6] Gerardo Canfora, Massimiliano Di Penta, Raffaele Esposito, and Maria Luisa Villani. 2005. An approach for QoS-aware service composition based on genetic algorithms. In *Proceedings of the Conference on Genetic and Evolutionary Computation (GECCO'05)*. 1069–1075.

[7] Tao Chen, Ke Li, Rami Bahsoon, and Xin Yao. 2018. FEMOSAA: Feature-guided and knee-driven multi-objective optimization for self-adaptive software. *ACM Trans. Softw. Eng. Methodol.* 27, 2, Article 5 (June 2018), 50 pages.

[8] Tao Chen, Miqing Li, and Xin Yao. 2019. Standing on the shoulders of giants: Seeding search-based multi-objective optimization with prior knowledge for software service composition. *Inf. Softw. Technol.* 114 (2019), 155–175.

[9] Betty H. C. Cheng, Andres J. Ramirez, and Philip K. McKinley. 2013. Harnessing evolutionary computation to enable dynamically adaptive systems to manage uncertainty. In *Proceedings of the Workshop on Combining Modelling and Search-Based Software Engineering (CMSBSE'13)*.

[10] Shang-Wen Cheng and David Garlan. 2012. Stitch: A language for architecture-based self-adaptation. *J. Syst. Softw.* 85, 12 (2012), 2860–2875.

[11] Zack Coker, David Garlan, and Claire Le Goues. 2015. SASS: Self-adaptation using stochastic search. In *Proceedings of the International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'15)*. 168–174.

[12] Rogério de Lemos et al. 2013. *Software Engineering for Self-Adaptive Systems: A Second Research Roadmap*. Springer, Berlin, 1–32.

[13] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Trans. Evol. Comput.* 6, 2 (2002), 182–197.

[14] John M. Ewing and Daniel A. Menascé. 2014. A meta-controller method for improving runtime self-architecting in SOA systems. In *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering (ICPE'14)*. 173–184.

[15] Erik Fredericks, Ilias Gerostathopoulos, Christian Krupitzer, and Thomas Vogel. 2019. Planning as optimization: Dynamically discovering optimal configurations for runtime situations. In *IEEE 13th International Conference on Self-Adaptive and Self-Organizing Systems (SASO'19)*. 1–10. DOI: https://doi.org/10.1109/SASO.2019.00010

[16] Erik M. Fredericks, Byron DeVries, and Betty H. C. Cheng. 2014. Towards run-time adaptation of test cases for self-adaptive systems in the face of uncertainty. In *Proceedings of the International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'14)*. 17–26.

[17] Simos Gerasimou, Radu Calinescu, and Giordano Tamburrelli. 2018. Synthesis of probabilistic models for quality-of-service software engineering. *Automat. Softw. Eng.* 25, 4 (2018), 785–831.

[18] S. Gerasimou, G. Tamburrelli, and R. Calinescu. 2015. Search-based synthesis of probabilistic models for quality-of-service software engineering. In *Proceedings of the International Conference on Automated Software Engineering (ASE'15)*. 319–330.

[19] Alicia Grech and Julie Main. 2004. *Case-Base Injection Schemes to Case Adaptation Using Genetic Algorithms*. Springer, Berlin, 198–210.

[20] Mark Harman. 2007. The current state and future of search based software engineering. In *Proceedings of the International Conference on Software Engineering*. 342–357. DOI: https://doi.org/10.1109/FOSE.2007.29

[21] Mark Harman, Yue Jia, William B. Langdon, Justyna Petke, Iman Hemati Moghadam, Shin Yoo, and Fan Wu. 2014. Genetic improvement for adaptive software engineering (keynote). In *Proceedings of the International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'14)*. 1–4.

[22] Scott A. Hissam, Sagar Chaki, and Gabriel A. Moreno. 2015. High assurance for distributed cyber physical systems. In *Proceedings of the 2015 European Conference on Software Architecture Workshops (ECSAW'15)*. ACM, New York, NY. DOI: https://doi.org/10.1145/2797433.2797439

[23] Yanrong Hu and S. X. Yang. 2004. A knowledge based genetic algorithm for path planning of a mobile robot. In *Robotics and Automation*, Vol. 5. 4350–4355. DOI: https://doi.org/10.1109/ROBOT.2004.1302402

[24] Pooyan Jamshidi, Amir M. Sharifloo, Claus Pahl, Andreas Metzger, and Giovani Estrada. 2015. Self-learning cloud controllers: Fuzzy q-learning for knowledge evolution. In *Proceedings of the International Conference on Cloud and Autonomic Computing (ICCAC'15)*. IEEE, 208–211.

[25] Leslie Pack Kaelbling, Michael L. Littman, and Anthony R. Cassandra. 1998. Planning and acting in partially observable stochastic domains. *Arif. Intell.* 101, 1 (1998), 99–134.

[26] Jeffrey O. Kephart and David M. Chess. 2003. The vision of autonomic computing. *Computer* 36, 1 (2003), 41–50.

[27] Narges Khakpour, Saeed Jalili, Carolyn Talcott, Marjan Sirjani, and Mohammadreza Mousavi. 2012. Formal modeling of evolving self-adaptive systems. *Sci. Comput. Program.* 78, 1 (2012), 3–26.

[28] Dongsun Kim and Sooyong Park. 2009. Reinforcement learning-based dynamic adaptation planning method for architecture-based self-managed software. In *Proceedings of the ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'09)*. IEEE, 76–85.

[29] Cody Kinneer, Zack Coker, Jiacheng Wang, David Garlan, and Claire Le Goues. 2018. Managing uncertainty in self-adaptive systems with plan reuse and stochastic search. In *Proceedings of the 13th International Conference on Software Engineering for Adaptive and Self-Managing Systems*. ACM, 40–50.

[30] Cody Kinneer, Rijnard Van Tonder, David Garlan, and Claire Le Goues. 2020. Building reusable repertoires for stochastic self-* planners. In *Proceedings of the 2020 IEEE Conference on Autonomic Computing and Self-organizing Systems (ACSOS'20)*.

[31] Cristian Klein, Martina Maggio, Karl-Erik A. Arzén, and Francisco Hernández-Rodriguez. 2014. Brownout: Building more robust cloud applications. In *Proceedings of the International Conference on Software Engineering (ICSE'14)*. 700–711.

[32] John R. Koza. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA.

[33] M. Kwiatkowska, G. Norman, and D. Parker. 2011. PRISM 4.0: Verification of probabilistic real-time systems. In *Proceedings of the International Conference on Computer Aided Verification (CAV'11)*. 585–591.

[34] Neal Lesh, Nathaniel Martin, and James Allen. 1998. Improving big plans. In *Proceedings of the Conference on Artificial Intelligence/Innovative Applications of Artificial Intelligence (AAAI/IAAI'98)*. 860–867.

[35] S. J. Louis and J. McDonnell. 2004. Learning with case-injected genetic algorithms. *Trans. Evol. Comp* 8, 4 (2004), 316–328. DOI: https://doi.org/10.1109/TEVC.2004.823466

[36] Sushil J. Louis and Gregory J. E. Rawlins. 1992. Syntactic analysis of convergence in genetic algorithms. In *Foundations of Genetic Algorithms 2*. Morgan Kaufmann, 141–151.

[37] Mausam and Andrey Kolobov. 2012. Planning with Markov decision processes: An AI perspective. *Synth. Lect. Artif. Intell. Mach. Learn.* 6, 1 (2012), 1–210. DOI: https://doi.org/10.2200/S00426ED1V01Y201206AIM017

[38] David J. Montana. 1995. Strongly typed genetic programming. *Evol. Comput.* 3, 2 (1995), 199–230.

[39] Gabriel Moreno, Cody Kinneer, Ashutosh Pandey, and David Garlan. 2019. DARTSim: An exemplar for evaluation and comparison of self-adaptation approaches for smart cyber-physical systems. In *Proceedings of the 2019 IEEE/ACM 14th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'19)*. IEEE, 181–187.

[40] Gabriel A. Moreno, Javier Cámara, David Garlan, and Bradley Schmerl. 2015. Proactive self-adaptation under uncertainty: A probabilistic model checking approach. In *Proceedings of the European Software Engineering Conference and the Symposium on the Foundation of Software Engineering (ESEC/FSE'15)*. 1–12.

[41] Gabriel A. Moreno, Javier Cámara, David Garlan, and Bradley Schmerl. 2018. Flexible and efficient decision-making for proactive latency-aware self-adaptation. *ACM Trans. Auton. Adapt. Syst.* 13, 1, Article 3 (Apr. 2018), 36 pages. DOI: https://doi.org/10.1145/3149180

[42] Héctor Muñoz-Avila and Michael T. Cox. 2008. Case-based plan adaptation: An analysis and review. *IEEE Intell. Syst.* 23, 4 (2008), 75–81. DOI: https://doi.org/10.1109/MIS.2008.59

[43] Ashutosh Pandey, Gabriel A Moreno, Javier Cámara, and David Garlan. 2016. Hybrid planning for decision making in self-adaptive systems. In *Proceedings of the Internayional Conference on Self-Adaptive and Self-Organizing Systems (SASO'16)*. 12–16.

[44] Gustavo G. Pascual, Mónica Pinto, and Lidia Fuentes. 2013. Run-time adaptation of mobile applications using genetic algorithms. In *Proceedings of the International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'13)*. 73–82.

[45] Mateusz Pawlik and Nikolaus Augsten. 2015. Efficient computation of the tree edit distance. *ACM Trans. Database Syst.* 40, 1, Article 3 (2015), 40 pages. DOI: https://doi.org/10.1145/2699485

[46] Riccardo Poli, William B. Langdon, Nicholas F. McPhee, and John R. Koza. 2008. *A Field Guide to Genetic Programming*. Lulu.com.

[47] Andres J. Ramirez, Betty H. C. Cheng, Philip K. McKinley, and Benjamin E. Beckmann. 2010. Automatically generating adaptive logic to balance non-functional tradeoffs during reconfiguration. In *Proceedings of the International Conference on Autonomic Computing (ICAC'10)*. 225–234.

[48] V. Roberge, M. Tarbouchi, and G. Labonte. 2013. Comparison of parallel genetic algorithm and particle swarm optimization for real-time UAV path planning. *IEEE Trans. Industr. Inf.* 9, 1 (2013), 132–141. DOI: https://doi.org/10.1109/TII.2012.2198665

[49] Amir Molzam Sharifloo, Andreas Metzger, Clément Quinton, Luciano Baresi, and Klaus Pohl. 2016. Learning and evolution in dynamic software product lines. In *Proceedings of the 11th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'16)*. ACM, New York, NY, 158–164. DOI: https://doi.org/10.1145/2897053.2897058

[50] E. L. Da Silva, H. A. Gil, and J. M. Areiza. 2000. Transmission network expansion planning under an improved genetic algorithm. *IEEE Trans. Power Syst.* 15, 3 (2000), 1168–1174.

[51] Tyron Stading, Petros Maniatis, and Mary Baker. 2002. Peer-to-peer caching schemes to address flash crowds. In *Proceedings of the International Workshop on Peer-to-Peer Systems (IPTPS'02)*. 203–213.

[52] Daniel Sykes, William Heaven, Jeff Magee, and Jeff Kramer. 2007. Plan-directed architectural change for autonomous systems. In *Proceedings of the Conference on Specification and Verification of Component-based Systems (SAVCBS'07)*. 15–21.

[53] Leonardo Trujillo. 2011. Genetic programming with one-point crossover and subtree mutation for effective problem solving and bloat control. *Soft. Comput.* 15, 8 (2011), 1551–1567.

[54] Manuela M. Veloso. 1994. Flexible strategy learning: Analogical replay of problem solving episodes. In *Proceedings of the National Conference on Artificial Intelligence (AAAI'94)*. 595–600.

[55] Thomas Vogel and Holger Giese. 2010. Adaptation and abstract runtime models. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'10)*. ACM, New York, NY, 39–48. DOI: https://doi.org/10.1145/1808984.1808989

[56] Eckart Zitzler, Marco Laumanns, and Lothar Thiele. 2001. SPEA2: Improving the strength Pareto evolutionary algorithm. Technical Report. Eidgenössische Technische Hochschule Zürich (ETH).

[57] Parisa Zoghi, Mark Shtern, Marin Litoiu, and Hamoun Ghanbari. 2016. Designing adaptive applications deployed on cloud environments. *Trans. Auton. Adapt. Syst.* 10, 4, Article 25 (2016), 26 pages.

Received December 2018; revised September 2020; accepted November 2020