Managing Uncertainty in Self-Adaptive Systems with Plan Reuse and Stochastic Search

Cody Kinneer
School of Computer Science,
Carnegie Mellon University
Pittsburgh, PA
ckinneer@cs.cmu.edu

Zack Coker
School of Computer Science,
Carnegie Mellon University
Pittsburgh, PA
zfc@cs.cmu.edu

Jiacheng Wang
Computer Science Department,
Dickinson College
Carlisle, PA
wangjia@dickinson.edu

David Garlan
School of Computer Science,
Carnegie Mellon University
Pittsburgh, PA
garlan@cs.cmu.edu

Claire Le Goues
School of Computer Science,
Carnegie Mellon University
Pittsburgh, PA
clegoues@cs.cmu.edu

ABSTRACT

Many software systems operate in environments where change and uncertainty are the rule, rather than exceptions. 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 typically must replan from scratch. We propose to address this problem by reusing prior planning knowledge to adapt in the face of unexpected situations. We present a planner based on genetic programming that reuses existing plans. While reuse of material in genetic algorithms has recently applied successfully in the area of automated program repair, we find that naïvely reusing existing plans for self-* planning actually results in a loss of utility. Furthermore, we propose a series of techniques to lower the costs of reuse, allowing genetic techniques to leverage existing information to improve planning utility when replanning for unexpected changes.

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*;

KEYWORDS

plan reuse, self-* systems, planning, uncertainty, genetic programming, cloud services

SEAMS '18, May 28–29, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5715-9/18/05…$15.00
https://doi.org/10.1145/3194133.3194145

ACM Reference Format:
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 *SEAMS '18: SEAMS '18: 13th International Symposium on Software Engineering for Adaptive and Self-Managing Systems , May 28–29, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, Article 4, 11 pages. https://doi.org/10.1145/3194133.3194145

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 self-adjust is by making run-time adjustments according to an adaptation strategy, or *plan*. Humans can proactively plan for various situations by hand at design time [8]. 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 adaptation strategies at run-time [18, 32, 43, 48].

While these systems can quickly respond to the changing conditions that they were designed for, they often struggle to handle unforeseen adaptation scenarios [10]. 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 [42]). Even expensive human generated plans [8] cannot handle this challenge, requiring expensive replanning post design time in the face of unanticipated changes.

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, 45] and case-based reasoning [14, 28, 33] 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 change. 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 [9] 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, 11, 17, 38, 39, 41]. However, these approaches do not address the challenge of replanning for new, explicitly unforeseen contexts post-design time, which we propose to address through 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 from generation to generation, in effect reusing information from previous generations. This observation has been successfully applied in other domains, such as automated program repair [16]. We investigate the extent to which reusing existing plans in self-* planning can result in an improvement in the 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. Our main contribution is a series of techniques to support adapting to unexpected changes at runtime by lowering the costs of plan reuse during evolution, and an empirical study investigating the utility of plan reuse for several indicative change scenarios. Our key contributions are:

- 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 MDP planner [34] that shows the genetic programming planner can produce near-optimal plans (0.05% error in the single objective scenario and 9.4% in the multi-objective scenario).
- 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 fitness 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.

Our initial position [9] 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 [8], 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 [34], an experimental study investigating the trade-offs of plan reuse when adapting to unforeseen scenarios, and support for multi-objective search [47].

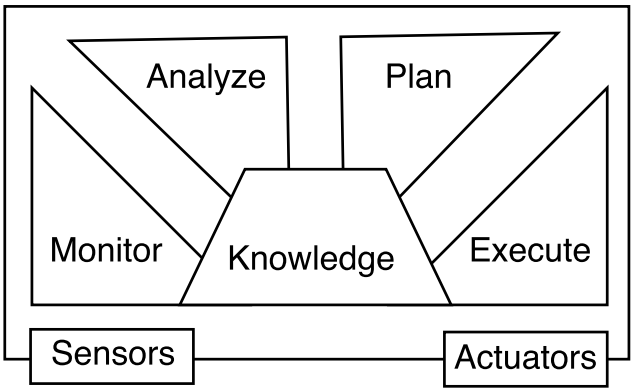

Figure 1: MAPE-K Loop for self-* systems.

The rest of the paper 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 [20], shown in Figure 1. We focus on the planning (P) component, which produces *strategies* 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 [26, 43], our approach is closest to Stitch [8]. An *online* planner [43] generates a plan at run-time, which can adapt quickly at a potential cost to optimality. An *offline* planner [8] precomputes strategies to handle common cases and then chooses between them at run-time. This allows for a fast, correct response to known or predicted situations, but cannot handle unanticipated adaptation needs. We compare to a previous hybrid online/offline approach [34], that relies in part on Markov decision processes (MDPs), formal models that can explicitly capture probabilistic behavior. Model checkers such as PRISM [26] 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 [24] (GP) 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 [37]. 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; and sub-optimal plans are often acceptable in real systems. Indeed, genetic algorithms have been successfully applied to self-* systems [6, 11, 38], 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 towards 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 [31], limiting exploration of some types of invalid solutions.

A problem-specific *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 trade-offs between several objectives. We use SPEA2 [47] to implement a multi-objective search, which selects a fixed quantity of non-dominated individuals to create the next generation.

3 RUNNING SCENARIO

We first present a system, adapted from prior work [34], that we use as a running example and in evaluation.

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 different types, 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 generated by current users, minus operating cost (corresponding to the number and cost of the servers), (2) *User latency*, or the mean time users 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.

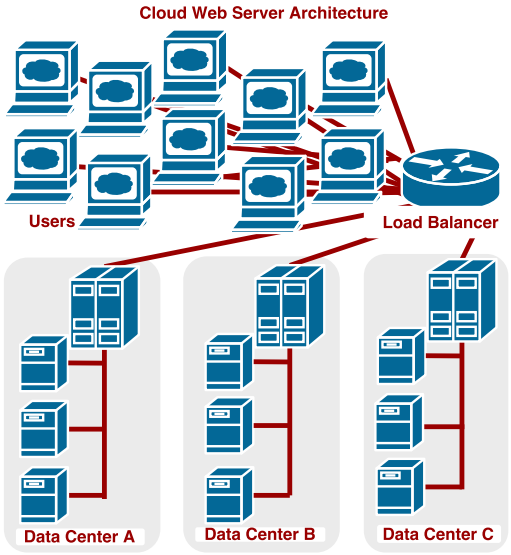

Figure 2: Cloud web server architecture.

For example, while the system uses ads for revenue, they increase latency. The system can remove ads to improve user experience and server load, while decreasing profit.

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 [23] on a per-data center basis). The dimmer level can be changed at 25% increments. At run-time, each of these adaptation tactics may fail. 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.

3.4 Post-design-time adaptation

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

⟨*plan*⟩ ::= '(' ⟨*operator*⟩ ')' | '(' ⟨*tactic*⟩ ')'

⟨*operator*⟩ ::= 'F' ⟨*int*⟩ ⟨*plan*⟩ (For loop)
| 'T' ⟨*plan*⟩ ⟨*plan*⟩ ⟨*plan*⟩ (Try-catch)
| ';' ⟨*plan*⟩ ⟨*plan*⟩ (Sequence)

⟨*tactic*⟩ ::= 'StartServer' ⟨*srv*⟩ | 'ShutdownServer' ⟨*srv*⟩
| 'IncreaseTraffic' ⟨*srv*⟩ | 'DecreaseTraffic' ⟨*srv*⟩
| 'IncreaseDimmer' ⟨*srv*⟩ | 'DecreaseDimmer' ⟨*srv*⟩

Figure 3: Grammar for specifying plans. 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) )
```

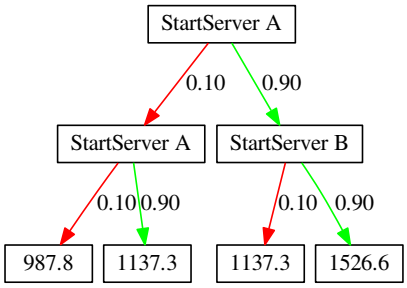

Figure 4: Top: An example plan. Bottom: This plan's system state tree.

4 APPROACH

We present a planner that reuses previously-known information (Section 4.4) 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. The mechanism for synchronizing the system model with the actual world is outside the scope of this paper; this may be done manually (likely with less effort than replanning), or automatically [21, 46]. 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.

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

4.1 Representation

Individuals in the population are plans organized as trees. Figure 3 gives a Backus-Naur grammar for our plans. Each consists of either one of six available *tactics* (described in Section 3), or one of three *operators* containing subplans. The `for` operator repeats the given subplan for 2–10 iterations; the `sequence` operator consecutively performs 2 subplans. The `try-catch` operator tries the first subplan. If the last tactic in that subplan fails, 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, 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 [8]. Unlike Stitch, our language does not consider plan applicability (guards that test state to determine when a plan can be used), which we leave 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 [26] plan 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 [44] selects a subtree in each of two parent plans (selected via tournament selection) 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 which 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.

4.3 Fitness

We evaluate candidate fitness by simulating the plan to measure the expected quality of the resulting system. 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 product of its edge probabilities.

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. Each leaf in the state tree represents a timeline of events (parent tactics succeeding or failing). This timeline is simulated to obtain the fitness accrued while the plan was executing as well as the fitness 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 fitness resulting from the provided plan. If the system will remain in a state for a long period of time, it may be worthwhile to spend more time planning since the system has more time to realize gains from the planning effort. On the other hand, 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. Total fitness accrued is equal to $s*p+d+a*(w-(t+p))$, where s is the system's initial fitness, p is the planning time, d is the fitness accrued during plan execution, a is the fitness value after the plan is executed, w is the *window size*, t is the time plan's execution time.

We investigated several additional heuristic modifications to fitness computation to manage invalid actions and plan size. First, we investigate an *invalid action penalty* per invalid tactic. To manage produced plan size, we include a *verboseness penalty*, which penalizes a plan proportional to its size; and a *parsimony pressure kill ratio*, which assigns a fitness of zero to a random proportion of individuals larger than the average population size. In reporting final plan fitnesses, we report actual fitness, unmodified by penalties.

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 adaptation strategies is created. These strategies 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. Seeding previously useful plans into the population allows for useful pieces of planning knowledge to spread to other plans during crossover.

Preliminary results showed that initializing the search by naïvely copying existing plans did not result in efficient planning, and in most cases was inferior compared to replanning from scratch with a randomly generated starting population. This is due to the high cost of calculating the fitness values of long starting plans. 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.

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

Lastly, 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 [25] 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.

5 EVALUATION

We built the genetic programming planner described in Section 4 on ECJ.[1] This section describes our evaluation. We investigated the following research questions:

(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 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, we evaluate on various scenarios based on the system shown in Figure 2 and described in Section 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, unless otherwise stated. We set the planning horizon to 20 for PRISM, and the maximum plan tree depth to 20 for the GP planner. Each experimental result is the median of 10 trial runs if randomness or timing is involved. Where statistical tests are used to asses 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 fitness objectives. 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

[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

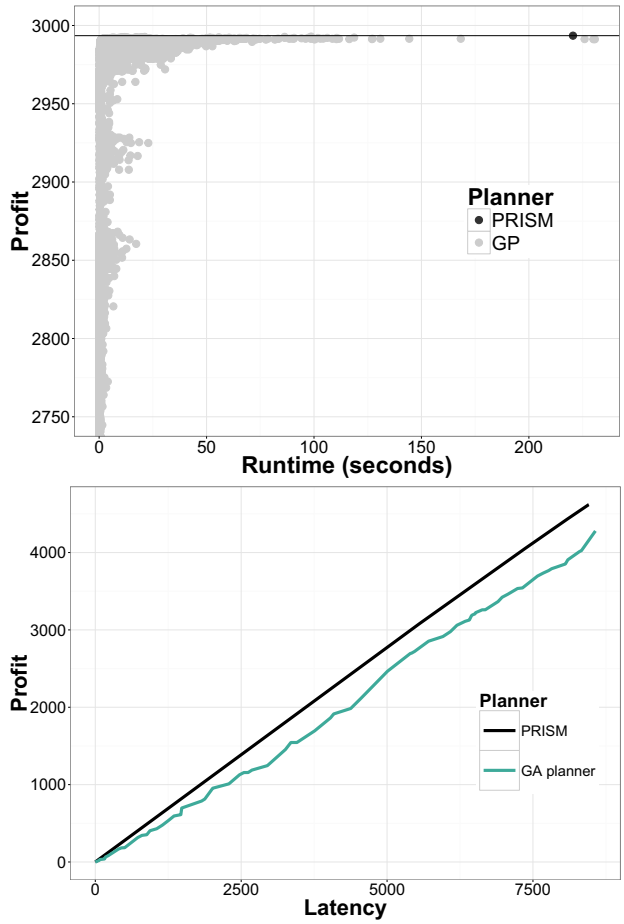

Figure 5: Top: Profit versus planning time for GP parameter configurations. Many configurations produce similar profit results to PRISM, significantly faster. Bottom: Pareto fronts for profit and latency from both planners.

is considered, we set the window size to be 10,000 seconds unless otherwise specified. Where we compare to searches from "scratch", we use ramped half-and-half[24] to initialize the population.

5.1 Comparative Study

5.1.1 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 [34], 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 particular environment state.

As in 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 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 minimal plan of four tactics that does not affect fitness.

The dark point at the top of Figure 5 shows the optimal system profit (fitness) and planning time (200 seconds) 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). The best configuration that produced plans in 0.50 seconds resulted in only 0.29% error, which demonstrates the 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.

5.1.2 Search space. Next, we evaluate and compare the planners' search space limitations. We varied search space size by adjusting the number of available server types (t) in our scenario, which caused the model states to grow exponentially following the equation $(6\ servers\_per\_type * 5\ possible\_dimmer\_values * 5\ possible\_traffic\_values)^t$.

We found that PRISM can plan to maximize profit for 3 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 run-time, 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.

5.1.3 Multi-objective search. The GP planner can create a Pareto frontier of plans to trade-off between multiple quality attributes, allowing system maintainers to evaluate the best possible combinations. PRISM can also generate a Pareto frontier for two objectives. The bottom of Figure 5 shows the Pareto fronts for the profit and latency objectives produced by PRISM and the GP. 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 seconds; 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

[2]Because the Pandey et al. approach [34] 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 paper.

Table 1: Improvement obtained by reuse enabling techniques.

Planning Technique	Utility	P Value
Scratch	1.000	
Scratch & *kill_ratio*	1.044	< 0.01
Reuse	0.962	0.06
Reuse & *kill_ratio*	1.072	< 0.01
Reuse & *kill_ratio* & *scratch_ratio*	1.077	0.63
Reuse & *kill_ratio* & *scratch_ratio* & trimmer	1.112	< 0.01

front. If we begin with plans previously optimized for profit, 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 trade-offs between latency and quality. We explore the trade-offs of plan reuse more directly in the next set of experiments.

5.2 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 introduce 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 10000, 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 used were *kill_ratio* = 0.75 and *scratch_ratio* = 0.5. These values were chosen based on a parameter sweep.

Table 1 shows the results, normalized to the utility of planning from scratch without the *kill_ratio*, such that this utility is 1. Using the *kill_ratio* improved utility to 1.044. Without any reuse-enabling techniques, reusing plans by initializing the population with mutated versions of the starting plan resulted in a fitness 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 makes improving planning fitness through reuse nontrivial, the proposed enhancements to GP planning can reduce this cost and achieve higher fitness than planning from scratch.

Table 2: Percent change reusing plans instead of planning from scratch. Statistically significant results ($P < 0.05$) are shown in bold font.

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**

5.3 Unforeseen Adaptation Scenarios

Our core experiments investigate the GP planner's ability to address unforeseen adaptation needs with plan reuse. We do this by constructing adaptation scenarios that cover different types of adaptation needs based on different sources of uncertainty, and assessing the planner's ability to respond when planning with reuse compared to planning from scratch. The considered scenarios are:

- **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 + New Data Center.** The system experiences a major spike in traffic, and gains access to a new server location to help address it. 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 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*.

For each adaptation 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 ten 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.

Table 2 shows the percent 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

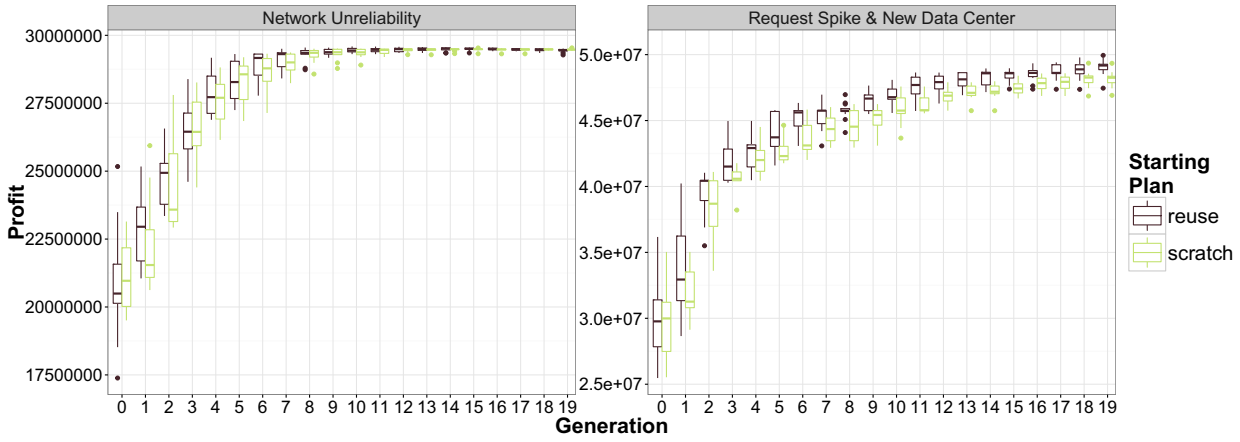

Figure 6: Profit versus generation for two scenarios.

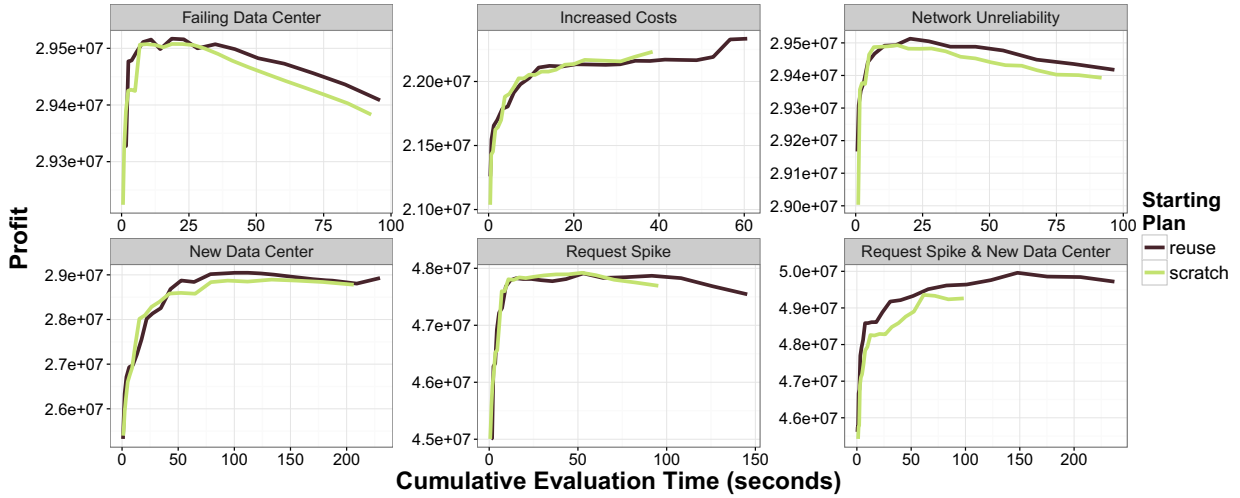

Figure 7: Profit versus cumulative runtime for all six scenarios.

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 fitness improvements.

5.3.1 Modified System or *Tactics.* Figure 6 shows the profit over generation produced by the GP for two of the considered scenarios. The left of Figure 6 shows results for the Network Unreliability scenario. For much of the first seven generations of planning, plan reuse outperforms planning from scratch, with the two eventually converging to the same fitness at generation eight. The Increased Costs and Failing Data Center scenarios showed similar results.

5.3.2 Changing Environment. The right of Figure 6 shows results 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 [42]). We also provide the system with a new data center, D, to possibly use to address this issue.[3] This scenario shows the most pronounced differences between plan reuse and planning from scratch, with plan reuse performing better for all 20 generations. The Request Spike and New Data Center scenarios alone showed a similar pattern but was less pronounced.

5.3.3 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 imperfect proxy for run time. Thus, Figure 7 shows results in terms of wall-clock time for each scenario. 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

[3]We also evaluated planning in response to the two changes independently. Both scenarios showed similar results to the Request Spike + New Data Center, although the benefits of plan reuse are more prominent in the hybrid scenario.

scenarios showed greater differences, in particular the Request Spike + New Data Center scenario showed a clear advantage to reusing existing plans.

5.4 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* [29], 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 algorithm [36].

Figure 8 shows population diversity across the scenarios. Diversity values from planning from scratch, as well as reusing plans both with and without trimming are shown. Planning from scratch produces a highly diverse starting population that gradually becomes less diverse as it converges.

As shown in Figure 8, 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 trade-off.

6 RELATED WORK

Planning: The artificial intelligence (AI) community has produced a considerable body of research in planning, including but not limited to probabilistic approaches like MDP [19, 30]. 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 [45]; 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 [27]; or iteratively transforming simple plans to produce complex plans [1]. Case-based plan adaption [33] explicitly reuses past plans in new contexts, in which context GAs have been explored directly [14, 28], 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 [18, 22]. 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 [40], or evolving models at runtime [46].

Self-* planning: Mutliple PRISM MDP planners appear in the self-* literature [5, 32, 34]. These techniques are typically offline, as is manual human planning [8], 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 [43] or using hill-climbing [48] (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 [38] 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 adaptation strategies, we focus explicitly on the utility of alternative starting strategies in the face of unanticipated scenarios. We also compare a GP planner to an optimal planner. 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 [11]. This prior work broadly substantiates the utility of stochastic algorithms in an self-* planning context, but otherwise focuses on very different domains than we do.

In our own previous work [9], we demonstrated the feasibility of plan reuse using genetic programming in self-* contexts. We expand upon that initial concept in the ways described in Section 1.

Search-based software engineering: The field of Search-Based Software Engineering (SBSE) uses meta-heuristic and stochastic search to solve multiple software engineering problems [15]. 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 [7, 35], set configuration parameters for systems with strict quality requirements [13], and improving self-adaptive system test cases [12]. 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 [16].

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

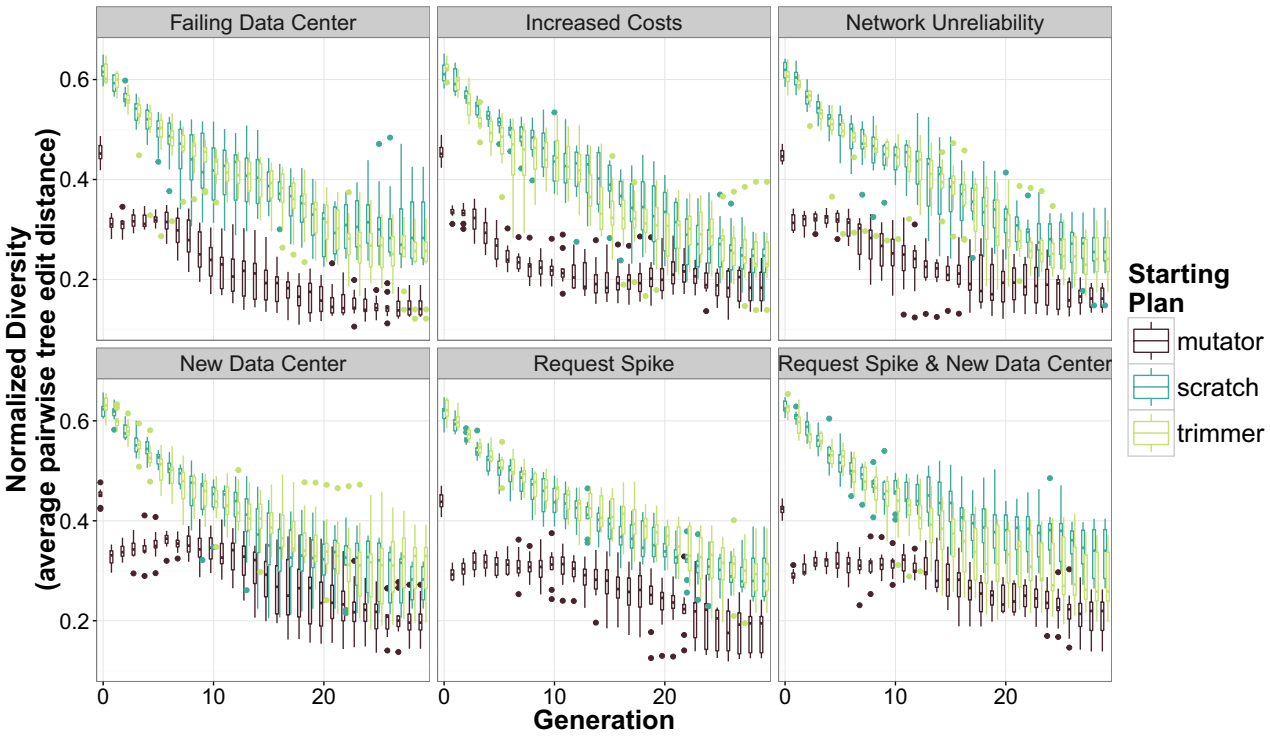

Figure 8: Diversity versus generation for all six scenarios.

of these approaches. We propose to use stochastic search to deal with unexpected adaptation strategies, 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 run-time savings depends on both the size of the starting plan and its relationship to the new scenario. Our diversity analysis corroborates these results.

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 a cloud system used to assess prior work [34], and designed to approximate real-world systems, such as an application running on Amazon's Cloud Web Services. 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.

ACKNOWLEDGEMENTS

This research supported in part by the National Science Foundation (CCF-1618220) and the National Science Foundation Graduate Research Fellowship Program (DGE1745016). 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.

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. 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 *Int. Symp. on Soft. Testing and Analysis (ISSTA)*. 257–269. https://doi.org/10.1145/2771783.2771796

[4] Micheal Beetz and Drew McDermott. 1994. Improving Robot Plans During Their Execution. In *Int. Conf. on AI Planning and Scheduling (AIPS)*. 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 *Symp. 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 *Conf. on Genetic and Evol. Computation (GECCO)*. 1069–1075.

[7] Betty H. C. Cheng, Andres J. Ramirez, and Philip K. McKinley. 2013. Harnessing Evolutionary Computation to Enable Dynamically Adaptive Systems to Manage Uncertainty. In *Workshop on Combining Modelling and Search-Based Soft. Eng. (CMSBSE)*.

[8] Shang-Wen Cheng and David Garlan. 2012. Stitch: A Language for Architecture-based Self-adaptation. *J. Syst. Softw.* 85, 12 (2012), 2860–2875.

[9] Zack Coker, David Garlan, and Claire Le Goues. 2015. SASS: Self-adaptation Using Stochastic Search. In *Int. Symp. on Soft. Eng. for Adaptive and Self-Managing Syst. (SEAMS)*. 168–174.

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

[11] John M. Ewing and Daniel A. Menascé. 2014. A Meta-controller Method for Improving Run-time Self-architecting in SOA Systems. In *5th ACM/SPEC Int. Conf. on Performance Eng. (ICPE '14)*. 173–184.

[12] 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 *Int. Symp. on Soft. Eng. for Adaptive and Self-Managing Syst. (SEAMS)*. 17–26.

[13] S. Gerasimou, G. Tamburrelli, and R. Calinescu. 2015. Search-Based Synthesis of Probabilistic Models for Quality-of-Service Software Engineering. In *Int. Conf. on Automated Soft. Eng. (ASE'15)*. 319–330.

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

[15] Mark Harman. 2007. The Current State and Future of Search Based Software Engineering. In *Int. Conf. on Soft. Eng.* 342–357. https://doi.org/10.1109/FOSE.2007.29

[16] 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 *Int. Symp. on Soft. Eng. for Adaptive and Self-Managing Syst. (SEAMS)*. 1–4.

[17] 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. https://doi.org/10.1109/ROBOT.2004.1302402

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

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

[20] Jeffrey O. Kephart and David M. Chess. 2003. The Vision of Autonomic Computing. *Computer* 36, 1 (2003), 41–50.

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

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

[23] Cristian Klein, Martina Maggio, Karl-Erik AArzén, and Francisco Hernández-Rodriguez. 2014. Brownout: Building More Robust Cloud Applications. In *Int. Conf. on Soft. Eng. (ICSE '14)*. 700–711.

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

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

[26] M. Kwiatkowska, G. Norman, and D. Parker. 2011. PRISM 4.0: Verification of Probabilistic Real-time Systems. In *Int. Conf. on Computer Aided Verification (CAV '11)*. 585–591.

[27] Neal Lesh, Nathaniel Martin, and James Allen. 1998. Improving Big Plans. In *Conf. on Artificial Intelligence/Innovative Applications of Artificial Intelligence (AAAI/IAAI)*. 860–867.

[28] S. J. Louis and J. McDonnell. 2004. Learning with Case-injected Genetic Algorithms. *Trans. Evol. Comp* 8, 4 (2004), 316–328. https://doi.org/10.1109/TEVC.2004.823466

[29] Sushil J. Louis and Gregory J. E. Rawlins. 1992. Syntactic Analysis of Convergence in Genetic Algorithms. In *Found. of Genetic Algorithms 2*. Morgan Kaufmann, 141–151.

[30] Mausam and Andrey Kolobov. 2012. Planning with Markov Decision Processes: An AI Perspective. *Synthesis Lectures on Artificial Intelligence and Machine Learning* 6, 1 (2012), 1–210. https://doi.org/10.2200/S00426ED1V01Y201206AIM017

[31] David J. Montana. 1995. Strongly Typed Genetic Programming. *Evol. Comput.* 3, 2 (1995), 199–230.

[32] Gabriel A. Moreno, Javier Cámara, David Garlan, and Bradley Schmerl. 2015. Proactive Self-adaptation Under Uncertainty: A Probabilistic Model Checking Approach. In *European Softw. Eng. Conf. and the Symp. on the Found. of Soft. Eng. (ESEC/FSE 15)*. 1–12.

[33] H. MuÃśoz-Avila and M. T. Cox. 2008. Case-Based Plan Adaptation: An Analysis and Review. *IEEE Intelligent Syst.* 23, 4 (2008), 75–81. https://doi.org/10.1109/MIS.2008.59

[34] Ashutosh Pandey, Gabriel A Moreno, Javier Cámara, and David Garlan. 2016. Hybrid Planning for Decision Making in Self-adaptive Systems. In *Int. Conf. on Self-Adaptive and Self-Organizing Syst. (SASO '16)*. 12–16.

[35] Gustavo G. Pascual, Mónica Pinto, and Lidia Fuentes. 2013. Run-time Adaptation of Mobile Applications Using Genetic Algorithms. In *Int. Symp. on Soft. Eng. for Adaptive and Self-Managing Syst. (SEAMS)*. 73–82.

[36] Mateusz Pawlik and Nikolaus Augsten. 2015. Efficient Computation of the Tree Edit Distance. *ACM Trans. Database Syst.* 40, 1, Article 3 (2015), 40 pages. https://doi.org/10.1145/2699485

[37] Riccardo Poli, William B Langdon, Nicholas F McPhee, and John R Koza. 2008. *A field guide to genetic programming.* Lulu.com.

[38] 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 *Int. Conf. on Autonomic Computing (ICAC)*. 225–234.

[39] 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. on Industrial Informatics* 9, 1 (2013), 132–141. https://doi.org/10.1109/TII.2012.2198665

[40] 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, USA, 158–164. https://doi.org/10.1145/2897053.2897058

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

[42] Tyron Stading, Petros Maniatis, and Mary Baker. 2002. Peer-to-Peer Caching Schemes to Address Flash Crowds. In *Int. Workshop on Peer-to-Peer Systems (IPTPS '02)*. 203–213.

[43] Daniel Sykes, William Heaven, Jeff Magee, and Jeff Kramer. 2007. Plan-directed Architectural Change for Autonomous Systems. In *Conf. on Specification and Verification of Component-based Syst. (SAVCBS '07)*. 15–21.

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

[45] Manuela M. Veloso. 1994. Flexible Strategy Learning: Analogical Replay of Problem Solving Episodes. In *Nat. Conf. on Artificial Intelligence (AAAI '94)*. 595–600.

[46] 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, USA, 39–48. https://doi.org/10.1145/1808984.1808989

[47] Eckart Zitzler, Marco Laumanns, and Lothar Thiele. 2001. *SPEA2: Improving the Strength Pareto Evolutionary Algorithm.* Technical Report.

[48] 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.