# Search-based Plan Reuse in Self-* Systems

**Cody Kinneer**

CMU-ISR-21-104

May 2021

Institute for Software Research
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

**Thesis Committee:**
Claire Le Goues, Co-chair
David Garlan, Co-chair
Fei Fang
Betty Cheng (Michigan State University)

*Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Software Engineering.*



This research supported in part by the National Science Foundation (CCF-1618220). This material is based upon work supported by the NSA under Award No. H9823018D0008. This research supported in part by a grant from the CyLab Security and Privacy Institute at Carnegie Mellon University. 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.

**Keywords:** self-*, planning, uncertainty, reuse, evolutionary computation

## Abstract

Increasingly software systems operate in environments of change and uncertainty, where the system's ability to satisfy its quality objectives depends on its ability to adapt. Self-adaptation allows these systems to manage this challenge by autonomously adapting to changes in their environments. While self-* systems are designed precisely to manage uncertainty, unexpected changes may violate design assumptions, resulting in the system failing to satisfy its quality attribute requirements. When this occurs, the planner must generate a new plan, an expensive operation for large systems. As autonomous systems increase in size, interconnectedness, and complexity, this cost can quickly become prohibitive.

This thesis addresses this problem by leveraging information contained in prior plans to reduce the replanning necessary to respond to an unexpected change. Even in the face of an unexpected change, some of the insights contained in existing plans are likely to remain applicable. For example, an autonomous aerial vehicle encountering an unexpected obstacle will need to replan to avoid the obstacle, but the drone may be able to return to its prior plan after this maneuver. A larger change will reduce the amount of reuse that is possible, for example changing the drone's mission to fly to a new location, but still, the takeoff and landing procedures may be reused. This thesis reuses existing adaptation plans by seeding a genetic algorithm with these plans. This enables a scalable self-* planner that can replan in complex systems with large search spaces.

While the idea of plan reuse is intuitive, in practice plan reuse is difficult and may even be worse than replanning from scratch if not performed carefully. This dissertation provides reuse enhancing approaches to reduce the evaluation time of candidate plans, an approach for building reusable repertoires of plans and identifying generalizable plan fragments, and a co-evolutionary extension to enable plan reuse for security. The thesis is evaluated on three simulated case study systems, including a cloud-based web service provider, a team of autonomous aerial vehicles, and an enterprise business system under a cyber attack. Ultimately, plan reuse will enable large self-* systems to replan even after unexpected changes.

## Acknowledgments

First I would like to thank my friends and colleagues that I met on my PhD journey. Their support and advice were crucial for the completion of this thesis. Next I would like to thank the faculty of ISR for creating a unique environment for research success, which promotes excellence by focusing on the quality of the work while avoiding the traps of becoming overly fixated on arbitrary metrics.

I would like to thank my committee members, Fei Fang and Betty Cheng. Fei's help enabled the work on security to be possible, and I thank her for teaching me what I know about game theory, pushing me on the formalisms and assumptions, and for her contributions to the Observable Eviction Game. I am also grateful for my conversations with Betty, who I can always count on to ask the tricky big picture questions. Her insights on the applicability of the approaches have made the thesis stronger.

I owe a great deal of thanks to my advisors, Claire Le Goues and David Garlan. Despite being very different from one another in most dimensions of my internal model of advisors, I benefited from their ability to reach a consensus on the key points of the research, and from their skill of knowing when to press me and when to let me go off on my own. I am grateful for David's insights on the big picture, our sometimes philosophical discussions on adaptive systems, and for his uncanny skill of sometimes crafting the perfect sentence to convey an idea after a few moments of reflection, ideas over which I have wrestled with longer than I would like to admit.

I thank Claire for our discussions on the more nuanced aspects of evolutionary computation and program analyses, and her guidance on balancing the high level story with the lower level implementation details. I am also grateful for the times we've spent editing papers together and her superhuman power over latex documents. To the extent that I have been successful in the PhD, this success can be primarily attributed to good advising.

Lastly, I thank my family for supporting me through this journey. I am especially grateful to my other half Liz, for supporting me through the ups and downs of this adventure.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Increasingly software systems operate in environments of change and uncertainty, where the system's ability to satisfy its quality objectives depends on its ability to adapt. Approaches for imbuing such systems with autonomic adaptation, often called self-healing, self-protecting, self-adaptive, or self-* systems, have been successful in allowing these systems to automatically respond to the changes in their environments [15, 41, 76].

Such adaptation is often enabled by a planner, which decides on a course of action in response to an environmental change, producing an adaptation strategy or plan. Planning in self-* systems may be done in either an online or offline setting, where online planners generate plans during run time [57], and offline planners [11] prepare a collection or repertoire of adaptation strategies beforehand, which are then selected from at run time. Despite progress in automated planning, many self-* systems in practice use manually constructed human written plans that seek to anticipate possible changes that the system may encounter and prescribe responses to them. In either case, the planner helps the system manage uncertainty by making decisions that take into account the capabilities of the system and the environment, including making trade offs between competing quality objectives like performance and cost.

Self-* planners must respond to a number of sources of uncertainty during their operation, such as noise in the system's sensors, stochastic behavior in the environment, failures in the system, etc. These uncertainties are often categorized as being *foreseen*, *unforeseen*, or *unforeseeable*. While self-* systems are designed precisely to manage uncertainty, unexpected, unforeseeable unknown unknowns may violate the assumptions for which the system was designed, resulting in the system failing to satisfy its quality attribute requirements. These unknowns unknowns include changes such as the addition or removal of available adaptation tactics, changes in

the effects of tactics, or changes to the system's quality objectives. For example, a cloud web server self-* system might be designed to start additional servers when demand increases; however this strategy will not be effective if the reserve servers are unavailable.

When an adaptive system confronts an unexpected change, the system, along with the planner, must *evolve* to continue performing well in the new situation. Evolution requires that the system's models are updated to reflect the new state of affairs following the unexpected change, and the planner must generate new plans. For human written plans, this necessitates an expensive replanning process. Even automated planning approaches must often generate new plans from scratch.

As adaptive systems grow larger, more connected, and more complex, the size of the search space for plan generation continues to increase. This makes generating new plans from scratch increasingly expensive. Plan *reuse* with stochastic search is a promising potential strategy for enabling future generation self-* systems to effectively evolve. Plans contain information about how the system should adapt in a particular context, and although an unexpected change may result in some of this information becoming incorrect, the previous plans may still encode usable knowledge that applies, and can be reused, after such a change occurs.

Stochastic search is useful when the search space is large and not well understood, and has shown promise in related domains such as reusing source code to automatically repair programs [18]. This is the case for planning in response to an unexpected change, since by its nature the search space cannot be known a priori. Prior work [13] proposed using genetic algorithms to facilitate plan reuse. Genetic algorithms are a natural choice for the problem since they operate by incrementally evolving existing members of a population, in effect, reusing information from previous generations to find better solutions in the next generations.

While genetic algorithms have been applied to planning in autonomous systems [10, 62, 63], applying these approaches to effectively reuse existing planning knowledge in self-* systems is nontrivial. Reusing plans can in fact be worse than replanning from scratch [54], and while genetic algorithms have been employed to reuse information for certain limited problem cases [17, 44], the self-* domain poses unique challenges that remain to be addressed, such as the many sources of uncertainty in these systems. An ideal planning approach should be able to replan quickly, be applicable to a broad range of unexpected changes, and be able to reason about adversarial interactions such as promoting security.

## 1.1 Thesis Statement

This thesis explores using stochastic search and knowledge reuse to address these challenges and improve the ability of self-* systems to effectively evolve, as expressed in the following thesis statement:

> *We can enable self-* systems with large state spaces to evolve in response to unexpected changes by reusing existing plans with stochastic search in the following three ways: (a) reusing existing plans using genetic programming and reuse enhancing approaches to reduce evaluation time, (b) building reusable repertoires by identifying generalizable plan fragments to build resilience against a wide range of unexpected scenarios, and (c) reusing strategies in adversarial settings.*

Plans are reused by seeding the genetic algorithm with existing plans or plan fragments, and evolved to improve their fitness after an unexpected change occurs. The first research thrust (a) addresses the problem that reusing plans requires candidate plans to be evaluated, which for self-* systems with complex models often requires a significant amount of time. This thrust investigates heuristic approaches for reducing the total amount of evaluation time needed for replanning, including reusing a percentage of individuals and initializing the remainder randomly, trimming long plans to obtain smaller plan fragments, and prematurely terminating the evaluation of the longest evaluating plans. The second thrust (b) focuses on the challenge that self-* systems must be robust to many kinds of unexpected changes, and investigates how plan reuse can be applied to build reusable repertoires of plans that improve the system's ability to adapt to a range of unexpected scenarios. This thrust uses ideas from program analysis, specifically clone detection, to identity commonly occurring plan fragments that are useful in a range of change scenarios, to assemble a repertoire of generalizable plans. The third thrust (c) investigates the unique challenges of plan reuse in adversarial domains, where one or more adversaries can also take actions that affect the system's utility, often negatively. This thrust explores how co-evolution can be applied to reuse plans in this environment.

## 1.2 Claims

The thesis is evaluated by the improvement in evolution ability compared to a baseline of planning from scratch in three case study systems, a cloud-based web server, a team of autonomous areal vehicles, and a security scenario inspired by a well documented data breach at the Target chain of stores [75, 38]. Evolution ability is measured by the number of generations until convergence to a good plan, the im-

provement in wall-clock time needed for replanning, and the range of unexpected change scenarios that the approach can address.

## 1.3 Contributions

The contributions of the thesis are the following:

1. An approach for plan reuse with stochastic search for more effective replanning following unexpected changes.
   (a) A planner using genetic programming and initial population seeding to support reusing existing adaptation strategies.
   (b) A collection of reuse enabling approaches to reduce the evaluation time of existing strategies to facilitate effective plan reuse.

   Publications:

   - Cody Kinneer, Zack Coker, Jiacheng Wang, David Garlan, and Claire Le Goues. 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*, pages 40–50. ACM, 2018
   - Cody Kinneer, David Garlan, and Claire Le Goues. Information reuse and stochastic search: Managing uncertainty in self-* systems. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 15(1):1–36, 2021
   - Gabriel A Moreno, Cody Kinneer, Ashutosh Pandey, and David Garlan. Dartsim: An exemplar for evaluation and comparison of self-adaptation approaches for smart cyber-physical systems. In *Proceedings of the 14th International Conference on Software Engineering for Adaptive and Self-Managing Systems*, pages 137–143. ACM/IEEE, 2019

   Artifacts:

   The source code for the GP planner is publicly available for extension and replication at the following GitHub repository: `https://github.com/squaresLab/sass`. The DARTSim exemplar is available at `https://github.com/cps-sei/dartsim`. Data and analysis code for this thrust is available at: `https://github.com/squaresLab/seams2018-data`, and also at `https://github.com/squaresLab/taas-2018-data`.
2. Techniques for generating reusable repertoires of adaptation strategies to broaden the types of unexpected changes that self-* systems can replan for effectively.

(a) An approach inspired by chaos engineering for obtaining planning knowledge for a range of change scenarios.

(b) Analysis approaches for extracting reusable planning components including clone detection and syntactic transformations.

Publications:

- Cody Kinneer, Rijnard Van Tonder, David Garlan, and Claire Le Goues. Building reusable repertoires for stochastic self-* planners. In *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, pages 222–231. IEEE, 2020

Artifacts:

The source code for the approaches described in this research thrust are available with the GP planner at `https://github.com/squaresLab/sass`. Data and analysis code is available at: `https://github.com/squaresLab/acsos2020-data`.

3. An adversarial extension to support plan reuse to promote the security quality attribute.

   (a) The Observable Eviction Game (OEG), a game theoretic model of system defense laying the foundation for self-* systems that can autonomously adapt in response to unexpected changes in the security landscape.

   (b) A co-evolutionary extension to support reusing adaptation strategies when planning for adversarial situations, enabling self-* systems to replan in the face of unexpected security threats.

   Publications:

   - Cody Kinneer, Ryan Wagner, Fei Fang, Claire Le Goues, and David Garlan. Modeling observability in adaptive systems to defend against advanced persistent threats. In *Proceedings of the 17th ACM-IEEE International Conference on Formal Methods and Models for System Design*. ACM-IEEE, 2019

   Artifacts:

   The source code for the co-evolutionary extension including the Bullseye exemplar is included with the GP planner code at `https://github.com/squaresLab/sass`. Source code for solving the Bullseye exemplar system with Gambit is available at `https://github.com/squaresLab/bullseye-gambit`. Source code for the Observable Eviction Game is available at `https://github.com/squaresLab/oeg-code`. Data and analysis code is available at `https://github.`

`com/squaresLab/sass-coev-data`.

## 1.4 Outline

The remainder of the thesis is organized as follows: Chapter 2 provides relevant background, including self-* systems, genetic algorithms, and game theory. Chapter 3 describes the core solution approach for reusing plans using stochastic search, including reuse enabling approaches for reducing the cost of plan reuse. Chapter 4 expands on the core approach, adding support for reusing repertoires of adaptation strategies including approaches for identifying generalizable plan fragments to build reusable plan repertoires. Chapter 5 discusses a co-evolutionary extension for applying plan reuse in domains with adversarial interactions. Chapter 6 describes the validation, including a description of the case study systems and claims. Chapter 7 provides a discussion of when our approach to plan reuse is beneficial, addresses the limitations of the thesis, and outlines some promising avenues for future work before concluding the thesis.

# Chapter 2

# Review of Literature and Background

This section provides the relevant background for the thesis. Section 2.1 provides an overview of the problem domain, including self-* systems and uncertainty. Section 2.2 describes details on the search-based approaches used in the thesis, including genetic algorithms and genetic programming, and also discusses prior work in search-based planning approaches. Lastly, Section 2.4 provides background for the unique challenges posed by the security domain including game theory concepts.

## 2.1 Self-* Systems

Self-adaptive, self-managing, self-protecting, or generally self-* systems are software systems that autonomously adapt to continue fulfilling their quality objectives in response to change. These systems are often composed of two subsystems, the *managed* system itself, and a *managing system*, which enables adaptation. Self-* systems frequently follow the five-component MAPE-K architecture [26], depicted in Figure 2.1. In this paradigm, the managing system gains information about the state of the managed system and its environment through sensors, and affects adaptation using actuators. A *monitoring* component gains information from the sensors, an *analysis* component examines this information and determines when adaptation is necessary, a *planning* component determines how the system should adapt by producing an adaptation *strategy* or plan, and an *execute* component carries out the plan using the actuators. Additionally, a fifth *knowledge* component provides shared information to each of the other four components to facilitate adaptation.

This thesis focuses on the planning component, which generates an adaptation

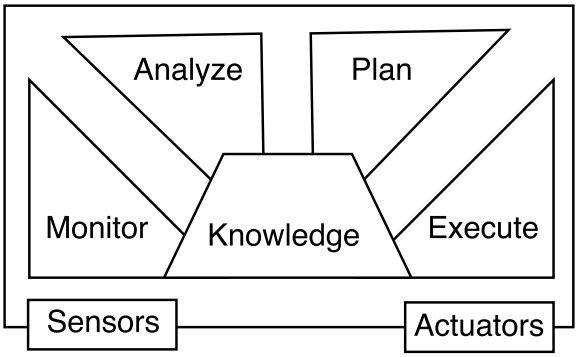


Figure 2.1: MAPE-K Loop for self-* systems.

strategy. These strategies frequently consist of several adaptation tactics, which are the atomic operations that the system can perform in order to adapt. In a cloud-based web services provider, an adaptation tactic might be "start a new server at data center C", while for an autonomous aircraft, a tactic could be "descend 1000 feet". Adaption strategies can consist of several tactics utilized together with control flow, for example: "start a new server at data center C, if successful reduce brownout, otherwise retry". Planners are broadly divided into *online* and *offline* planners based on when the planner generates a plan. Online planners generate a plan during run time, often trading off optimally in return for planning speed [57], while offline planners [11] precompute strategies for common or anticipated situations, which are then selected from at run time. These planners provide good solutions for cases that were considered during planning, but can struggle when confronted with novel situations.

Broadly, the purpose of self-adaptation is to enable systems to cope with uncertainty. Uncertainty can be defined as "... a state of incomplete or inconsistent knowledge such that it is not possible for a (dynamically adaptive system) to know which of two or more alternative environmental or system configurations hold at a specific point" [64]. Understanding uncertainty has been a focus of research in self-* systems, including modeling and taxonomizing uncertainty [3, 64, 43, 60], or managing it [47, 9, 48]. Often, uncertainty is categorized by its "anticipation" [3], "prospect" [43], or "level" [60], indicating the epistemic degree of uncertainty. One common demarcation of uncertainty is according to foreseen, foreseeable, and unforeseen types of changes [3, 64, 43], where foreseen changes are those aspects that were considered at design time and explicitly addressed, foreseeable are types of changes that are acknowledged but not currently addressed, and unforeseen are changes that by their nature cannot be anticipated. Another proposed categorization is accord-

ing to orders of ignorance [4, 60], with the 0th order of ignorance being knowledge, the 1st a lack of knowledge (but knowing the lack of knowledge exists), the 2nd is lacking knowledge as well as not realizing the lack of knowledge, the 3rd is lacking a process that facilitates the discovery that a lack of knowledge exists, the 4th order is lacking knowledge about the orders of ignorance. Existing approaches for self-adaptation focus on managing uncertainty that can be identified at design time, the so called known unknowns. In the planning component, unexpected changes from other sources of uncertainty necessitate replanning. This thesis explores approaches for more effective replanning in the face of this higher order of uncertainty.

## 2.2 Genetic Algorithms

Genetic algorithms (GAs) are stochastic search-based procedures for optimizing an objective function inspired by the principles of biological evolution [21]. At a high level, genetic algorithms iteratively improve a population of candidate solutions over a series of generations. A genetic algorithm consists of an individual solution representation, a fitness function to evaluate the quality of a candidate solution, and reproduction rules that specify how the next generation of solutions should be generated. At every generation, higher-quality solutions are more likely to be selected for reproduction, allowing the search to exploit promising solutions. To promote exploration of the search space, mutation operators populate the next generation of solutions by randomly modifying the selected individuals. After successive generations, the algorithm is expected to converge to high-quality solutions. Genetic algorithms are heuristic search algorithms, and while they often perform well in exploring large search spaces, finding the optimal or even a high-quality solution is not guaranteed.

*Genetic programming* (GP) is an evolutionary approach where individuals are represented as trees [37, 61]. This allows GPs to evolve abstract syntax trees for synthesizing code like artifacts such as adaption strategies. *Coevolution* is an evolutionary approach where multiple populations of individuals are evolved and influence one another's fitness. This can be done to evolve individuals representing the behavior of multiple agents, and has been used to find strategies in games [36]. GP and how it is used in my thesis approach is explained in more detail in Chapter 3.

There are many approaches for planning, including those that apply evolutionary approaches. Plato and Hermes [63] 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, commensurate with the different domain.

However, the key distinction in the work of this thesis is the focus on information reuse to handle uncertainty. That is, although Hermes is initialized with existing adaptation strategies, I focus explicitly on the effectiveness of reusing alternative starting strategies in the face of unanticipated scenarios.

EvoChecker [16] 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. This work 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. Moreover, this thesis also addresses identifying reusable planning components and adversarial environments.

Case-based plan adaption [52] explicitly reuses past plans in new contexts, in which context GAs have been explored directly [17, 44], e.g., by injecting solutions to previous problems into a GA population to speed the solution of new problems. Although the mechanism is similar, the presented approach is importantly novel in that it addresses a broader class of uncertainty, namely, where the source of uncertainty may be in the available tactics, the environment, or in the system's objective function.

Exhaustive planners are an alternative to heuristic planners. These planners evaluate every possible plan, which allows them to always find the optimal plan. Models checkers such as PRISM [42] can be used to exhaustively compute an optimal sequence of tactics to maximize one or more system objective (e.g., profit) for systems formalized as MDPs. While these planners will always find the best plan, they often suffer from poor scalability when the number of possible plans is large. In practice, exhaustive planners are often unable to cope with the complexity of large systems. In these cases, it is necessary to accept sub-optimal solutions to cope with the scalability challenge—for example, as is done by using stochastic model exploration.

## 2.3 Clone detection

Section 4.2.1 presents clone detection as an approach for building reusable repertoires of adaptation strategies. Clone detection is an approach for analyzing software for duplicate source code [65, 35, 25], which may be used to aid developers in refactoring code to promote maintainability or eliminate technical debt. Common approaches for performing clone detection include operating on abstract syntax trees [7], or

program dependence graphs [39]. Deckard [24] is a clone detection tool using a tree-based approach for performing clone detection that operates at the AST level. Deckard encodes program AST subtrees as vectors, and then computes the distance between these vectors to identify similar code regions. A clustering step results in Deckard outputting a list of clone clusters, groups of similar code-clones. Deckard is configurable and allows the user to specify the minimum number of code clones in a cluster, the minimum similarity for detecting clones, and the size of the stride, which influences the size of the detected clones. Section 4.2.1 describes work using clone detection as a means of extracting reusable planning components.

## 2.4 Security and Advanced Persistent Threats

Self-* systems frequently adapt in response to environment changes to maintain quality attributes; however, the security quality attribute poses unique challenges to self-* systems. A key challenge in self-securing systems is the presence of an adversary who can also take actions to affect the system, and can themselves adapt to the environment, including the system and its self-securing actions. This requires an approach to adaptation that can also consider the adversary's best response to the actions of the system, in effect planning for both agents at the same time. Chapter 5 presents a research thrust towards applying plan reuse to adversarial settings.

The most sophisticated adversaries are known as advanced persistent threats (APTs). The US National Institute of Standards and Technology (NIST) defines an APT as "An adversary that possesses sophisticated levels of expertise and significant resources.... The advanced persistent threat: (i) pursues its objectives repeatedly over an extended period of time; (ii) adapts to defenders' efforts to resist it; and (iii) is determined to maintain the level of interaction needed to execute its objectives" [33].

Each APT has a set of tactics, techniques, and procedures (TTPs) that is used to carry out an attack. These TTPs include the tooling and methods used by a group of individuals dedicated to a particular purpose, such as gathering intelligence, stealing merchantable artifacts, or causing disruption. In some cases, a threat actor may have multiple APT groups defined by their distinct TTPs [1]. Because TTPs represent the accumulated knowledge, skills, and abilities of attackers, they can be difficult to change. However, a nation-state with multiple APT groups under its control could reassign responsibility for attacking a target from one APT group to another, or a single APT group could swap out one set of tooling and command and control infrastructure for another if need be. In the most sensitive operations, APT groups will use multiple sets of TTPs, including multiple types of malware, to ensure persistent presence even in the case of detection.

For the defender, knowledge of an attacker's TTPs is often crucial to successful attack mitigation because that knowledge can be used to look for likely places where the system might have been compromised and to predict future courses of action. Such knowledge can be gained in several ways, such as simply waiting to see what the attacker will do next, or putting in place active detection mechanisms, or *active measures* (e.g., honeynets or camouflage) [23, 67].

**Game Theory.** Unlike other quality attributes that a self-* system may optimize like quality of service, security presents a unique challenge in the form of an attacker. Like the self-* system itself, the attacker can take actions that affect the system, and can themselves gather information and adapt to the behavior of the system to further their own interests. Game theory provides a framework for reasoning mathematically about interactions between multiple agents, or players [55]. A normal-form game is defined by a tuple $(\mathcal{N}, \mathcal{A}, u)$. $\mathcal{N} = \{1...n\}$ is the set of players. $\mathcal{A} = \Pi_{i=1}^{n} \mathcal{A}_i$ is the set of joint actions, where $\mathcal{A}_i$ is the set of actions for player $i$. $u = (u_1, ...u_n)$ and $u_i : \mathcal{A} \rightarrow \mathbb{R}$ is the payoff or utility function for player $i$ that maps the players' joint action profiles to an outcome value. Each player is seeking to maximize their own individual utility. A player's strategy can be pure (i.e., take a deterministic action) or mixed (i.e., randomly choose an action according to some probability distribution). The Nash equilibrium (NE) of a game is the strategy profile $\sigma = (\sigma_1, ...\sigma_n)$ for all players such that no player can gain from unilaterally changing their strategy. That is, that each player is playing the best response to each other player.

More complicated games with sequences of actions are often modeled by extensive-form games (EFG), which can be represented by a game tree where each node corresponds to a unique history of actions taken by all players and chance from the root of the game, and each edge corresponds to possible actions available to the player (could be a chance player) who will choose an action at the node. Players get payoffs at the leaf nodes of the game tree and then the game terminates. In addition, each players' choice nodes can be partitioned into information sets to model the imperfect information in the game. A player cannot distinguish between nodes in the same information set. A pure strategy for player $i$ in an EFG assigns one action for each information set of player $i$. Stochastic behavior can be modeled by introducing a nature, or chance player, who moves according to a fixed probability distribution. By enumerating pure strategies for all players, we can get an induced normal form game of an EFG and the NEs are preserved.

In complete information games, all players know the identity and payoff functions of all other players. Bayesian games relax this assumption by allowing multiple types of players and that at least one player is unsure of the type of another player. Formally, a Bayesian game extends the aforementioned normal-form game model by

introducing $\Theta = \Pi_{i=1}^{n} \Theta_i$ where $\Theta_i$ is the set of possible types for player $i$ and a common prior of joint probability distribution of players' types $\mathcal{P} : \Theta \to [0, 1]$. The utility function of a player is dependent on the players' joint type profile and joint action profile, i.e., $u_i : \mathcal{A} \times \Theta \to \mathbb{R}$. The Harsanyi transformation [19] converts a Bayesian game to a normal-form game.

Stackelberg games assume two kinds of players: a leader, and one or more followers. Rather than each player choosing a strategy simultaneously, in Stackelberg games, the leader acts first and commits to a strategy. The followers then observe the strategy of the leader and best respond. This game type is useful in a security context to capture the fact that a sophisticated attacker can often observe the defender's strategy before acting. Several approaches exist for efficiently solving Strong Stackelberg Equilibria (SSE), in both Bayesian and extensive form settings [58, 40, 73].

# Chapter 3

# Approach Overview: Responding to unexpected changes with plan reuse and stochastic search

While self-* systems can enable systems to autonomously respond to situations that they were designed for, they struggle when confronted with unexpected changes. This thesis presents an approach for reusing existing plans with stochastic search to allow self-* systems to more effectively replan when faced with these changes. This section describes the first contribution of the thesis, a planner, based on genetic programming, that reuses existing adaption strategies after an unexpected change occurs. This planner will serve as the foundation for further extensions facilitating plan reuse. The planner is described in terms of a case study system, a cloud-based web server (Omnet), that will serve as a running example.

The planner reuses previously-known information using GP to efficiently generate plans in a large, uncertain search space in response to unforeseen adaptation scenarios. The 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.

Stochastic search requires a fitness function for estimating the utility or fitness of candidate solutions. In this work, we assume this is provided via a model of the system that can simulate executing candidate plans and output the expected resulting utility. After an unexpected change occurs, the system model must be updated to reflect the new behavior after the unexpected change, and this update triggers the planner to replan. The mechanism for synchronizing the system model with the

actual world is outside the scope of the thesis; this may be done manually (likely with less effort than replanning), or automatically [74, 27] (this assumption is discussed in more detail in Section 7.2.1). 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 adaptation strategies.

The planner 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 (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. Sections 3.2–3.4 provide the necessary technical details on the GP implementation. The approach is explained in terms of the cloud-based web server case study explained in detail in Section 3.1.

Algorithm 1 shows how the GP planner works at a high level. First, on line 1, a population of candidate solutions (adaptation strategies) 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. 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 3.5.

**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
5:
6: return best_ind(p)
```

A new GP application is defined by how individuals are represented (Section 3.2); how they are manipulated during reproduction through mutation and crossover (Section 3.3); and how the fitness of candidate solutions is calculated (Section 3.4).

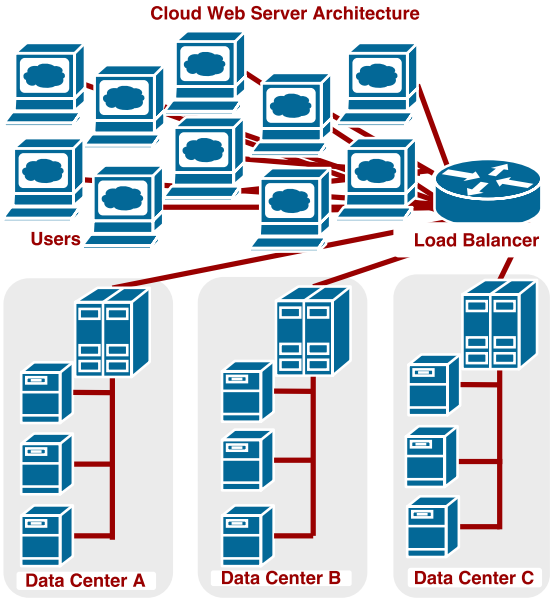


Figure 3.1: Cloud web server architecture.

## 3.1 Cloud Web Server

The thesis is evaluated on three case study systems. The first case study system will serve as a running example for the remainder of the thesis; the remaining case studies are described in Section 6.2. The first case study system is a cloud-based web server with an N-tiered architecture, depicted in Figure 3.1[1]. The system distributes requests from users between several data centers using a load balancer. Each data center contains servers of a different type, with each type having different performance characteristics. Generally, the more requests that a server can handle per unit time the higher its cost. The system generates revenue by delivering ads along with requests.

**Utility.** The system has several quality attributes that may be optimised, including (1) *Profit*, the revenue generated by serving ads minus the operating costs, (2) User *latency*, the delay that users experience when the number of incoming requests exceeds the capabilities of the running servers, and (3) User-perceived *quality*, the percentage of users viewing high-fidelity content, as opposed to a lower quality version that may be delivered by a "brownout" mechanism (requests serviced with the

[1]This case study system has been used to evaluate self-* planners in related work [57, 12], work published as a contribution of the thesis [29], and as a standalone exemplar [50]

lower-quality version are said to be "throttled").

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. 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 $S_i$.

When evaluating latency, 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 adaption 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.

**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, unthrottled and throttled. 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 [34] 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 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. These values were selected for illustrative purposes and in practice would need to be empirically determined or estimated.

**Change Scenarios.** 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: e.g., users might switch from using one feature on a platform to another, or might use the system at a different time.

To explore a representative variety of different change scenarios, 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.** 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*.

## 3.2 Representation

Individuals in the population are plans represented as trees. Trees are a natural choice since they used to represent computer programs, which we observe are similar to adaptation strategies. Figure 3.2 gives a Backus-Naur grammar for the plans. Each plan consists of either (a) one of six available *tactics* (described in Section 3.1),

⟨*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 3.2: 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.

or (b) 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 3.3 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 [11]. Intuitively they resemble decision trees in which the next action taken is determined by the success or failure of prior actions. Unlike Stitch, this language does not consider plan applicability (guards that test state to determine when a plan can be used). Instead, applicability will be determined by choosing the plan with the highest expected utility, making explicit guards in the planning language unnecessary. Note that any plan expressible in the planning language could be expressed with only the `try-catch` operator, and that the language can represent any PRISM MDP [42] plan (or policy) as a tree of `try-catch` operators with depth $2^h$, where $h$ is the planning horizon.

## 3.3 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 (this limit was selected to help prevent excessively large plans

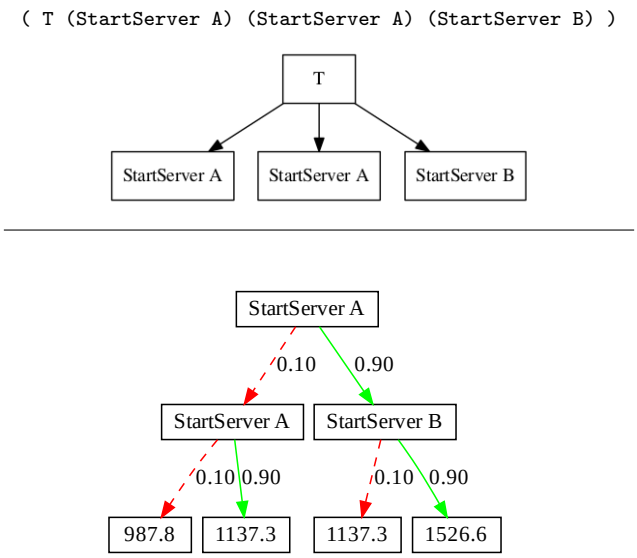


Figure 3.3: Top: An example plan. Bottom: This plan's system state tree. Dashed red arrows denote tactic failure, while solid green denotes success.

that take too much time to evaluate). The crossover operator [72] selects a subtree in each of two parent plans (selected via tournament selection [37]) and swaps them to create two new plans. Syntax rules are enforced 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. Such plans are penalized rather than prevented, to allow the search to break out of local optima.

## 3.4 Fitness

Candidate fitness is evaluated 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, evaluation 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 representing the expected fitness of the plan. Path fitness is the quality of the leaf node system state, measured as one or more of *profit*, *latency*, and *user perceived quality* (Section 3.1). 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.

To illustrate, Figure 3.3 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 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 [49]. 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, 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. The 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*, $t$ is the time plan's execution time.

As with many optimization techniques, a GP typically includes many tunable parameters that require adjustment to achieve good results. 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

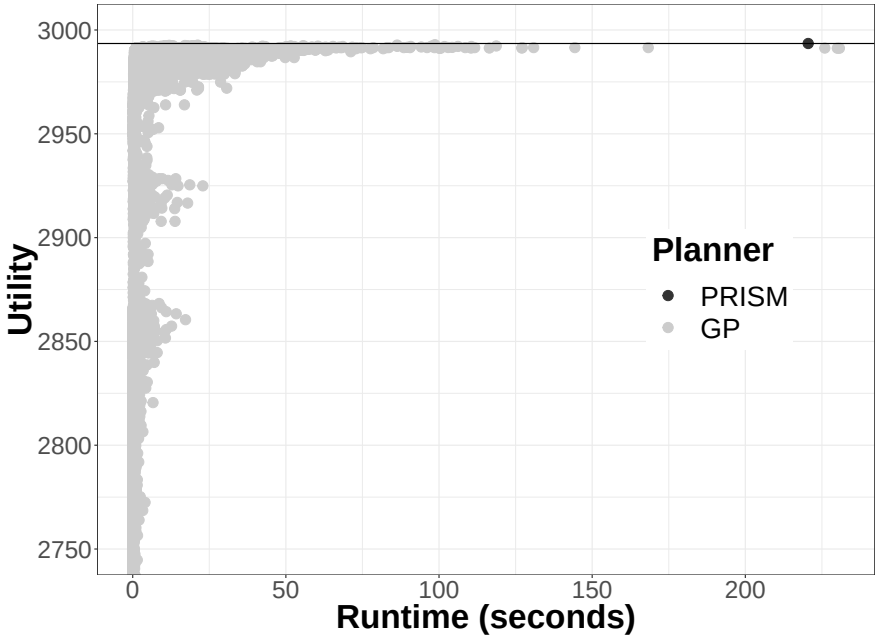


Figure 3.4: Utility versus planning time for GP parameter configurations. Many configurations produce similar utility results to PRISM, significantly faster.

Table 3.1: 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. |

penalty thresholds. Figure 3.4 shows results from this parameter sweep performed on the cloud-based web server case study. The dark point at the top of Figure 3.4 shows the optimal system profit (fitness) and planning time (200 seconds) of the PRISM planner, which performs an exhaustive search of all possible plans. 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).

While the planner described in this section is a first step towards self-* systems that can reuse information to replan effectively in response to unexpected changes, several key challenges remain, including the long evaluation time required to compute the expected fitness of prior plans, deciding how to seed the initial population to maximize the number of situations that can be replanned for efficiently, and the difficulty in planning for the security quality attribute when an adversary is also adapting in response to the system. Section 3.5, Chapter 4, and Chapter 5 present extensions to this planner to address these key challenges.

## 3.5 Reducing plan evaluation time with reuse enabling approaches

Preliminary results show that initializing the search by naïvely copying existing plans does not result in efficient planning, and in most cases is inferior to replanning from scratch with a randomly generated starting population [29]. This is due to 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, this section presents 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 3.1 shows a summary of these approaches.

To reduce the number of long starting plans that the planner needs to evaluate, a `scratch_ratio` percent of the starting are initialized with short (a maximum depth of ten) randomly generated plans, and only the remaining $1 - $ `scratch_ratio` individuals are seeded 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, a `kill_ratio` parameter is used, which 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, the search is initialized with small plan "trimmings" from the initial plan. The planner generates trimmings by randomly choosing a node in the starting plan using Koza's node selector [37] (an approach that randomly selects from every node in a tree, based on whether the node is a terminal, nonterminal, or root) 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.

The reuse enabling approaches are evaluated specifically in Section 6.3.1. The results show statistically significant improvements for two out of the three approaches

(*kill_ratio* and *trimmer*), with the *scratch_ratio* showing a small improvement but not statistically significant. These reuse enables are also used throughout the evaluation where appropriate, and contribute to the effectiveness of plan reuse.

# Chapter 4

# Building reusable repertoires by identifying generalizable plan fragments

Section 3.5 discussed improving planning utility by randomly trimming existing plans into smaller sub-plans, balancing reused plans with short randomly generated plans in the starting population, and prematurely terminating the evaluation of a proportion of long running plans. While results show that these techniques resulted in an improvement (see Section 6.3.1 for results on this part of the approach in detail), they focus on reusing a single adaptation strategy only, and they ignore the insight that some plan features are more reusable and amenable to evolution than others. For example, many plans generated for the cloud web server case study frequently start instances of server type C. Since this server type happens to have the best computation ability per operating cost, starting more of them is often useful for a range of unexpected occurrences. As long as type C is the most economical type, the system should make sure that these servers are being utilized. This insight can be extracted from the adaption strategies generated for the case study by observing the repetition of this tactic, even without the domain knowledge needed to explain why the tactic is generally useful (because of the performance and cost associated with the servers that it starts).

To obtain the maximum benefit from existing plans, this chapter explores building and reusing repertoires of adaptation strategies, including analyzing plans to identity commonly occurring tactics encoded in sub-plans, which we hypothesize will be more likely to be used in subsequent planning iterations, and thus result in improved planning utility compared to selecting sub-plans randomly. Analysis techniques to

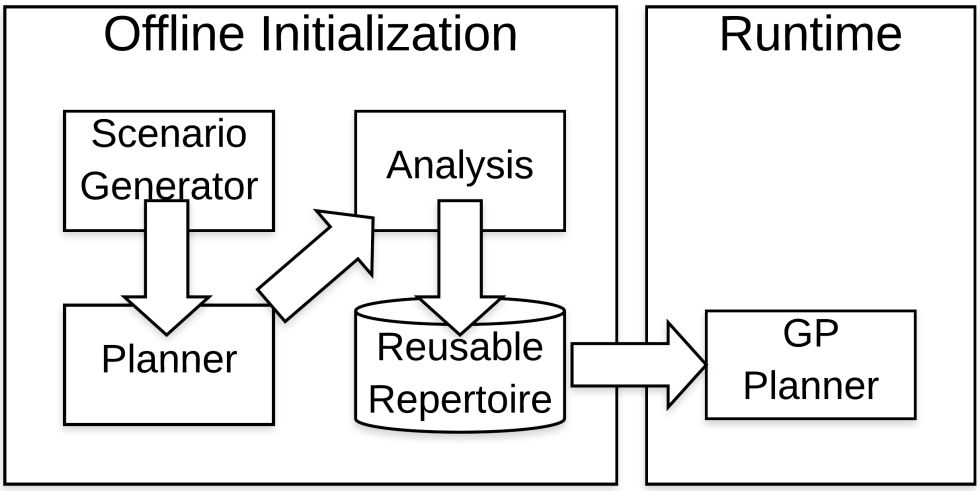


Figure 4.1: A high level view of the approach.

determine common code patterns, or code clones have been applied to source code, such as Deckard [24] or program dependence graphs [39]. Since Deckard is a well-known clone detection technique with a publicly available implementation, it provides a good starting point for detecting commonly occurring elements in a collection of plans. Self-* plans can be translated to representative Java code for analysis by Deckard. Deckard can be given a collection of existing self-* plans generated for an existing adaptation scenario, and will output which plan elements are most commonly occurring. These plan elements will be converted to plans and taken as the initial population of the search when replanning for an unexpected change scenario. We will then compare the fitness utility of using common subplans to seed the search with planning from scratch and planning with the techniques presented in prior work [29].

Figure 4.1 overviews the approach, which divides the planning process into an offline and runtime step. During the offline initialization phase, we construct a reusable repertoire of adaptation strategies for the planner to incrementally evolve at runtime. This phase is further subdivided into a two step process: firstly, exploring the space of randomly generated change scenarios and producing adaptation strategies to address them, and then analysing the generated adaptation strategies to extract generalizable and cost effective components for the repertoire. In the online phase, we extend our prior genetic programming planner [29] by seeding it with the adaptation strategies in the repertoire.

A key idea behind repertoire construction is that certain "pieces" of plans are par-

ticularly informative for reuse. For example, repeated planning components, such as starting more instances of the most cost effective server type, are likely to generalize. Thus, effective repertoire construction requires:

1. a diverse set of previously-produced plans, constructed in response to a wide variety of potential system changes, and
2. a way to consolidate and identify the most plan components that hold the most promise for future reusability

For (1), we build on the idea of *chaos engineering* to explore the space of possible changes by randomly generating change scenarios to generate a diverse base of planning knowledge; we explain in more detail in Section 4.1. For (2), we make the observation that plans are, effectively, small programs, and our goal in analyzing them is to identify semantically-meaningful programs or program pieces that may be informative for future use. We thus present two techniques for this analysis phase, one that adapts clone detection to this domain (Section 4.2.1), and another that proposes a set of rule-based plan transforms to identify cost-effective plan pieces (Section 4.2.2).

## 4.1 Generating Unexpected Changes

Our technique requires a diverse set of starting strategies that may generalize to future situations. To obtain these strategies, we explore the space of unexpected changes by generating change scenarios using a mutation-based approach inspired by *chaos engineering.* Chaos engineering is an approach to promote software quality attributes such as availability and robustness in large complex systems [6]. It involves subjecting the target system to chaos experiments, which should be conditions that may result in system entering an undesirable state, with the goal of verifying that the system appropriately responds to the experiment. If the system does not respond in an acceptable way, then it can be improved to be more robust to similar situations that might be encountered in production. An example of chaos engineering is Netflix's Simian Army [22].

We therefore propose an approach for building a reusable repertoire by performing chaos experiments offline to obtain a diverse set of adaptation strategies for later reuse. At a high level, this approach randomly selects a scenario attribute, and then randomly mutates it. Because the vast majority of attributes (150 out of 159) are the availability zone specific parameters, random attribute selection is biased to favor the other attributes, to promote scenario diversity. Table 4.1 shows this distribution. Attributes within the same type are chosen uniformly at random. Since different at-

| Attribute Type | Selection Rate |
|---|---|
| Utility Coefficients | 13.33% |
| Tactic Failure Rates | 23.33% |
| Number of Users | 15.75% |
| Instance Cost | 15.75% |
| Instance Power | 15.75% |
| Instance Brownout | 15.75% |

Table 4.1: Scenario attribute type and selection probability during mutation.

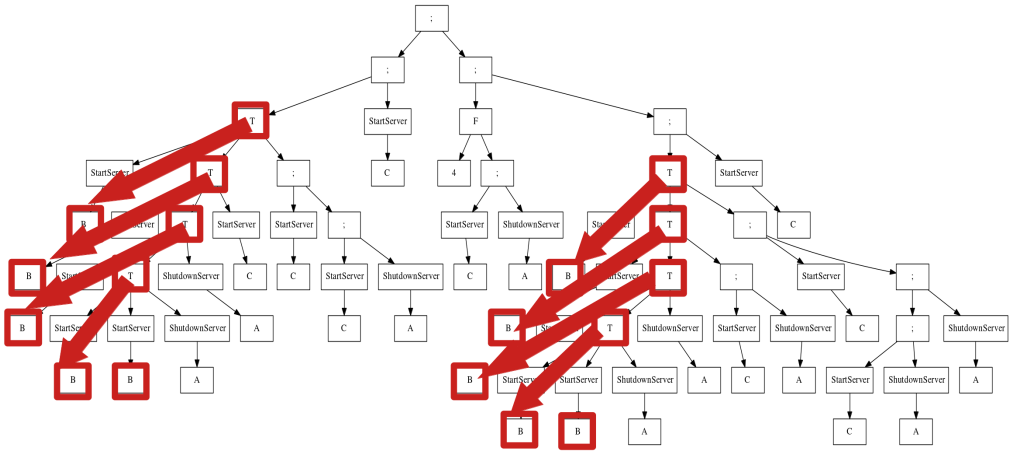

Figure 4.2: An example of a clone within a plan.

tributes have different sensitivity to change, the particular mutation applied depends on the attribute selected. This mutation procedure is repeated $m$ times, where $m$ is the number of desired mutations.

## 4.2 Extracting Reusable Components

### 4.2.1 Clone detection

Our first intuition for how to improve a repertoire constructed from a diverse set of plans is that some planning motifs are more likely to generalize to unexpected situations. For example, more servers of the most efficient type (the best performance per cost) is useful in a variety of situations, e.g., if the number of users increases or if the

processing resources per request increases. Of course, there are other changes where this tactic is not helpful (such as when the quality requirements change dramatically), but overall this applies to many change scenarios. This motif may therefore appear in many of the diverse plans generated in the first phase.

Thus, our first approach leverages *clone detection* to identify reusable plan components that appear in many plans in the scenario set. Clone detection analyzes software for duplicate source code (see refs. [65, 69] for surveys), which aids developers in refactoring code to promote maintainability or eliminate technical debt. Although this technique is more commonly applied with the aim of reducing redundancy, we observe that the idea can identify planning components that are more likely to be generalizable. Figure 4.2 shows an example of a clone within an adaptation plan. In this plan, a subplan is repeated. Because this clone is duplicated, it possibly contains important planning knowledge; this key knowledge may be more likely to generalize. By extracting just the clone rather than repeating the full plan(s) in the repertoire, the planner can reuse this prior knowledge more cost-effectively. We therefore apply clone detection to the generated adaptation strategies to find those adaptation strategy components that occurred multiple times throughout the considered change scenarios.

**Implementation.** Our implementation builds on the Deckard [24] clone detection tool. Deckard performs clone detection by encoding abstract syntax tree (AST) subtrees as vectors, and computing the distance between these vectors to identify similar code regions using clustering.

Note that our approach can generalize to any clone detection mechanism. We use Deckard because it operates on generic tree structures (and can thus be straightforwardly adapted to our plan representation), it considers semantics, is scalable to large AST sizes, and has a publicly-available implementation.

We must make changes to the vector generation step to effectively adapt Deckard's approach to our planning context. Converting an AST into numerical vectors produces a representation amenable to clustering; Deckard generates vectors for AST subtrees based on the number and type of child nodes. By default, Deckard does not consider variable identifier names during vector generation. This is sensible for analyzing large programs written in a general purpose language like Java, where identifiers often vary between clones and where the large number of identifiers quickly explodes vector size. However, our planning language is simple by comparison. More importantly, tactic names (like `StartServer` encode considerable semantically meaningful information. We therefore developed our own vector generator step for the planning language that tracks the occurrence of tactic names.

| | | | |
|---|---|---|---|
| `seq-take-first` | `(; (:[1]) (:[2]))` | ⇒ | `(:[1])` |
| `seq-take-second` | `(; (:[1]) (:[2]))` | ⇒ | `(:[2])` |
| `try-take-first` | `(T (:[1]) (:[2]) (:[3]))` | ⇒ | `(:[1])` |
| `try-take-second` | `(T (:[1]) (:[2]) (:[3]))` | ⇒ | `(:[2])` |
| `try-take-third` | `(T (:[1]) (:[2]) (:[3]))` | ⇒ | `(:[3])` |
| `try-unnest` | `(T (:[1]) (T (:[1]) (:[2]) (:[3])) (:[3]))` | ⇒ | `(T (:[1]) (:[2]) (:[3]))` |
| `for-prune` | `(F i:[1] (:[2]))` | ⇒ | `(:[2])` |
| `for-decr`† | `(F i:[1] (:[2]))` | ⇒ | `(F i:[1] (:[2]))` |

Table 4.2: Syntax transformation rules for pruning plans. Hole syntax, like `:[1]`, binds an identifier `1` to an expression. Each rule either replaces a nonterminal expression with a subexpression, or reduces the number of times a subexpression is evaluated. †The `for-decr` rule decrements the loop iterator matched by `:[1]` within the fixed integer range 3–10. For brevity, we elide the rewrite rule that decrements these values.

### 4.2.2 Rule-based Plan Transformation

The clone detection approach can automatically identify reusable repeated planning components. However, human domain expertise, particularly in the peculiarities of the planning language and domain, provides an important avenue for further improvement to repertoire construction. Naive human replanning is time-intensive and expensive, and so any mechanism for incorporating expert knowledge into planning must be sensitive to this cost.

We therefore propose a second approach to repertoire improvement based on human-provided, rule-based source-level transformation templates. Such templates are useful for improving general software quality [28], suggesting that transformation templates for our program-like adaptation strategies could usefully improve their quality, in terms of their generalizability and reusability. For example, we can exploit a priori knowledge of our plan grammar and operator semantics to apply plan transformations that avoid generation of redundant or known-expensive subplans.

We use `Comby` for declaratively specifying templates [2]. `Comby` performs transformations on trees using declarative templates that are syntactically close to the underlying programming language; this is our planning language, in this context. Such templates are therefore lightweight and relatively easy-to-write, easing the burden of manually specifying transformation templates. `Comby` generically supports language syntax with little or no configuration, and is thus a suitable tool for generalizing our template-driven approach to other planning languages like `Stitch` [11]

or PRISM [42].

**Transformation rules for plan reuse.** Table 4.2 summarizes the eight transformation rules we produced for plans in our exemplar system. Each rule reduces the size of the plan by removing subexpressions, corresponding to subplans. To illustrate, consider the first rule provided in Table 4.2. The `seq-take-first` rule matches a sequence expression (denoted by `;`) and binds named identifiers `1` and `2` to its two respective subexpressions. The `:[ ]` syntax denotes a structural *hole* that binds to expressions. The transformation, denoted by ⇒ reduces the sequence expression to only the first subexpression, corresponding to identifier `1`.

All syntax besides hole syntax refers to *concrete* syntax in the underlying language, including operator keywords like `T` or `F` and parentheses. `Comby` rules always match balanced parentheses, which ensures that both matched and transformed subexpressions and plans are syntactically well-formed. `Comby` is thus well-suited to transforming expressions corresponding to subtrees (like balanced parentheses), corresponding to subplans. These transformations are generally not expressible using regular expressions and would be otherwise difficult to implement programmatically.[1]

Our rules are informed by the grammar in Figure 3.2: for each nonterminal operator (i.e., Sequence, Try-catch, and For loop) we wrote a rule that extracts a respective subexpression (`seq-take-*`, `try-take-*` rules), or reduces the number of iterations that subexpressions are evaluated (`try-unnest`, `for-*` rules). In particular, the `seq-take-first` and `seq-take-second` rules pick the first (resp., second) expression from a sequence expression. The `try-take-*` rules pick one of three Try subexpressions. The `try-unnest` rule prunes Try expressions that share identical child nodes in the first and third arguments.[2] The intuition is that structurally similar subtrees can yield similar benefits, and nested repetitions imply duplicative evaluation unlikely to improve performance. Similarly, `for-prune` and `for-decr` reduce the number of times a For loop executes.

Our experience is that writing programs (e.g., in Java) for transformation rules inside the genetic planner is possible but disadvantageous. Transformations expressed in code are less readable, and can contribute to a planner becoming a black-box, motivating our use of transformation rules. Declarative rules easily express lightweight transformations, and decouples the rule-based system from probabilistic plan discovery, offering greater flexibility.

[1] Applying a rule to expressions in a plan requires a simple command-line invocation: `comby '(T (:[1]) (:[2]) (:[3]))' '(:[1])' plan.ast`

[2] When the same hole identifiers are used in a rule, the expressions must be syntactically equal for the rule to match.

**Rule application.** We apply the eight rules to the initial repertoire, selectively removing expressions, which results in smaller plans overall. The general intuition is that smaller plans lead to quicker evaluation times, while retaining particularly valuable subplans for reuse, and thus contribute to greater overall utility. The genetic programming planner explores coarse-grained changes (both adding or deleting subplans), with the overall effect of performing additive changes that create ever-larger plans. Thus, it may miss the opportunity to prune less useful subplans (especially those containing large subexpressions), akin to getting stuck in local optima.

The approaches for generating reusable repertoires of adaptation strategies described in this chapter are evaluated in Section 6.3.2. The clone detection component of the approach shows the strongest results, with an 11% improvement over reusing a single plan only. The clone detection approach resulted in a smaller improvement, but could result in useable strategies ten seconds faster due to reducing the evaluation overhead. The best performing syntactic transformation rule resulted in an additional 3.5% improvement over reusing the repertoire without additional analysis.

# Chapter 5

# Plan reuse in an adversarial setting

This chapter describes a co-evolutionary extension to the GP planning approach to enable plan reuse with stochastic search in self-* systems with adversarial properties such as promoting the security quality attribute. Adaptation to promote security raises unique challenges compared to other quality attributes. Promoting security involves ensuring a system is protected from the malicious activity of other actors, or attackers. Like the system, the attacker can also adapt in response to change, including changes that the system makes to adapt to the attacker. In the context of stochastic search, this complication requires non-trivial extensions to the approach described for other quality attributes. In this chapter we first describe the Observable Eviction Game, a model for reasoning about APT defense in self-* systems, which will serve as the foundation for our security extension. This chapter then describes the extension to the GP planner using co-evolution to support adaptation and strategy reuse in the presence of an adversary.

## 5.1 Foundations: The Observable Eviction Game

Self-protecting systems dealing with APT scenarios need to automatically decide between attempting to evict an attacker versus attempting to gather more information about them, as explained in Section 2.4. To support designing these systems, a useful model of decision making should include the following elements:

(C-1) Multiple types of attackers, each with different goals and available TTPs

(C-2) A defender who must choose between attempting to evict the attacker or gathering more information, with or without active measures

(C-3) A defender who must balance thwarting the attacker with minimizing disrup-

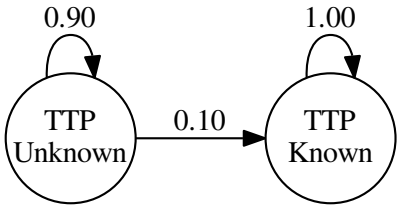


Figure 5.1: Markov process for TTP observability.

tion to the system

(C-4) Eviction success should be predicated on the defender's knowledge about the attacker's identity

(C-5) An attacker who changes their behavior and becomes more difficult to evict after an unsuccessful eviction

We present a novel game model called the Observable Eviction Game (OEG) that satisfies all of the above criteria. OEG is a Bayesian game with two players, i.e., $\mathcal{N}$={1,2}. Player 2 is an attacker who seeks to compromise the system, and player 1 is a defender who wants to minimize the attacker's success and disruptions to the system's operations. There is one defender type, but the attacker's type is drawn from a set supporting modeling different APT threat groups (C-1), denoted as $\Theta = \{\theta_1, ..., \theta_M\}$. OEG depicts sequential actions of players and therefore can be described as an EFG.

In the remainder of the section, Section 5.1.1 explains how the game proceeds and the actions available to each player, Section 5.1.2 explains the utility functions, and Section 5.1.3 describes solving the game for equilibria.

### 5.1.1 Actions

OEG models observability as a first-class concern and we leave out details that are less relevant to observability. As an overview of the game, OEG models the defender-attacker interaction in discrete time. The attacker chooses their attack plan, i.e., TTP, at the beginning of the game, which is initially unobserved or unknown to the defender. The game lasts $\tau$ time steps. Each time step the defender can perform an eviction attempt or perform some other observational tactic (such as waiting or taking active measures) to gain more information about the attacker (C-2). In addition, in each time step, nature, or the chance player, randomly determines whether the attacker's attack plan will become observable to the defender.

Rather than considering every possible path an attacker might take through an

attack tree [68] to attack the system, we consider attacker strategies at a higher level of abstraction. The attacker's attack plan is described by a TTP, and the attacker may choose among several TTPs. This is consistent with the behavior of real APT groups (Section 2.4) since these groups often gain expertise in a particular set of techniques and operate in a particular way. In the remainder of the paper, we will use the terms TTP and attacker plan interchangeably. More concretely, in time step 0, the attacker chooses an action $\gamma_j \in \Gamma = \{\gamma_1, ...\gamma_Z\}$ that indicates their attack plan or TTP. They will not change it unless they observe that the defender makes an eviction attempt or takes an active measure.

In time step $t \in \{1, ...\tau\}$, the defender can choose an action from the set of eviction attempts $\Omega = \{\omega_1, \omega_2, \ldots, \omega_L\}$. The success of an eviction attempt depends on the suitability of the eviction action to the TTP chosen by the attacker. We denote by $\chi_{jl} \in [0, 1]$ the effectiveness of the defender's eviction action $\omega_l \in \Omega$ to evict the attackers TTP $\gamma_j$, which describes the probability that the attacker can be successfully evicted, with 1 indicating always successful and 0 indicating always unsuccessful. We assume that for each TTP $\gamma_j$, there exists some eviction attempt $\omega_l$ such that $\chi_{jl} = 1$ and can successfully evict the attacker. If the defender performs a successful eviction attempt, the attacker is immediately evicted, ending the game. If the defender performs an unsuccessful eviction attempt, the attacker remains in the system and is alerted to the defender's knowledge of them, making it more difficult for the defender to evict them in a subsequent attempt (C-5). Instead of modeling the attacker's change of behavior and the defender's subsequent attempts in detail, we simply assume the attacker will stay in the system until the end of the $\tau + F$ time step without being interrupted by the defender. Equivalently in the game tree, the failed eviction leads to a leaf node as no more actions will be taken. This modeling approach is motivated by the high cost of a failed eviction attempt in practice, which can result in an attacker digging in and becoming more difficult to evict in the future (see Section 2.4). Since the game ends on a eviction action in either case, the defender can only take at most one eviction action throughout the game, and the defender can choose to not take an eviction action within $\tau$ time steps. This model reflects the fact that a defender's ability to evict an APT attacker depends on the defender's knowledge about the attacker (C-4). If the defender knows the attacker's TTP, the defender can choose the most suitable eviction action with $\chi_{jl} = 1$. On the other hand, if the defender uses an eviction action without knowing the TTP of the attacker, the defender may choose an ineffective action, resulting in a failed eviction attempt.

In addition to choosing an eviction attempt, the defender can also choose an action from the set of observational tactics $\Phi = \{\phi_{L+1}, \phi_{L+2}, \ldots, \phi_{L+Q}\}$ where $\phi_{L+1}$

is the default tactic of "wait" and $\phi_l, \forall l \in L+2, ..., L+Q$ are active measures that can be applied. Whether or not the attacker's TTP is known to the defender is determined by nature and the observation tactics chosen by the defender. Intuitively, without any active measures taken by the defender, the longer the attacker stays in the system, the more observable the attacker's TTP, i.e., the more likely the defender learns the attacker's TTP. We model the observability of the attacker's TTP over time as a two-state Markov process. Figure 5.1 shows an example TTP observability model. Initially the defender does not know the attacker's choice of TTP, i.e., the attacker is in the "TTP Unknown" state. After each time step, the defender has some chance of learning the attacker's chosen TTP and the attacker may move to the "TTP Known" state. We use $q_j \in [0, 1]$ to denote the transition probability $P(\text{TTP Known}|\text{TTP Unknown})$ if the attacker chooses TTP $j$. A lower $q_j$ means TTP $j$ is stealthier and harder to observe. The attacker remains in the "TTP Unknown" state with probability $1 - q_j$. Therefore, in each time step $t \geq 1$, the chance player determines whether the attacker's TTP becomes known according to probability distribution $\langle q_j, 1 - q_j \rangle$ if the defender always choose to wait. If the defender learns the attacker's TTP, we assume that the defender will choose to evict the attacker immediately, ending the game. If the defender does not learn the attacker's TTP, the game continues.

If the defender uses an active measure, $q_j$ will increase and the defender may learn the attacker's TTP earlier. However, the defender takes the risk of being noticed by the attacker, resulting in a change in the attacker's behavior. An observational tactic $\phi_l$ is associated with a scalar representing the effectiveness of the tactic, denoted as $x_l$, and scalars representing how observable the tactic is to each attacker TTP, denoted as $y_{jl}$. The first element, $x_l \in [0, 1]$ describes the decrease in the attacker's probability of remaining hidden if the tactic is not noticed by the attacker. Let $q_j^t$ to denote the transition probability at time $t$. If the defender applies $\phi_l$ at time $t$ without being noticed by the attacker, then $q_j^t = 1 - x_l(1 - q_j^{t-1})$. In the second element, $y_{jl}$ represents the probability that the attacker notices the defender's action $\phi_l$, dependent on the TTP $\gamma_j$ they are using. This allows the model to capture the fact that different TTPs may be more or less likely to observe the defender's countermeasures. In the game tree, the stochasticity of attacker noticing the defender's action can be represented by having the chance player determine the observability of action $\phi_l$ right after the defender taking action $\phi_l$. If the attacker noticed the defender's action, the attacker changes their behavior and becomes more difficult to evict, resulting in the same outcome as a failed eviction attempt, and the game ending. The defender may take multiple active measures throughout the game. For the default tactic "wait", $x_{L+1} = 1$ and $y_{j(L+1)} = 0, \forall j$.

### 5.1.2 Utilities

When the game terminates, each player gets a utility or payoff. We model the attacker's utility as the amount of time they remain in the system multiplied by the appropriateness $\alpha_{ij} \in [0, 1]$ of their chosen TTP $\gamma_j$ to their type $\theta_i$. This modeling approach captures the fact that having more time in the system gives the attacker more opportunity to accomplish their goals. However, some TTPs may be more aligned with their goals than others. In addition, APT groups are often trained in a particular set of techniques amenable to their goals, and other TTPs are not available to them. Modeling the appropriateness of the attacker's TTP captures these facts, as a higher $\alpha_{ij}$ indicates more alignment between the TTP and their goal and $\alpha_{ij} = 0$ indicates that TTP $\gamma_j$ is not available to the attacker of type $i$.

Similarly, we model the defender's utility as the negation of the amount of time the attacker remains in the system multiplied by $\delta_i \in [1, 10]$, a coefficient describing how disruptive the attacker type $i$ is. Coefficients $\delta_i$ model the fact that different attackers may also be more or less disruptive to the defender. An adversary observing the number of orders being processed for intelligence purposes for example, is less disruptive to the defender than one that attempts to cause physical damage to the defender's resources. In addition, if the defender chooses an eviction attempt $\omega_l$ or an observational tactic $\phi_l$, the defender pays a cost $\kappa_l \in [0, 1]$. This modeling choice allows the OEG to model an important practical reality of defense, that certain defensive measures that might be more effective, e.g., shutting down an infected server, might come at a high cost to the defender's operation (C-3).

Therefore, at a leaf node of the game tree, if the attacker stays in the system for $T$ time steps, the attacker gets a utility of $T \cdot \alpha_{ij}$ and the defender gets a utility of $-T \cdot \delta_i - \nu$ where $\nu$ is the total cost of the defender actions. Note that if the game terminates with a successful eviction attempt, $T$ is the number of time steps that the attacker has been in the system so far, and if the game terminates due to a failed eviction attempt or an observed active measure, $T = \tau + F$. In this utility model, the utility for the defender depends on the type of the attacker (through the coefficient $\delta_i$), modeling that different attacker types may have goals resulting in varying degrees of damage to the defender.

Let $\Pi_i$ be the set of pure strategies for player $i$ in the game. Given the game model, the attacker's pure strategy set is the same as the set of TTPs, i.e., $\Pi_1 = \Gamma$. A pure strategy for the defender assigns an action for each information set at which the defender needs to take an action from $\Omega \cup \Phi$. Since the defender may choose to take an active measure at each time step without being noticed by the attacker, the set of pure strategies for the defender is exponential in size with respect to the number of active measures. When the attacker of type $\theta_i$ is following a pure

strategy $\gamma_j$ and the defender is following a pure strategy $\pi_k$ (the $k^{th}$ pure strategy in $\Pi_2$) the utilities for the players are nondeterministic due to the existence of the chance node. We use $u^1_{jk}$ and $u^2_{ijk}$ to represent the expected utility for the defender and the attacker respectively. To find the game equilibria, we derive the reduced normal form game [55] where actions at irrelevant information sets are omitted, and the computation of $u^1_{jk}$ and $u^2_{ijk}$ becomes necessary. Here we explain how these quantities can be computed.

A defender's pure strategy consists of a sequence of observational actions from $\Phi$, followed by an eviction action from $\Omega$. The set of pure strategies $\Pi_2$ can be enumerated by a recursive function $e(s, t, \tau)$, which, for each timestep $t \leq \tau$ adds the pure strategies where the defender uses an eviction action at timestep $t$ to the set $s$. For expository purposes, we use $a^t_k$ to denote the action that the defender plays in time step $t$ when following $\pi_k$. Let $\epsilon_{ijk}$ represent the total expected time that the attacker will stay in the system. To compute $\epsilon_{ijk}$, we introduce a helper quantity $\gamma^t_{ijk} \in \mathbb{R}$, which returns the probability that the game reaches timestep $t$, that is, the probability that the attacker is still in the system and has not noticed the defender's activity by timestep $t$. Since the defender takes at most one eviction action, let $T$ denote the timestep the eviction action is taken ($T = \tau + 1$ if no eviction action is taken) where $T \leq \tau$. Then the probability $\gamma^t_{ijk}$ can be obtained by the following equation:

$$\gamma^t_{ijk} = \begin{cases} \gamma^{t-1}_{ijk} \cdot (1 - q^{t-1}_j) \cdot (1 - y^{t-1}_j) & \text{if } 0 < t \leq T \\ 1 & \text{if } t = 0 \\ 0 & \text{if } t > T \end{cases}$$

Here we use $y^{t-1}_j$ to denote the probability that the observational tactic used in timestep $t-1$ is observed by the attacker, i.e., $y_{jl}$ where $a^{t-1}_k = \omega_l$. The equation can be interpreted as follows: Conditioned on that the game reached timestep t-1, the game can reach timestep t if (1) the defender has not taken an eviction action in previous timesteps (i.e., $t \leq T$); (2) nature does not reveal the attacker's true TTP to the defender (the second term); (3) the attacker does not notice the defender's observational tactic (the third term). Note that the second term depends on the defender's chosen strategy $\pi_k$, since an active measure might modify $q$.

The attacker's total expected time in the system is then given by summing the product of the probability of each possible outcome with the time the attacker would

remain in the system if that outcome were to occur. This is given by:

$$\epsilon_{ijk} = \sum_{t=1}^{T-1} \left( \gamma_{ijk}^{t} \cdot \left( y_j^t \cdot (F+\tau) + \left(1 - y_j^t\right) \cdot q_j^t \cdot t \right) \right) + \gamma_{ijk}^{T} \cdot \left( \chi_{jl} \cdot T + \left(1 - \chi_{jl}\right) \cdot (F+\tau) \right)$$

This expression sums the product of the probability of occurrence and the number of timesteps that the attacker remains in the system over each timestep. The inside of the summation handles all but the final timestep, which is treated differently since the defender's last action is always an eviction tactic, while the defender's other actions are always observational tactics. Inside the summation, the first term is the probability that the game has not already ended in previous timesteps. The game can end in the current timestep in two ways. The first is by the attacker noticing the defender's active measure, which results in the failed eviction penalty number of timesteps, handled by the term $y_j^t \cdot (F+\tau)$. The second way the game can end is by the defender learning the attackers TTP and evicting them on the current timestep, which is captured by the term $(1-y_j^t) \cdot q_j^t \cdot t$. Note that the this can only occur if the attacker has not noticed the observational tactic, hence the need to multiply by one minus the chance that the attacker observes the defender, and the term $q_j^t$ depends on the defender's strategy since it may have been modified by an active measure in an earlier timestep. The term outside the summation handles the eviction action at the end of the defender's chosen strategy. This is the probability that the game does not end before this timestep, multiplied by both possible outcomes, either the eviction succeeds and the attacker is evicted on timestep $T$, given by $\chi_{jl} \cdot T$, or the eviction fails resulting in the failed eviction penalty, $(1-\chi_{jl}) \cdot (F+\tau)$.

The defender's utility $u_{ijk}^1$ is given by the expected time the attacker remains in the system, modified by how disruptive the attacker type $i$ is, i.e., $\delta_i$. The defender also pays a cost dependent on which defensive action is performed based on how disruptive it is $\kappa_l$.

Since a defender strategy that involves non-evicting actions may result in using different eviction actions depending on whether the attacker is observed by the defender, the expected cost can be found in a similar way to the expected time that the attacker remains in the system, summing the product of the probability of each outcome by the cost of that outcome, given by $\nu_{ijk} \in \mathbb{R}$. Due to the similarity of this equation to $\epsilon_{ijk}$, we omit the details of this function. The defender's utility is then given by $u_{ijk}^1 = -\epsilon_{ijk} \cdot \delta_i - \nu_{ijk}$.

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

⟨*operator*⟩ ::= 'R' ⟨*double*⟩ ⟨*plan*⟩ ⟨*plan*⟩ (Randomize)
| ';' ⟨*plan*⟩ ⟨*plan*⟩ (Sequence)
| 'I' ⟨*conditional*⟩ ⟨*plan*⟩ ⟨*plan*⟩ (If)

Figure 5.2: Grammar for specifying plans with the co-evolutionary extension.

### 5.1.3 Computing Equilibria

In this thesis, we are interested in the Nash equilibrium, although we also investigated solving for the Strong Stackelberg equilibrium in other work [30] since it is often hard to know how much the attacker knows about the defender's strategy. To solve the game when neither side knows the strategy profile of the other, we use the Gambit software tools for game theory [46] to obtain the Nash Equilibrium. The AI community has made significant progress towards solving Stackelberg games efficiently [58, 73, 40], and we leave the investigation of more efficient solution approaches for the OEG to future work.

## 5.2 Co-evolutionary Extension

Competitive co-evolution is a genetic search approach that evolves multiple populations of individuals at a time. Unlike in traditional evolution where the fitness of an individual depends soley on that individual's characteristics, fitness depends on an interaction between individuals from different populations. This allows the individuals in each population to adapt in response to each other. While the Omnet cloud-based web server case study has served as a running example throughout the thesis thus far, the case study system does not consider any adversarial interactions. To evaluate the applicability of plan reuse with stochastic search in adversarial settings, we introduce a new case study system called Bullseye, a business enterprise network under attack from an advanced persistent threat inspired by the Target data breach [38]. A detailed description of this case study is provided in Section 6.2.2. While in principle co-evolution can be applied to an arbitrary number of populations, we make a simplifying assumption that there is one attacker and one defender, and evolve a population of candidate strategies for each side.

### 5.2.1 Individual Representation

We represent the planning problem as a two population co-evolutionary search, one population to evolve a strategy for the defender, and one population to evolve a strategy for the attacker. Since the available actions for each agent is different, a different representation for each population is required, and the specific representation is domain specific. However, there are some general principles that are generalizable between domains. Figure 5.2 shows the generalizable Backus-Naur form grammar for specifying the individual representations. The specific conditionals and tactics must be specified for each domain.

Agents have access to a new randomization operator, which allows the agents to non-deterministically choose between courses of action based on a specific set probability. Randomization is an important concept in game theory [55] and allows agents to reduce their predictability by randomly changing their behavior. Additionally, there is a theoretical result that at least one Nash equilibrium always exists using randomized strategies [53]. The randomization operator executes the first subplan with a probability equal to the specified value, and the second subplan otherwise. An `if` operator is provided to allow agents to change their behavior based on information they have about the current game state. This operator checks the state of the system with a condition, and executes the first subplan if the condition is true, and the second subplan otherwise.

**Algorithm 2** How fitness is assigned to each individual in the co-evolutionary search.

```
1: for subpop in subpops do
2:    for individual in subpop do
3:       individual.results = []
4:       for competitionSelector in competitionSelectors do
5:          for i++ < competitionSelector.number do
6:             competitor = competitionSelector.getCompetitor()
7:             individual.results.append(fitnessFunction(individual, competitor))
8:          end for
9:       end for
10:      individual.fitness = aggregate(individual.results)
11:   end for
12: end for
```

### 5.2.2 Fitness Calculation

The fitness of individuals in both populations is evaluated by performing a series of competitions with individuals from the competing population, and aggregating the results. Algorithm 2 shows how fitness is assigned to each individual. On line four, a collection of competition selectors is used to to choose competitors in a number of ways. We use four selectors, which include: (1) six individuals randomly selected from the current generation, (2) five individuals selected by tournament selection from the previous generation, (3) five guru individuals from the previous generation, and (5) ten individuals randomly selected from a hall of fame. The five guru individuals are the individuals with the highest fitness from the previous generation, while the hall of fame consists of the best individuals from every previous generation. The hall of fame concept [56] is one proposed means of mitigating a common failure mode of co-evolutionary searches where populations become trapped in a cycle of local optima rather than continually make global progress in the solution space. The hall of fame accomplishes this by calculating fitness not only against the current generation, but also against the best individuals from historical generations, thus promoting progress against all historical solutions rather than only the previous generation's solutions, although this approach is not guaranteed to enforce global progress. These selection strategies were selected to balance obtaining individuals in a variety of ways with evaluation time based on some preliminary experiments. The instantaneous fitness on line seven is calculated by performing a Monte Carlo simulation of 500 trials.

On line ten in Algorithm 2, the results of the many competitions are aggregated to obtain a single fitness value. The aggregation function is the minimum when calculating the defender's fitness and the average when calculating the attacker's fitness. Preliminary experiments showed that this arraignment resulted in the best global progress for the defender's strategy, while using the average for both populations resulted in the search becoming trapped in cycles of local optima, and even when existing strategies are reused they can quickly "de-volve" to this degenerate state.

### 5.2.3 Reuse and Repertoire Generation

Strategies are reused by seeding both populations of individuals with previously existing strategies. This is done for each subpopulation using a ratio of reused individuals to individuals randomly generated from scratch. Reusable repertoires can be constructed for this case study using the same high-level chaos engineering idea as described in Section 4.1, with a few implementation changes.

The co-evolutionary extension is evaluated in Section 6.3.5. The results show that plan reuse using a repertoire resulted in a large improvement, particularly during early generations of planning where the improvement was as large as 92%. Reusing a single strategy only also resulted in an improvement over planning from scratch, although less pronounced.

# Chapter 6

# Validation

This chapter validates the claims of the thesis expressed in the thesis statement given in Section 1.1 and restated below.

> *We can enable self-* systems with large state spaces to evolve in response to unexpected changes by reusing existing plans with stochastic search in the following three ways: (a) reusing existing plans using genetic programming and reuse enhancing approaches to reduce evaluation time, (b) building reusable repertoires by identifying generalizable plan fragments to build resilience against unexpected changes, and (c) reusing strategies in adversarial settings.*

Chapter 3 presented an overview of the approach for reusing plans with stochastic search, while Section 3.5 described our approaches for reducing evaluation time discussed in (a). Chapters 4 and 5 presented extensions for building and reusing repertoires of adaptation strategies, and for performing reuse in adversarial settings as described in (b) and (c) respectively.

The contributions of the thesis will be evaluated based on the following three claims, which are further described in Section 6.1:

1. Plan reuse will lower the number of generations until convergence to a good plan.
2. Plan reuse will decrease the wall-clock time needed to generate a good plan compared to planning from scratch.
3. Plan reuse is applicable to a range of unexpected change scenarios, including adversarial settings.

These claims are evaluated on three case study systems, described in detail in Section 6.2:

1. A cloud based web server.

Table 6.1: Table of approaches and representation in the case studies.

| | Omnet | DART | Bullseye |
|---|---|---|---|
| Core Approach | Yes | Yes | Yes |
| Reuse Enablers | Yes | Where appropriate | Where appropriate |
| Reusable Repertoires | Yes | No | Yes |
| Co-evolutionary Extension | No | No | Yes |

2. The DART team of autonomous aerial vehicles.

3. An enterprise system under attack by advanced persistent threats.

To validate claims 1 and 2, we present results on planning effectiveness and timeliness for each of the contributions described in Chapters 3, 4, and 5. To validate claim 3, we evaluate the thesis on three unique case study systems, each facing a diverse range of unexpected changes, including a system under attack from an advanced persistent threat.

Table 6.1 shows the main approaches presented in the thesis and which case studies each approach is evaluated on. The core approach to plan reuse of seeding adaptation strategies represented as tree structures to the population of a GP is evaluated on all case studies. The reuse enabling approaches presented in Section 3.5 are evaluated in depth on the Omnet case study, where the presence of tactic failure resulted in large complex tree structures for which the reuse enables were designed to handle. Where applicable, reuse enablers are also utilized in the other case studies. Techniques from Chapter 4 on building reusable repertoires are utilized in both the Omnet and Bullseye case studies. These approaches were not studied on the DART case study due to the unique property of DART's search space, which allowed for order of magnitude improvements when reusing a single adaptation strategy (see Section 7.1.3 for an in depth discussion of this issue). The co-evolutionary extension is only applicable for systems where security is a concern, which is only present in the Bullseye case study.

## 6.1 Claims

The goal of the thesis is to improve the ability of self-* systems to respond to unexpected changes with plan reuse. Ideally, a planner should generate the plan that obtains the highest possible utility for the system, and generate this plan instantly after a change occurs. In practice however, the complexity of self-* systems results

in large search spaces that are often infeasible to exhaustively explore, and require planners to make tradeoffs between solution quality and timeliness. Thus, to show that plan reuse results in more effective planning, the evaluation will show how reuse impact the timeliness and quality of planning in response to unexpected changes. Whether the goal of improving planning quality is achieved will be evaluated by the following three claims:

1. Plan reuse will lower the number of generations until convergence to a good plan.
2. Plan reuse will decrease the wall-clock time needed to generate a good plan compared to planning from scratch.
3. Plan reuse is applicable to a range of unexpected change scenarios, including adversarial settings.

In the remainder of this Section, I will elaborate on each of these claims and specify how they will used to evaluate the thesis along with the three case study systems.

### 6.1.1 Plan reuse will lower the number of generations until convergence to a good plan.

This claim evaluates the basis on which plan reuse in stochastic search can be expected to result in an improvement on planning from scratch. The idea is that by seeding the search with individuals that have previously been effective, the search will develop more effective plans more quickly, and will converge to a good solution in a fewer number of generations compared to planning from scratch. Evaluating this claim requires specifying exactly what is meant by a "good" solution. While stochastic solutions like the one presented in this thesis have scalability advantages compared to exhaustive approaches, they are not guaranteed to find the optimal solution. In practice however, an optimal solution is not often necessary, and obtaining a satisfactory answer in a reasonable amount of time is preferable.

In this work, we are interested in obtaining plans that suffice. Whether a plan is satisfactory or not is a complex issue that is domain and context dependent, and depends not only on the quality of the plan that is produced, but the amount of time taken to develop the plan. In some situations, for example, if a drone is about to collide with an obstacle, a plan that avoids the obstacle in a time and energy inefficient manner, but is generated quickly, is much better than a higher quality plan that cannot be generated in time to avoid the obstacle [57].

Anytime planning approaches (such as GAs) provide flexibility in determining

how much time to spend planing, since they can always be stopped and the best available solution taken; however, the issue of deciding when to stop planning is outside of the scope of the thesis. This issue is discussed in more detail in Section 7.2. There are several ways of establishing criteria that specify when a good plan is obtained. These include a plan being within some threshold percent difference compared to a benchmark such as the optimal value discovered from exhaustive planning or the highest value obtained during a high-budget heuristic search. Another criteria is the percent change in utility from before and after a plan is executed being higher than a threshold. Another possible approach that attempts to take planning time into the equation is examining the area under the utility curve over time. Since the notion of a good plan is domain dependent, good plans will be assessed on a case-study basis.

### 6.1.2 Plan reuse will decrease the wall-clock time needed to generate a good plan compared to planning from scratch.

If plan reuse reduces the number of generations needed to find a good plan, then it should be possible to obtain a better plan more quickly. This is especially important if the approach is to be used online rather than offline. However, the number of generations of planning is an imperfect indicator of planning time. There are several reasons that could cause the actual time spend planning to vary independently of the number of generations. To complete a generation of planning, the planner must evaluate the fitness of the individuals in the population, and this time may vary from generation to generation. For plan reuse to improve the effectiveness of planning, the amount of wall-clock time needed to arrive at a good plan must be lower than planning from scratch. Unfortunately, the time needed to evaluate the fitness of large precomputed plans may be longer than the time needed to evaluate short plans generated from scratch. An important aspect of the approaches described in the thesis is developing mitigations for this problem, including strategies for speeding up the evaluation time and by identifying the most promising plan fragments to reuse.

### 6.1.3 Plan reuse is applicable to a range of unexpected change scenarios, including adversarial settings.

In addition to showing an ability to improve the effectiveness of planning, the approach should also generalize to a range of unexpected change scenarios, including adversarial settings. While it may not be possible to always obtain a significant

Table 6.2: A comparison of the case study systems.

| | Omnet | DART | Bullseye |
|---|---|---|---|
| Tactics can fail | Yes | No | No |
| Planner executions | Once | Every timestep | Once |
| Plan execution | Entire plan | First tactic only | Entire plan |
| Changing environment | Before planning | Every timestep | Before planning |
| Adversarial | No | No | Yes |

improvement, the thesis will characterize the types of situations when improvement can be expected. This claim will be validated by demonstrating that the approaches presented in the thesis can be applied successfully to three unique case study systems, including a system under an attack from an advanced persistent threat. While not all techniques are utilized in all case studies, the successful application of reuse to all three diverse case studies and in response to a wide range of unexpected change scenarios provides validation for this claim. A discussion of the lessons learned from this diverse validation including when reuse is likely to result in an improvement is presented in Chapter 7.

## 6.2 Case Study Systems

The thesis is evaluated on three case study systems. The case studies were selected to be representative of several types of self-* systems from different domains, and with different planning assumptions, quality attributes, and types of changes. Table 6.2 compares the three case study systems from a planning perspective. The first case study system is the cloud web server described in detail in Section 3.1, and includes an environment where tactics may fail, and planning is performed only once. The second case study system is DART, a team of autonomous aerial vehicles that need to navigate a hostile environment to detect targets. Apart from the different domain, tactics are assumed to be reliable, and planning is repeated over a series of time steps. The final case study system is a business enterprise system attempting to defend itself from advanced persistent threats. This case study provides a self-* system where security is a primary quality attributes, and allows us to investigate plan reuse in an adversarial setting with an attacker who can also affect the system. The first cloud web server system is described in Section 3.1, this chapter describes the remaining case studies in detail.

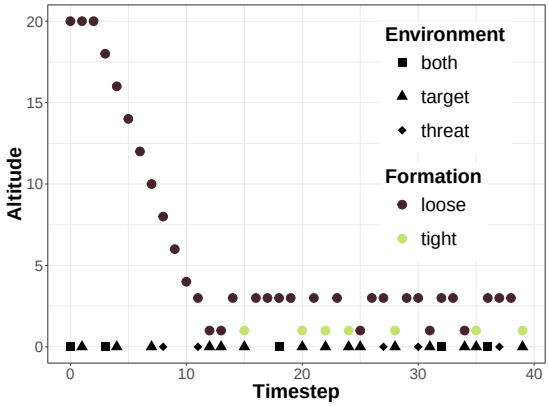


Figure 6.1: An example trace of the DART team moving through an environment.

### 6.2.1 DART

The second case study system, DART [51] is inspired by a scenario from the DART systems project [20], using the same modeling approach and parameters as related work [49]. 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. 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. Lastly, the team can enable electric counter measures (ECM), which attempts to confuse threats by overwhelming their sensors, decreasing 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 6.1 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.

### Utility

The drone 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.

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 in order 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. A complete treatment of how $d$ and $g$ are computed is provided in prior work [51].

### 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.

#### Change Scenarios

We examine three types of change scenarios for this case study: changes to the positions of the threats and targets present in the environment, changes in the available adaption tactics, and changing the desired utility tradeoff between the survivability of the team and the expected number of targets detected.

### 6.2.2 Bullseye

To study reusing repertoires of adaptation strategies with stochastic search in an adversarial setting, we introduce a new case study system called Bullseye. Bullseye models an APT attacker infiltrating a business enterprise system, inspired by the Target data breach [75, 38], and the Observable Eviction Game [30] (Section 5.1).

#### Architecture

Figure 6.2 shows the Bullseye case study system at a high level. The system, controlled by the defender, consists of a web server, a payment server, and a collection of point of sale (POS) devices. An APT attacker begins outside of the system, and seeks to gain presence in the system's assets to exfiltrate valuable information. Due to providing web services, the web server is more vulnerable to exploitation than the more protected payment server. Separate credentials are required for authorized users to access either server, with more users including outside vendors having access to the web server while a more limited set of users has access to the payment server. The POS devices cannot be accessed directly, but periodically download firmware updates from the payment server, which allows the attacker to exploit the POS devices if they have control of the payment server.

The architecture of the case study is inspired by the Target data breach [75, 38], where an APT attacker succeeded in exfiltrating data for millions of customers' credit card accounts by compromising POS devices. In the Target data breach, the attackers were able to obtain access to a web server after phishing the credentials

from a vendor. The attackers then moved laterally to a payment server that POS devices downloaded updates from, allowing them to compromise many POS devices.

Bullseye also draws inspiration from the Observable Eviction Game (OEG) [30], a game proposed to model the interaction between an APT attacker and defender when the defender cannot directly observe the attacker and must decide between gathering more information about the attacker or attempting to evict them from the system. Bullseye extends the OEG by considering specific attacks and paths that the attacker can take through the system. In Bullseye we assume there is only one type of attacker, while the OEG can support reasoning about multiple types of unknown attacker. Bullseye makes the following assumptions: the defender and attacker both know the available actions and utility function of the other, neither side can directly observe the actions of the other, and the attacker knows what exploits and passwords they have obtained (including losses of the same caused by actions of the defender). We model Bullseye as a turn based game that takes place over two timesteps.

In this work, our goal is to study the reusabilty of repertoires of prior strategies, and we designed the case study to balance realism with keeping the game size small enough to analyze using the Gambit [46] solver. We note that an advantage of stochastic approaches is scalablity, however objectively measuring global progress in the search is non-trivial, and we use the solver to provide an objective comparison between the approaches in the evaluation.

### Utility

The attacker's goal is to exfiltrate valuable information from the system. The attacker exfiltrates information by maintaining presence in the system's assets. For each of the three zones shown in Figure 6.2, the attacker earns a reward for having presence in that zone for each timestep. The reward the attacker earns depends on the value of the information in that zone to the attacker, and the rate that the attacker can exfiltrate the information (which can be influenced by the defender). Given that the attacker is in the system for $T$ timesteps, an equation for the attacker's utility $U_a$ is given below:

$$U_a = \sum_{i=1}^{T} h^i \cdot \sum_{j=1}^{Z} p^{ij} \cdot v^j$$

The first summation in the equation sums the reward that the attacker generates over the $T$ timesteps that they remain in the system. The $h^i$ term is a value between 0 and 1 inclusive that represents the slowdown to the attacker's exfiltration speed caused by the defender at timestep $i$. The second summation sums the reward the

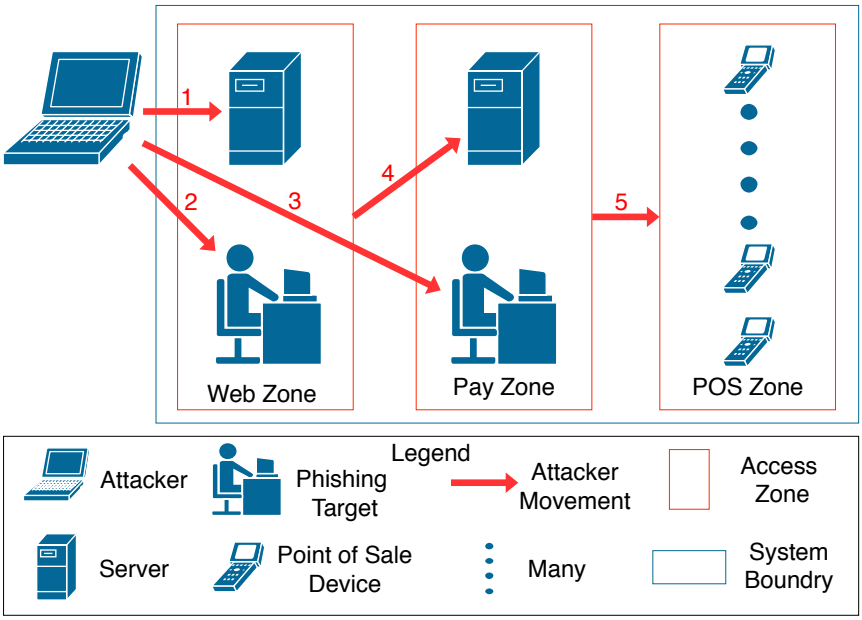


Figure 6.2: An overview of the Bullseye case study system, showing the assets under the system's control, and the attacker's available paths to move through the system.

attacker gets from each of the $Z$ zones. The reward for each zone $j$ depends on whether the attacker has presence in that zone for that timestep, represented by $p^{ij}$ which is either 1 or 0, multiplied by the value of the information in that zone to the attacker, given by $v^j$, a real number at least 0. The value of information in each zone to the attacker $v^j$ is set at the beginning of the interaction and does not change. The attacker's presence at each time in each zone $p^{ij}$ changes based on the attacker's and defender's chosen actions.

The attacker's observability to the defender is modeled in line with the Observable Eviction Game [30], where after each timestep $i$ the defender with some probability $q^i$ observes the attacker and can then immediately evict them from the system. In this case study, $q^i$ depends only on the attacker's chosen action at timestep $i$, with each action having its own observability value, a percentage that denotes the probability that the defender observes the attacker performing the action. A lower observability means that an action is stealthier. Thus, the observability of the attacker's selected actions influences how long the attacker remains in the system $T$ in the above equation.

For simplicity and faster experimentation time, we model Bullseye as a zero-sum game, which means that the defender's utility $U_d$ is the inverse of the attacker's

Table 6.3: Description of attacker tactics.

| Number | Name | Attacker gains | Prereqs | Observability |
|---|---|---|---|---|
| 1 | Exploit Web Server | web exploit | None | 5% |
| 2 | Phish Web Vendor | web password | None | 10% |
| 3 | Phish Employee | pay password | None | 20% |
| 4 | Exploit Pay Server | pay exploit | web presence | 5% |
| 5 | Exploit POS | POS exploit | pay presence | 10% |

utility $U_d = -U_a$. Intuitively this means that the defender's objective is to limit the attacker's utility. In real systems the defender also cares about limiting disruption to their operations. For example, the defender can always stop the attacker by turning off the system, but this is almost always unacceptable in practice. The Observable Eviction Game [30] explores this complication in greater detail.

### Adaptation Tactics

The red lines in Figure 6.2 correspond to the actions available to the attacker to gain presence in the system's assets. These actions consist of exploits, where the attacker can take advantage of a vulnerability in the system to gain presence in a vulnerable asset given the preconditions are met, and phishing attacks, where the attacker can manipulate privileged users to give up their credentials. Table 6.3 lists each of the five actions available to the attacker. The attacker starts with no presence in the system, and can take any of the first three actions. The attacker can gain presence on the web server by using an exploit on the web server (action 1), or by gaining a password to the web server by phishing an outside vendor (action 2). Alternatively, the attacker can gain presence on the pay server by phishing a password from an employee (action 3); however, since employees are closer to the company and better trained to report phishing attacks, this action is much more noticeable to the defender. If the attacker has presence on the web server (either by exploiting it or obtaining a password), the attacker gains the ability to exploit the payment server, (action 4), a stealthier alternative to the phishing approach. If the attacker has presence on the payment server, the attacker can exploit the POS device firmware to gain presence in the POS devices (action 5).

Table 6.4 shows the three actions available to the defender. The defender can revert the system to a previous safe state (action 1), which results in the attacker losing any exploits, but keeping passwords. The defender can also change the passwords

Table 6.4: Description of defender tactics.

| Number | Name | Description |
|---|---|---|
| 1 | Flash Servers | Attacker loses all exploits |
| 2 | Change Passwords | Attacker loses all passwords |
| 3 | Throttle | Attacker utility reduced |

(action 2), which causes the attacker to lose any passwords obtained from phishing, but any exploits are still installed. Lastly the defender has the ability to toggle on a throttle, which reduces the system's data transmission rate (action 3). When the throttle is on, the attacker's exfiltration rate is halved $h^i = 0.5$ and is 1 otherwise. The attacker has presence in a zone $p^{ij} = 1$ if the attacker has either an exploit or a password in that zone. Passwords and exploits persist across timesteps until they are removed by a defender action. At each timestep, attacker presence is determined by applying the attacker action first, and then the defender action. For example, if on the first timestep the attacker phished the vendor and the defender changed the passwords, the attacker would have no presence for utility calculation.

#### Change Scenarios

Bullseye supports eight attributes that can be modified to produce new scenarios. The first five values specify the observability of each of the attacker's actions to the defender. The final three attributes are the value of the information in each of the three access zones to the attacker. The starting values for the observability attributes are provided in Table 6.3. The value attributes are initialized to 2.

### 6.2.3 Summary

To evaluate the presented approach for plan reuse with stochastic search, three case study systems are used. These systems are a cloud based web service provider inspired by Amazon AWS, DART, a team of autonomous aerial vehicles, and Bullseye, a business enterprise network inspired by the Target data breach. These case studies were selected to evaluate the approach on a number of diverse systems with different goals and planning assumptions.

## 6.3 Evaluation

This section presents the empirical evaluation of the presented approaches. Section 6.3.1 focuses on the core approach and reuse enabling methods presented in Chapter 3. Sections 6.3.2, 6.3.3, and 6.3.4 evaluate the approaches for reusing repertoires of adaptation strategies from Chapter 4. Lastly, Section 6.3.5 evaluates the co-evolutionary extension presented in Chapter 5.

### 6.3.1 Core Approach and Reuse Enablers

Chapter 3 described an approach for replanning after unexpected changes occur in self-* systems using plan reuse and stochastic search. While intuitive, preliminary results showed that naïve reuse can actually result in worse results than replanning entirely from scratch. In this section we evaluate the core approach, including three reuse enabling approaches described in Chapter 3 to mitigate this issue. In this evaluation, we compare plan reuse with and without our reuse enabling approaches with an exhaustive planning approach, as well as replanning entirely from scratch. We built the genetic programming planner described in Chapter 3 on ECJ, a framework for evolutionary computation in Java[1].

We evaluate the core approach for plan reuse using two case study systems: Omnet, a cloud-based web server described in Section 3.1, and, DART, is a team of autonomous drones that must detect targets in a hostile environment while avoiding threats, described in Section 6.2.1. These case studies highlight reuse in different domains and with different planning assumptions. Section 6.3.1 describes the experimental setup and reports results for the Omnet case-study. Section 6.3.1 does the same for the DART system.

#### Omnet Evaluation

For the Omnet evaluation, we evaluate various change scenarios based on the system shown in Figure 3.1 and described in Section 3.1. First, as a sanity check to ensure that the planner is producing reasonable results, and as a way of tuning the many parameters of the approach, we present a comparative study between the GP planner and an exhaustive approach. We then move on to evaluating the key claims for this section, that plan reuse with the reuse enabling approaches results in fewer generations until convergence to a good plan, as well as less wall clock time.

[1]ECJ is available at `https://cs.gmu.edu/~eclab/projects/ecj/`.

The evaluation is performed on a simulator implemented in Java based on the description of the case study system in Section 3.1, and implements the fitness function described in Section 3.4. 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 ten 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 seconds. Where we compare to searches from "scratch", we use Koza's ramped half-and-half [37] algorithm for constructing random trees to initialize the population.

**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 [57], 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 crossover, mutation, and reproduction/copying)

[2]Because the Pandey et al. approach [57] 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 6.5: 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 |

and number of generations, population size, and all penalty thresholds (Section 3.4). We generated plans for the system's initial configuration (Section 3.1), 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 6.5 shows the parameter values covered in the sweep.

The dark point at the top of Figure 6.3 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). An example plan that achieved close to the optimal utility is shown in Figure 6.4. The best configuration that produced plans in 0.50 seconds 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.

**Comparative Study: 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, which caused the model states to grow exponentially following the equation $(6 \mathit{servers\_per\_type} \cdot 5 \mathit{possible\_dimmer\_values} \cdot 5 \mathit{possible\_traffic\_values})^t$.

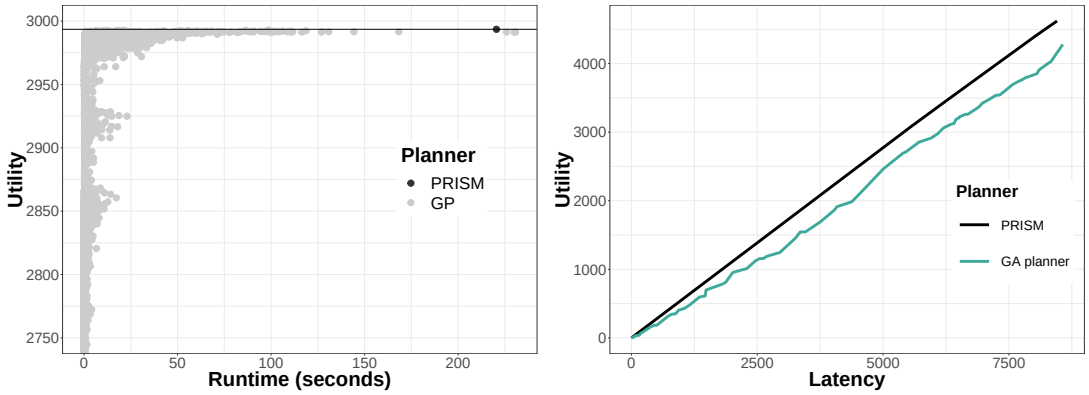


Figure 6.3: Left: Utility versus planning time for GP parameter configurations. Many configurations produce similar utility results to PRISM, significantly faster. Right: Pareto fronts for utility (higher is better) and latency (lower is better) from both planners.

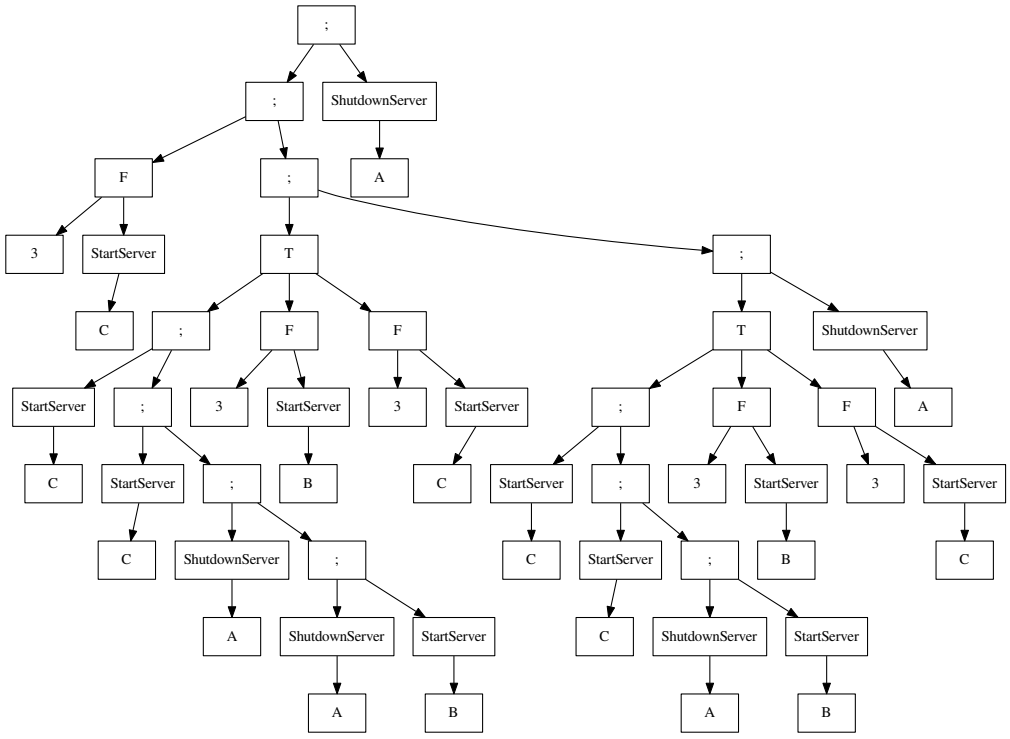


Figure 6.4: An example plan generated for the cloud web server case study.

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 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.

**Comparative Study: 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 right of Figure 3.4 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 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 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.

### 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 3.5), 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 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 6.6 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 presented enhancements to GP planning can reduce this cost and achieve higher utility than planning from scratch.

### Unforeseen Adaptation Scenarios

We now 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 assessing the planner's ability to respond when planning with reuse compared to planning from scratch. The considered scenarios are described in Section 3.1, and

Table 6.6: Improvement obtained by reuse enabling techniques.

| Planning Technique | % Reused | Utility | P Value |
|---|---|---|---|
| Scratch | 0 | 1.000 | |
| Scratch & *kill_ratio* | 0 | 1.044 | $< 0.01$ |
| Reuse | 100 | 0.962 | 0.06 |
| Reuse & *kill_ratio* | 100 | 1.072 | $< 0.01$ |
| Reuse & *kill_ratio* & *scratch_ratio* | 10 | 1.077 | 0.63 |
| Reuse & *kill_ratio* & *scratch_ratio* & *trimmer* | 10 | 1.112 | $< 0.01$ |

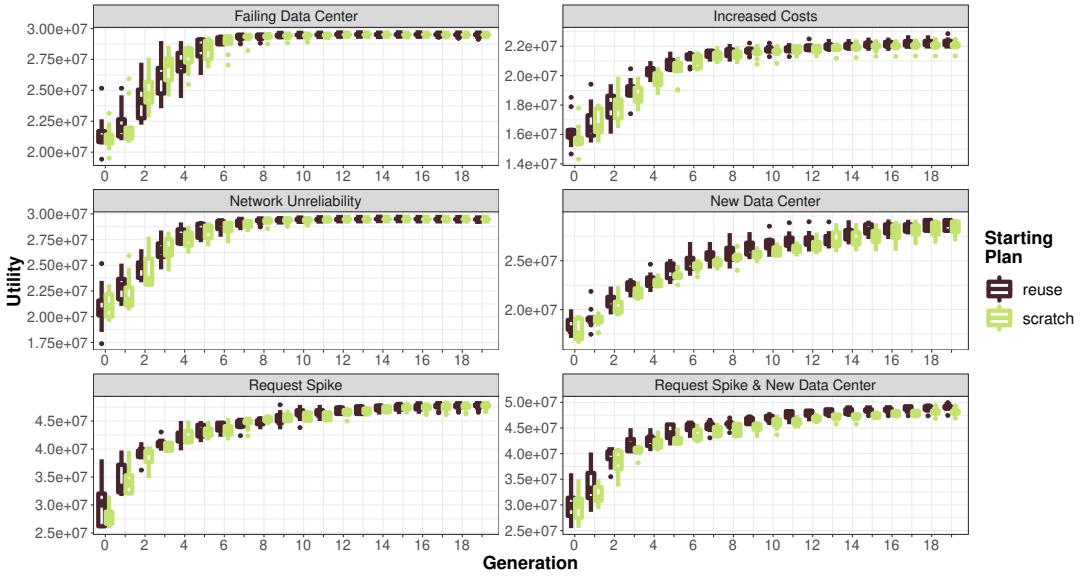


Figure 6.5: Utility versus generation for all six scenarios.

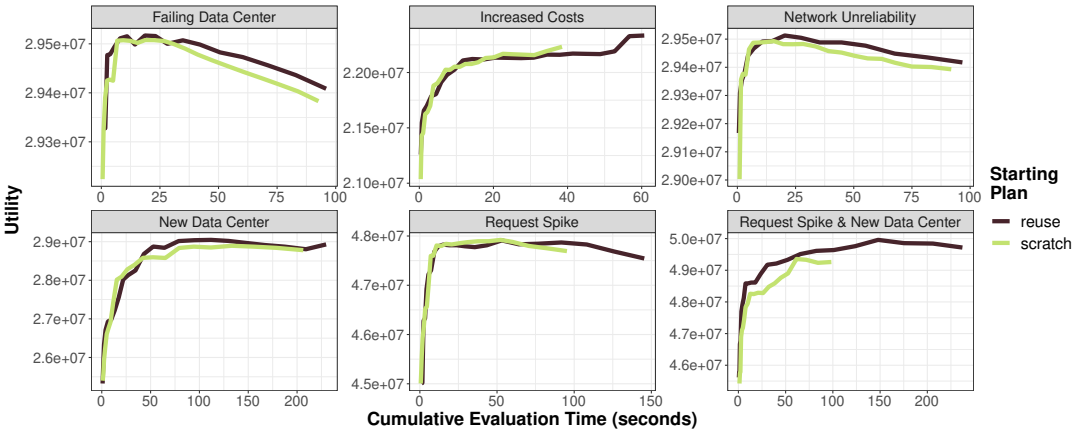


Figure 6.6: Utility versus cumulative runtime for all six scenarios.

Table 6.7: 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** |

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 3.1). 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. When performing reuse, all three reuse-enabling techniques are used.

Table 6.7 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 plan-

ning 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 6.5 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 [71]). 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 imperfect proxy for run time. Thus, Figure 6.6 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.

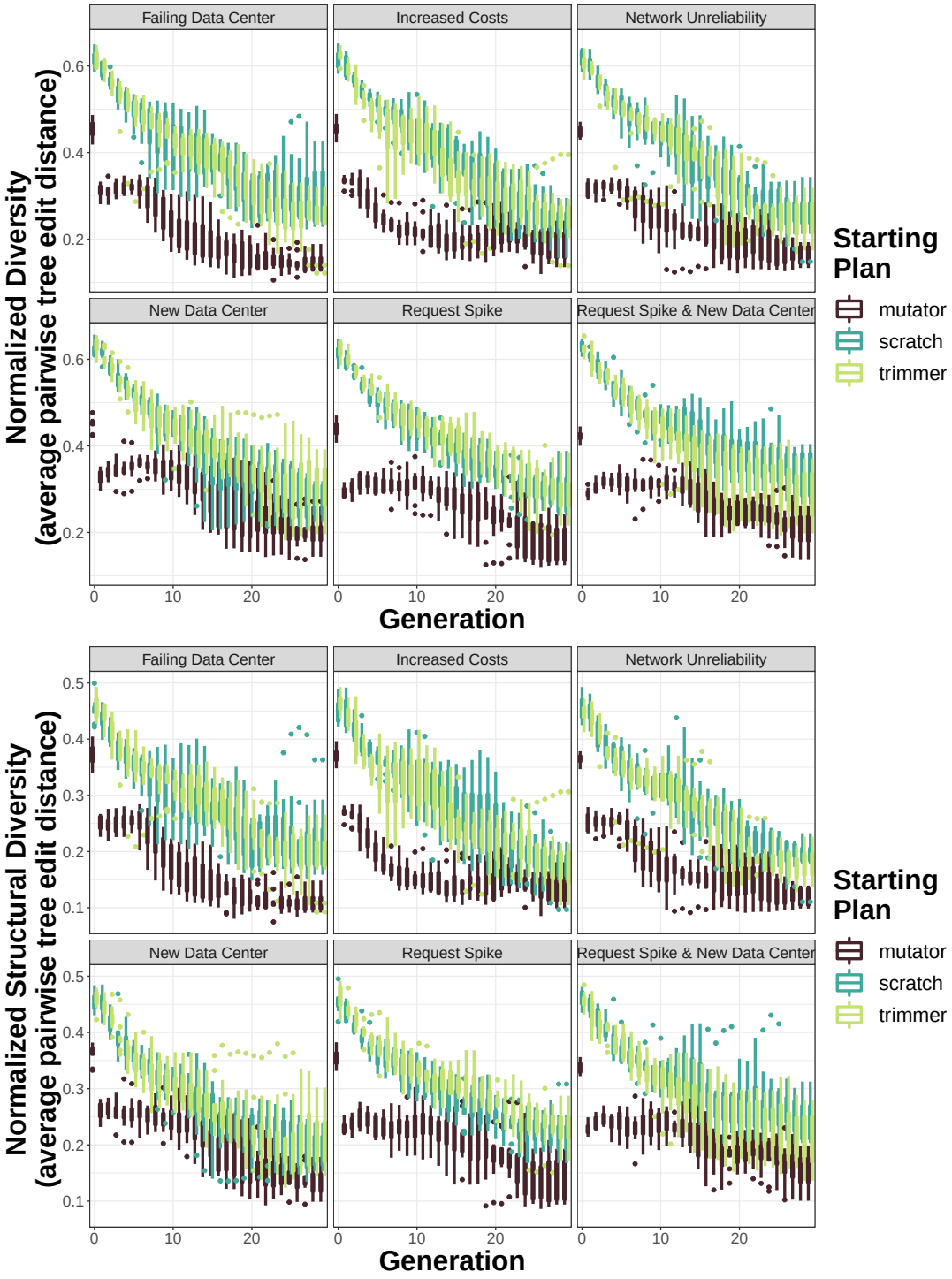


Figure 6.7: Diversity versus generation for all six scenarios.

### 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* [45], 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 [59] (a state of the art approach that uses dynamic programming to obtain polynomial runtime).

Figure 6.7 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 towards a high quality solution.

As shown in Figure 6.7, 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 as the population evolves.

### 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 [51] to accommodate the unexpected change scenarios introduced in our study and to integrate with the GP planner. 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. Lastly, 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

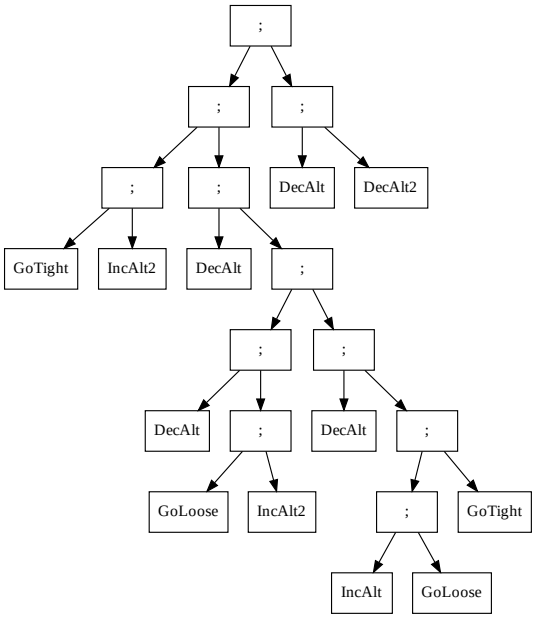


Figure 6.8: An example plan generated for the DART case study.

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 6.2.1 describes the DART system, including how we adapt our approach from Chapter 3 to handle the new case. Section 6.3.1 describes the results of the experiments involving DART.

**Integration with GA Planner.** The approach for using the GA planner is largely the same as described in Chapter 3, 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. On the other hand, the DART case-study

Table 6.8: The parameter settings in the parameter sweep.

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

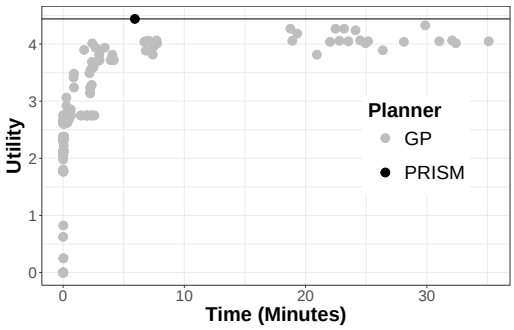


Figure 6.9: Utility versus planning time for GP configurations.

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 6.8 shows an example plan generated for the DART case study.

### 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 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 (PMC) presented in related work [47]. 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 6.8 shows the parameter values used in the sweep. Figure 6.9 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 1000 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.

### 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 ten 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 ten 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 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 towards 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 6.10 and 6.11 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).

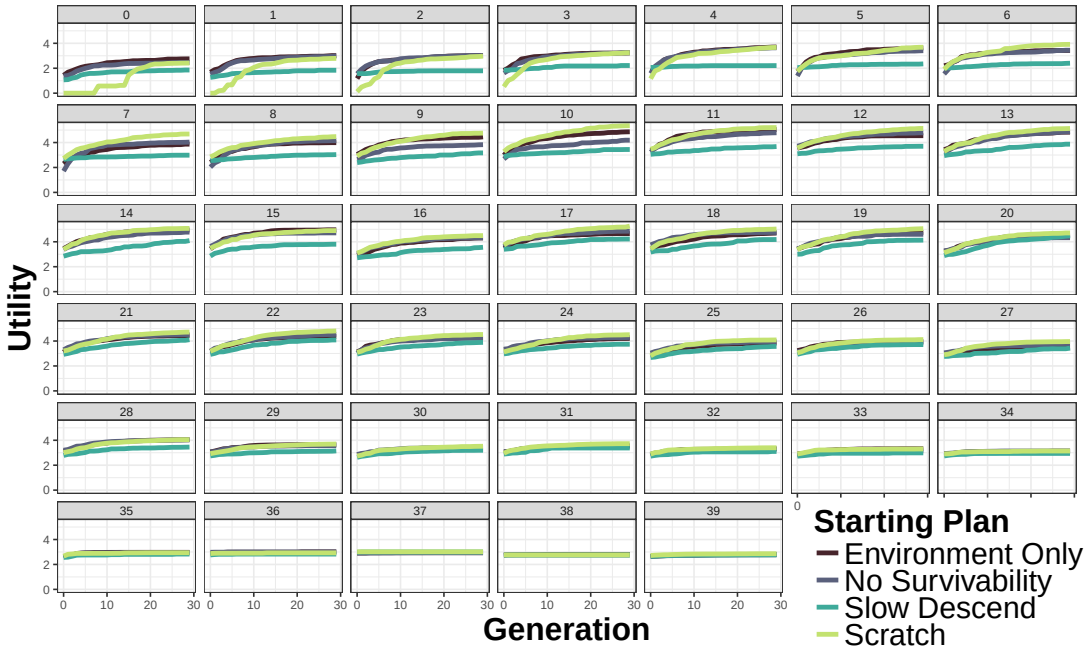


Figure 6.10: Utility versus generation by timestep.

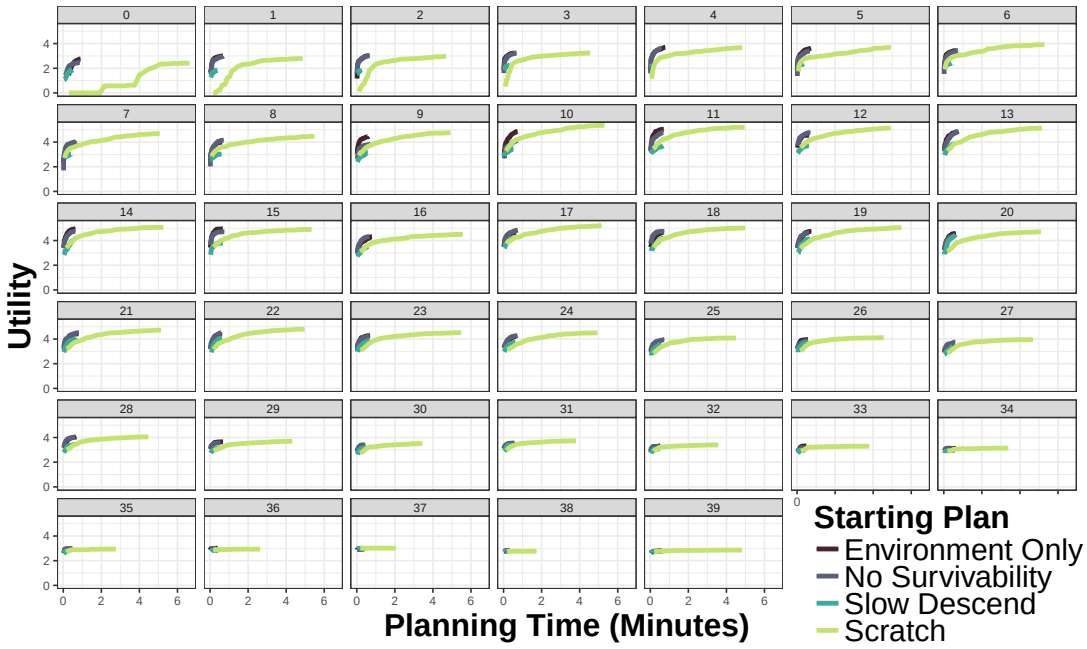


Figure 6.11: Utility versus runtime by timestep.

Figure 6.10 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 6.11 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 seconds of planning. The `Environment + Slow Descend` scenario sometimes outperformed planning from scratch in the first 30 seconds of planning, but usually underperformed compared to the other reuse approaches.

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.

**Expected Utility** Figure 6.12 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 6.13 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 mod-

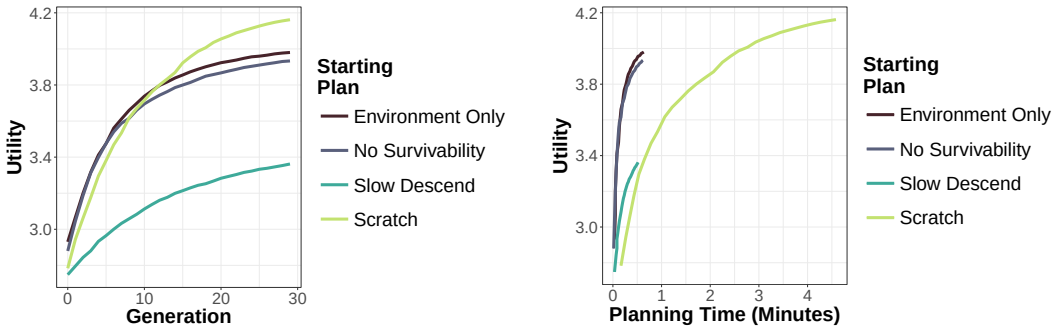


Figure 6.12: Left: Utility versus generation. Right: Utility versus planning time

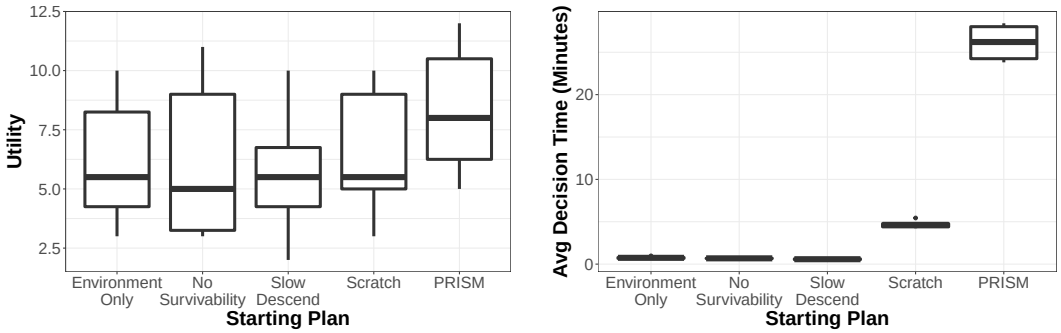


Figure 6.13: Left: Aggregate utility versus planning scenario. Right: Aggregate decision time versus planning scenario.

ifiers 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 seconds.

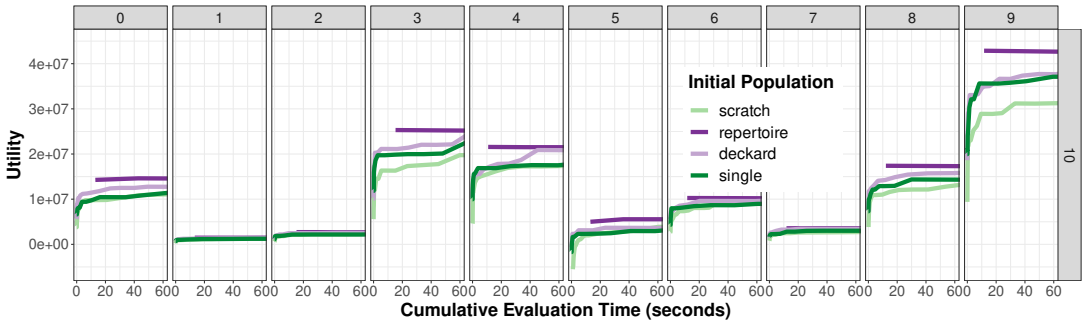


Figure 6.14: Results comparing planning from scratch, the repertoire, replanning from a single plan only, and replanning using Deckard. Deckard resulted in better utility for the first 13 seconds of planning, and is then overtaken by the repertoire.

### 6.3.2 Reusable Repertoires

The previous section evaluated the core approach and reuse enablers, demonstrating that the approach could result in fewer generations of planning, as well as an improvement in wall clock time for both the Omnet and DART case studies. This was achieved by reusing a single adaptation strategy only. In this section, we evaluate the first of the extensions to the core approach presented in Chapter 4, and show that a basic repertoire of adaptation strategies generated using a chaos engineering approach results in further improved planning in response to an unexpected change. Because Chapter 4 presents techniques with large complex plans in mind, this content will be evaluated on the Omnet case study where the plans have this property.

**Experimental Setup** We performed replanning on the simulated Omnet system for 30 randomly generated unexpected change scenarios. We report the utility obtained for replanning based on using (1) the generated repertoire of adaptation tactics using the chaos engineering approach described in Section 4.1, (2) a single plan (as in prior work [29]), and (3) from scratch (no reuse). We generated the unexpected change scenarios by creating 10 scenarios for each of 3 different different settings for the $m$ number of mutations parameter, 1, 5, and 10. This permits exploring how the size of the change influences replanning effectiveness for the approaches.

The repertoire comprises 200 adaptation strategies that we generated for 200 change scenarios. We generated the change scenarios by applying 1–5 random mutations to the baseline scenario (with the number of mutations selected uniformly at random). When replanning using a single adaptation strategy only, we selected the starting adaptation strategy for reuse randomly from the set of 200 adaptation

strategies. When replanning from scratch, the population is initialized completely randomly. To generate the starting population from the repertoire, 10% of the population is selected randomly from the repertoire, and the remaining 90% is generated from scratch; these values were taken from the prior work [29].

For all approaches, the genetic program was configured to plan for 30 generations using a population size of 1000. Planning was automatically terminated at 2000 seconds.

**Results** Figure 6.14 shows the results for the first 60 seconds of planning. For space constraints, only trials with 10 mutations (the highest and most challenging setting) are shown; the results for 1 and 5 were similar. The vertical axis is the utility obtained by the planner, and the horizontal axis is the planning time in seconds. The graph therefore shows the utility that the system would obtain by executing the best available plan produced by that planning approach at that time. Results from replanning using a single plan are labeled single.

For almost all randomly generated scenarios, the repertoire approach results in the highest planning utility. Sometimes the improvement compared to the next best planner was small, especially for single-mutation cases. For other scenarios the improvement was quite large (such as trial 9 in Figure 6.14. On aggregate, using the repertoire resulted in an average improvement of 11% to utility compared to reusing a single plan only. Planning using a single plan tends to only outperform planning from scratch, often slightly, reinforcing previous results [29]. One drawback to the repertoire approach is that it takes more time to produce the first plan (often taking around 15 seconds), although the plan that is obtained is often high quality. This is intuitive since effective plans are often large and expensive to evaluate, and the repertoire approach must evaluate many of these large and expensive plans. If planning in a domain where waiting 15 seconds is unacceptable, then reusing a single plan is better. Otherwise, the repertoire results in the highest expected utility.

### 6.3.3 Clone Detection

Next, we evaluate the clone detection approach for extracting reusable planning components from a repertoire presented in Section 4.2.1 We performed replanning on the same randomly generated unexpected change scenarios as in Section 6.3.2 using a clone detection approach to initialize the population. This approach is described in detail in Section 4.2.1. To do this, we ran Deckard on the repertoire of 200 adaptation strategies generated in the previous subsection to obtain a list of clones. Clones were selected from this list using tournament selection, selecting seven clusters randomly

| Rule | Trials Improved (%) | Overall % Change | 1 Mutation % Change | 5 Mutation % Change | 10 Mutation % Change |
|---|---|---|---|---|---|
| `seq-take-first` | 40.0 | -0.36 | 0.57 | -0.93 | -0.53 |
| `seq-take-second` | 26.7 | -2.13 | -2.36 | -0.89 | -3.01 |
| `try-take-first` | 96.7 | 3.51 | 3.79 | 3.27 | 3.53 |
| `try-take-second` | 36.7 | -0.75 | -2.36 | -0.89 | -1.56 |
| `try-take-third` | 63.3 | 0.46 | 0.89 | -0.06 | 0.59 |
| `for-decr` | 40.0 | -0.43 | 0.93 | -0.26 | -1.52 |
| `for-prune` | 63.3 | 0.56 | 0.58 | 0.71 | 0.42 |
| `try-unnest` | 60.0 | 0.26 | 0.51 | -0.13 | 0.40 |

Table 6.9: Improvement in maximum utility obtained by syntactic transforms over using the repertoire without transforms. `try-take-first` performed the best with a consistent 3.5% improvement.

from the list and returning a random clone from the largest cluster. The initial population was initialized with these clones. The result of this strategy is shown in Figure 6.14, labeled as deckard.

Overall, the clone detection approach results in an improvement compared to planning from scratch and replanning with a single plan only, but the maximum utility was obtained by reusing the repertoire rather than the extracted clones. Nevertheless, the clone detection approach yields plans more quickly. Given enough planning time, the repertoire approach eventually finds a better plan than the clone detection approach, but when a small amount of time is available, the clone detection approach is better. The breakeven point, where both clone detection and the repertoire are best for an equal number of the trials, occurs after 13 seconds of planning. For the first 10 seconds of planning, the clone detection approach yields the highest utility for 24 out of the 30 trials, with reusing a single plan being the best for 4 trials and planning from scratch the best for the remaining 2 trials. When planning for longer than 13 seconds the repertoire approach is expected to result in the highest utility.

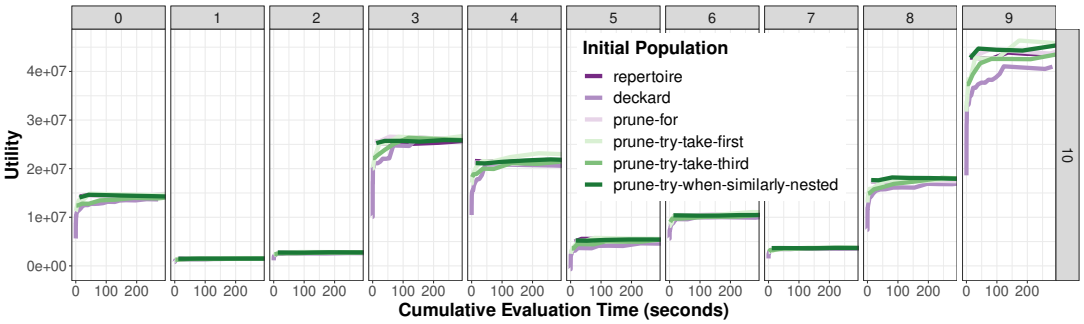


Figure 6.15: Utility versus planning time for the four beneficial syntactic transforms. Some transforms obtained results as quick as Deckard but with better utility. `try-take-first` is the overall best after around 2 minutes of planning.

### 6.3.4 Rule-based Syntactic Transforms

The third research approach for generating effective repertoires from Chapter 4 tailors reusable plan fragments with syntactic transformations. For this experiment we generated adaptation strategies for the same 30 unexpected change scenarios as before, while applying eight syntactic transformations (as described in in Section 4.2.2). For each syntactic transformation, we applied the transformation to the starting repertoire of 200 adaptation strategies from prior experiments, and then used the transformed repertoire to seed the initial population for replanning. The large number of trials makes the results of this experiment difficult to show visually, so results are shown in Table 6.9.

Of the eight transforms evaluated, four result in an improvement over the baseline (the repertoire with no transforms) more than half of the time. The `try-take-first` performed the best, improving on the baseline for 29/30 trials, and with an average improvement to expected utility of 3.51%. This improvement is consistent across each of the three numbers of mutations in the experiment. The `for-prune` transformation also results in an improvement for all three numbers of mutations, but with a lower percentage of trials improved (63.3%) and a lower overall improvement to utility (0.56%). The other two transforms that showed an overall improvement were `try-take-third` and `try-unnest`. These transforms both improved about 60% of trials for 0.46% and 0.26% average improvement respectively. The other transforms resulted in an overall decrease to expected utility.

The biggest takeaway from these results is the performance of the `try-take-first` transform, which improves utility for all but one trial for 3.51% on average. Compared to replanning with a single plan only, this transform resulted in an overall

average improvement of 15%, with the best overall improvement being 20% for trial 9. It is interesting that `try-take-first` performed much better than other `try-*` templates, since all these transforms similarly prune subtrees of Try-catch operators. The difference is that the `try-take-first` picks the first subtree of the Try-body and removes other subtrees, while remaining templates remove other parts of the Try-catch subtrees. This result makes sense intuitively since the transform captures what is likely the most important information contained in the Try-catch operator (the subplan that is attempted first) while reducing evaluation time on evaluating the contingencies. We've previously noted a common planning motif where an important subplan is tried multiple times to ensure that it is carried out, should it fail a few times. Crucially, when this occurs, capturing the important subplan with a tailored rule allows the planner to reuse the information learned during subsequent replanning while reducing the evaluation time.

Figure 6.15 shows the results of the syntactic transforms versus clone detection (deckard), and using the repertoire without modifications (repertoire). For presentation, we show only the four transforms that result in a positive average improvement. Compared to planning from scratch or reusing a single plan (shown as single), replanning using only the repertoire results in the highest utility, but typically requires more time to begin returning results (cf. Section 6.3.2). Overall, Table 6.9 shows that the `try-take-first` improves on the repertoire by 3.5%. Figure 6.15 shows that syntactic transforms generate plans as quickly as the Deckard-based planner, but with consistently higher utility. Interestingly, the `try-unnest` and `for-prune` transforms take about as long to start returning plans as the repertoire alone, around 15 seconds, but result in higher utility than `try-take-first` for the first minute or two of planning. If a plan is needed within a short time window, such as 10 seconds, `try-take-first` is the best approach. For an intermediate window between around 15 seconds to 60 seconds, `try-when-similarly-nested` or `for-prune` perform the best. When planning longer than 60 seconds is permissible, `try-take-first` is again most effective.

### 6.3.5 Adversarial Settings

The third claim in Section 6.1 is that: "Plan reuse is applicable to a range of unexpected change scenarios, including adversarial settings." In this section, we evaluate whether plan reuse can be effectively applied to adversarial settings using the coevolutionary approach described in Section 5.

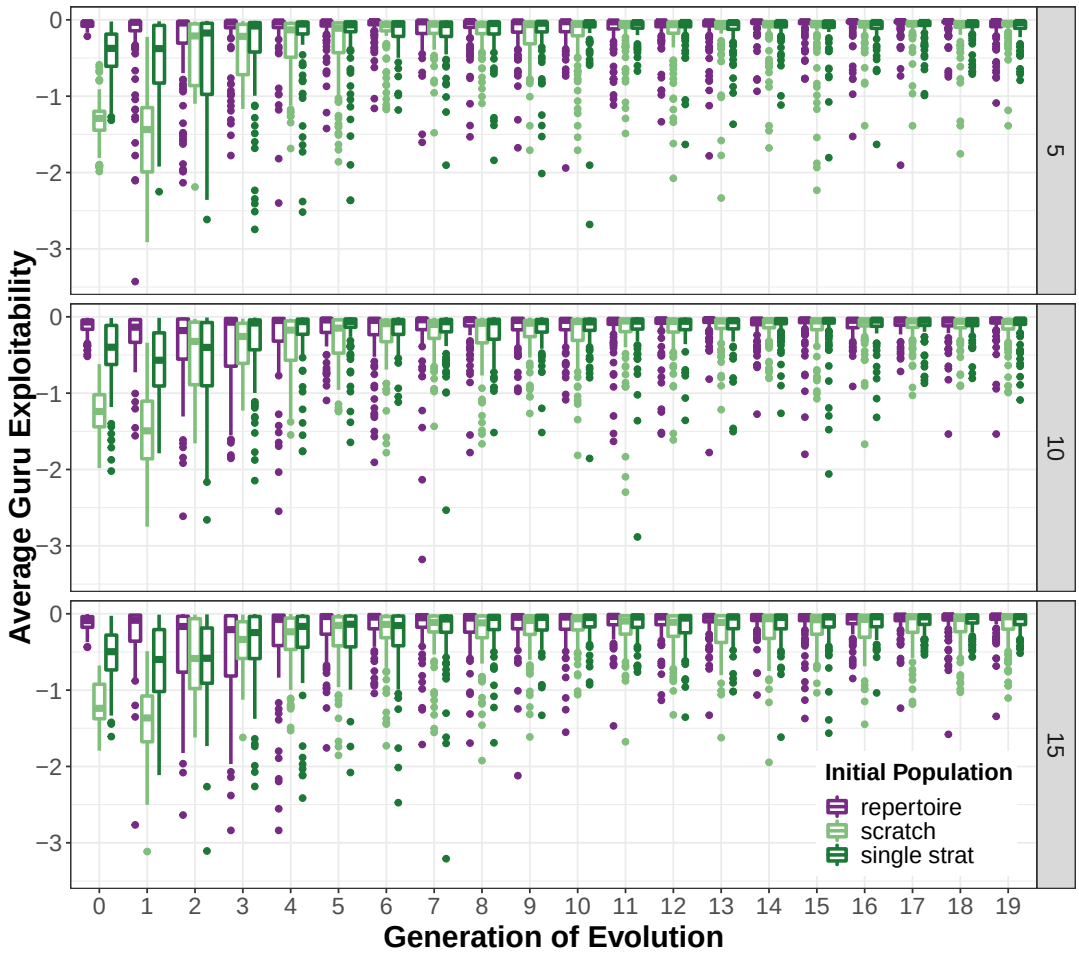


Figure 6.16: The average exploitability of each generation's guru individuals for each of the three studied reuse approaches for the Bullseye case study, broken down by the number of mutations used to generate the change scenarios. The reusable repertoire results in the best outcome for the defender, with a particular advantage during the first few generations of planning.

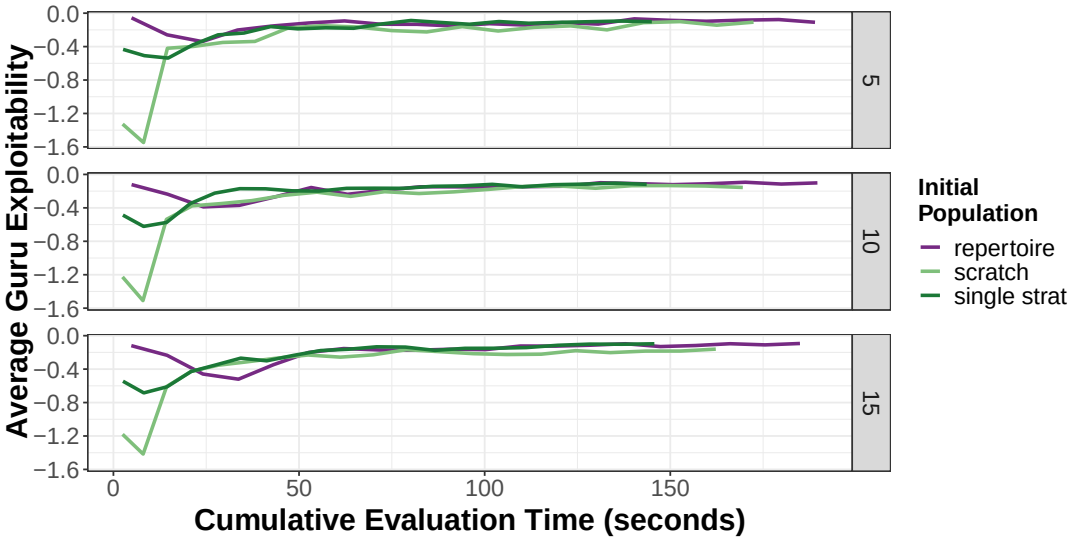


Figure 6.17: The average exploitability of guru individuals presented against planning time instead of generation, for each of the reuse approaches broken down by number of mutations. The reusable repertoire remains the best for the early phase of planning, but the approaches converge after around fifty seconds.

**Experimental Setup** This experiment uses the Bullseye exemplar system defined in Section 6.2.2, and the co-evolutionary extension to the GP planner explained in Chapter 5. Experiments for this research question were run on an Ubuntu 18.04.5 LTS server with OpenJDK version 11.0.9.1, Python 2.7.17, an Intel Xeon Gold 6240 CPU with 72 cores running at 2.60GHz, and 263 GB of RAM. Due to an implementation particularity with the ECJ library, each planning job was evaluated using a single thread at a time. Memory was restricted to 5 GB. There are a few key differences in available operators compared to the cloud service provider exemplar. For loops and try-catch operators have been removed. The try-catch operator is not needed since tactics are assumed to succeed as long as the preconditions are met. For loops are unnecessary since Bullseye takes place over a small number of timesteps for evaluation reasons. Instead, new randomization and conditional operators are provided. Conditionals are provided for the attacker to branch their actions based on the assets that they have compromised. Alternatively, since the defender does not have the ability to know what the attacker has compromised, the defender has no need for these conditionals. When mutating scenarios for Bullseye, an attribute is selected uniformly at random and modified with up to plus or minus 0.1 Gaussian noise, and then rounded to the nearest legal value if the allowed bounds are exceeded

(i.e., probabilities over 1 or under 0).

To evaluate this research question, we compare replanning entirely from scratch with no reuse, reusing a single strategy for the attacker and defender, and reusing a repertoire for both attacker and defender. We performed the experiment for a total of 30 different change scenarios, 10 each of three different numbers of mutations from a starting scenario. The number of mutations was 5, 10, and 15. Planning for each scenario was repeated 10 times and aggregate results are reported. For this experiment planning was performed for 20 generations with a population size of 250 for both the attacker and defender. The reusable repertoire was generated by saving the highest fitness individual from each generation of planning for 100 randomly generated scenarios, generated by applying five mutations to the starting scenario. During planing time, the starting populations are initialized by randomly selecting strategies from these repertoires. When replanning using a single strategy only, strategies generated by the Gambit solver for the starting scenario were reused.

While we evaluate the effectiveness of planning by comparing fitness elsewhere, the unique properties of security and co-evolution make direct fitness comparison between approaches less useful. This is because fitness depends not only on the quality of the best defender strategy found during the search, but also on the strategy used by the attacker. This means that during the course of evolution, the defender's fitness could decrease as the search discovers improved strategies for the attacker, even though the search could be making global progress by obtaining a defensive strategy that is robust to all of the attacker's best options that have been discovered. Likewise, rising local fitness does not necessarily mean that the quality of the system's plan is improving globally, since it is possible that the search is becoming trapped in a local optima of over-optimizing the defender's response to a sub-optimal attacker strategy, leaving the defender with a strategy that will result in poor fitness against a better attacker strategy that has not yet been discovered. Ideally for comparison we would have a means of measuring the true global utility of the defender's strategy. Thus in the results we report the exploitability of the defender's strategy rather than the fitness.

The exploitability is the difference between the defender's fitness obtained if the attacker plays the best response to the defender's chosen strategy minus the fitness the defender obtains playing a Nash equilibrium. The exploitability is a global way of objectively measuring the progress of search. The best possible exploitability is zero, which means that the search converged to a Nash equilibrium. A lower value indicates the utility that the defender loses if the attacker plays the best response compared to the defender's utility if they had played a Nash equilibrium strategy. To compute the exploitability, we use the Gambit [46] solver to find the attacker's

best response and the fitness obtained at Nash equilibrium. Note that this analysis relies on the property that Bullseye is a two-player zero-sum game, where all Nash equilibria are guaranteed to have the same utility [55]. We limit Bullseye to two timesteps because this is the highest number of timesteps that Gambit can reliably solve in the time budget of 1 minute.

**Results** Figure 6.16 shows a box and whisker plot of the average exploitability of the 5 guru (fittest) defender individuals for each generation of evolution, separated by the number of mutations used to generate the change scenarios. The results are similar to the cloud web server exemplar, with using the chaos generated repertoire resulting in the best exploitability for all three mutation sizes, especially during the first five generations. Reusing a single adaptation strategy performed better than planning from scratch, but worse than the repertoire. Over all scenarios, reusing the repertoire resulted in a 92% improvement in exploitability compared to planning from scratch after the first generation of planning, while reusing a single strategy for each agent resulted in a 61% improvement. After twenty generations of planning, the spread between the exploitablilites for all approaches tighten, but the repertoire remains the best with a 29% improvement from scratch compared to a 25% improvement when reusing single plans only.

Figure 6.17 shows the average guru exploitability by planning time rather than generations of evolution. These results also show that early in planning reusing the repertoire results in a big improvement compared to planning from scratch, and reusing a single adaptation strategy results in a smaller improvement. With longer planning however, the results are less consistent as the three approaches converge to similar values.

## 6.4 Summary

This chapter presented the evaluation of the thesis, supplying evidence for the three claims given in Section 6.1. Section 6.3.1 evaluated the core approach, finding that plan reuse resulted in fewer generations of planning before reaching a high quality plan for both the Omnet and DART case study systems (claim 1). Additionally, improvements in wall-clock time were found in both case studies (claim 2), including a case where an order of magnitude improvement was obtained in the DART case. Sections 6.3.2, 6.3.3, and 6.3.4 evaluated the results of the extensions described in Chapter 4 for reusing a repertoire of adaptation strategies, resulting in improved performance with respect to the metrics set out in claims 1 and 2. Finally, Section 6.3.5

evaluated the co-evolutionary extension described in Chapter 5, finding that the presented approach to reuse is also applicable in adversarial situations, in addition to the wide range of other change scenarios studied throughout the evaluation (claim 3).

# Chapter 7

# Discussion and Conclusion

This chapter provides a discussion of the lessons learned about plan reuse with stochastic search through the course of the thesis. First, a discussion of when our approach to reuse is likely to result in a large improvement (as opposed to a minimal improvement or even a slight reduction in effectiveness) is presented. Next, the key limitations of the thesis are addressed. Lastly, a discussion of promising avenues for future research is presented.

## 7.1 When is reuse applicable?

The results of the evaluation in Chapter 6.3 showed that plan reuse can often result in an improvement in planning effectiveness, both in terms of number of generations until convergence to a good plan, as well as wall-clock time. However, an improvement was not always observed, and the degree to which reuse resulted in improvements varied over the change scenarios and case studies in the evaluation. This section will discuss lessons learned about when an improvement can be expected.

### 7.1.1 When the change is small

Intuitively, the effectiveness of reuse after an unexpected change is linked to the size of the change. On one extreme, if the change is maximally trivial and has no impact on the system, simply reusing a pre-existing strategy will result in a good outcome. On the other extreme, if the change is so great that it renders all of the prior knowledge encoded in the existing plans inapplicable, then any time spent attempting to reuse them will be wasted compared to simply replanning entirely from scratch. Several

experiments in the evaluation of the thesis investigated altering the degree of the unexpected change.

At a high level, there are three types of unexpected changes that the approach can address with respect to a given repertoire of initial strategies generated from a given scenario generator. The first, and easiest, are those change scenarios that the scenario generator explicitly generated during the offline initialization phase. This would imply that there already exists a strategy in the repertoire that was generated precisely to handle the change. When this occurs, the planner simply needs to identify the correct strategy, which is done by evaluating the fitness of all plans in the starting population. Planning quality in this case depends solely on the amount of evaluation overhead that it takes to complete this process. If this overhead is less than the expected convergence time to a good strategy from scratch, then an improvement over planning from scratch is trivially obtained. The user also has a great deal of control over of the planning overhead, which is a function of the number of individuals in the population, and the time that it takes to evaluate them. The number of individuals is a parameter that can be adjusted, and this thesis presents several approaches for reducing the evaluation time of each individual. Convergence time from scratch may also be measured empirically for a given space of unexpected changes by simply running the planner on a scenario generator that can sample from the desired change space. The evaluation overhead of a given starting population can also be measured similarly.

A more complex type of change is when the specific unexpected change scenario was not explicitly generated during the offline initialization, but when the unexpected change comes from the same space of changes from which the scenario generator sampled. This implies that although the exact scenario was not anticipated, that possibility that such a scenario could occur was build into the scenario generator. In the evaluation of the thesis, this type of change corresponds to when replanning was evaluated for the same number of mutation operators that were used during the initialization phase. In all case studies where we studied repertoire construction, we were able to show at least a small improvement in planning from scratch for these types of changes. Predicting improvement in practice can be done by replanning using the same scenario generator as used during the initialization phase.

The third, and most interesting level of change, are those changes that the scenario generator could not have generated during the offline initialization phase. When improvement is possible in this case depends both the on evaluation overhead, as well as the degree of distance between the change scenario, and those scenarios considered during the offline phase. Given a novel scenario generator, it is possible to experimentally determine the degree to which improvement can be expected by per-

forming replanning using the scenario generator and observing the resulting average plan utility versus planning time curve and comparing this to the curve obtained by planning from scratch for the same generated scenarios. Precisely quantifying the space of changes for which a given repertoire will result in an improvement is a challenging problem that is not addressed in this thesis and left to future work, however in the course of the evaluation we observed strong results in planning effectiveness even when the number of mutations occurring at runtime were two times as large as those that were generated during the offline phase, providing some indication that this space is quite large.

### 7.1.2 When planning time is more constrained

One common pattern across the validation was that plan reuse resulted in greater improvement compared to planning from scratch early in the planning process. This suggests that plan reuse is more useful when planning time is more limited. Intuitively this makes sense, since plan reuse allows the planner to start planning with some existing knowledge. Given sufficient planning time, (assuming that the planner does not become trapped in a local optima) eventually the planner will arrive at the optimal plan regardless of the starting knowledge. On the other extreme, if there is no time for replanning, reusing an existing plan will likely be preferable to having no plan or using a randomly generated plan. Plan reuse then, is more likely to result in an improvement when planning time is more limited.

The threshold for when reuse is likely to be useful can be estimated in practice relatively easily by generating adaptation strategies from scratch and determining the expected convergence time to reach a good plan. If the planning overhead for reuse is greater than this time, then it would be better to simply plan from scratch. If the overhead is lower than the expected convergence time, then improvement is possible. Reuse should especially be considered when waiting for the expected convergence time is unacceptable, but care should be taken to ensure that the evaluation overhead is within acceptable levels (which can be estimated).

### 7.1.3 When (re)obtaining the initial strategies is more expensive

Another key insight regarding when reuse is useful was observed in the DART case study evaluation of the core approach. The results in Section 6.3.1 revealed a case where plan reuse resulted in a order of magnitude improvement compared to replanning from scratch. Intuitively, this improvement is due to a *coarse* region of

the search space: that is, a region where mutation does not result in a measurable change in fitness.

Ideally for stochastic search, the search space should be smooth. A small change in a candidate solution should result in a small change in the fitness. This allows the search to find small improvements which incrementally improve the quality of the candidate solutions until convergence to an acceptable solution. In a coarse search space however, a small change to the candidate solution might not result in a measurable change in fitness. This means that the search has no means of determining which changes are making progress in the search, since the changes are indistinguishable.

In the DART case study, a large portion of the search space exhibits this coarse property, resulting in a situation where obtaining an adaptation strategy to escape this coarse region of the search space is very costly, and a starting plan that contains this knowledge results in a large improvement (see Section 6.3.1).

Here, we attempt to quantify the improvement that should be expected from plan reuse versus planning from scratch in this case. 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, the search will fail to make progress and will not find a high quality solution.

In the DART case study, a team of drones 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.[1]

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 $\delta$ is the number of net levels the team must descend in order 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 $\delta$ if $\delta$ 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 $\delta$ 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:

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

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

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

[1]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.

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 $p = 10000$.

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 $p(s) > 1 - 10^{\frac{\log \tau}{P}}$.

This analysis demonstrates a property of the search space which makes re-obtaining the starting adaptation strategy (descending 16 levels) very expensive. When this is the case, having the starting knowledge results in a large improvement.

While we observed this type of search space in the DART case study, we can quantify the coarse region of the space analytically, and we can expect a similar degree of improvement in other systems that also have a search space with similar properties. This could be the case for systems that need to perform a very complex, specific sequence of steps to change states, such as spacecraft transitioning into or out of safe mode for example, or aircraft performing takeoff or landing maneuvers.

## 7.2 Limitations

While the work presented in the thesis strives to enable self-* systems to more effectively adapt following unexpected changes, there remain limitations. First, this section will address a major assumption made in the thesis: that at an up-to-date model of the system and its environment is available to the system after an unexpected change occurs. Next, threats to the generalizability of the results will be discussed. Finally, we will discuss the question of deciding when to stop planning.

### 7.2.1 The model update problem

A simplifying assumption made in the thesis is that an up-to-date model of the system and the environment is provided to the system after an unexpected change occurs. This is necessary to for the GP planner to predict the utility the system would obtain by executing candidate plans. For example, if a new adaptation tactic is made available to the system, and the planner is called upon to generate a new adaptation strategy without having its model of the system updated, it would be impossible for the planner to take advantage of the new tactic, since the planner would not know how using it would affect its utility (or even that such a tactic exists). The issues of how the self-* system detects that replanning is necessary and how it obtains an up to date model is considered out of scope, and not addressed in this thesis.

A self-* system using the proposed approaches in a production setting would need a model update mechanism. One approach is that a human operator could supply the model update. This is in line with current practice in self-* systems where human operators are required to update the adaptation strategies after an unexpected change occurs. Updating the models for an automated planning approach could likely be easier than updating an entire adaptation strategy, although in many cases the benefits provided by reuse may be small compared to the overhead of the time spent updating the models. An alternative approach is that the models could be updated automatically, a problem that is the subject of ongoing research [74, 27]. While further from implementation in practice, these approaches are more promising for the increasingly large and complex self-adaptive systems of the future.

While we envision systems that can learn and adapt in response to arbitrary changes, a more realistic path for more adaptable self-* systems could be model update approaches that can handle certain classes of unexpected change, rather than attempting to manage the entire change space. Addressing changes to known parameters that can be automatically measured for example, is easier and more plausible than autonomously detecting the existence of new adaptation tactics. Additional discussion of future research directions is provided in Section 7.3.

### 7.2.2 Threats to external validity

Another limitation of the thesis is threats to the validity of the empirical evaluation. We attempted to mitigate these threats by performing the validation on a collection of three case study systems from different domains, and with different planning assumptions. However, the case study systems are all simulated exemplars, and a threat remains that the results of the validation may not generalize beyond these

exemplars to other self-* systems. However, the consistency of some results across the case studies (such as the improvements obtained by reuse early in the planning process) provide some indication that the principles discovered in the thesis may generalize. Additionally, although only observed in one case study, the order of magnitude improvement in the DART case study can be explained analytically (see Section 7.1.3) and can be expected to be similarly effective in other self-* systems whose search spaces have similar properties.

### 7.2.3 When to stop planning

Another important question for self-* systems utilizing anytime planning approaches such as stochastic search is deciding when to stop planning and using the best plan available. This question is also outside of the scope of this thesis, although we did observe (see Section 3.4) that modifying the parameters of the search enables the user to make rich tradeoffs in planning time and optimality. Hybrid planning [57] has been proposed as a means for integrating slow and precise planners with fast and approximate planners, and similar principles apply to deciding when to stop planning with stochastic search. In particular, machine learning approaches were investigated in the hybrid planning work for predicting when a given planning approach should be used. A similar idea could be applied to stochastic search where a machine learning component could predict the optimal amount of time to invest in planning.

## 7.3 Future Work

There are many promising research directions involving plan reuse and stochastic search to improve the ability of self-* systems to respond to unexpected changes.

### 7.3.1 Reuse with Neuro-controllers

While this thesis explored approaches for reusing previous strategies using stochastic search, autonomous planners are increasingly using other planning paradigms such as neural network based controllers to decide on adaptation [66, 5]. Artificial Neural Networks (ANNs) are well-suited for this task since they can allow self-* systems to learn highly complex behaviors efficiently, and have been shown to outperform humans in difficult tasks like chess [70] and poker [8].

While these approaches are powerful, they are typically treated as black boxes, and can stop working in the face of unexpected situations that they were not trained

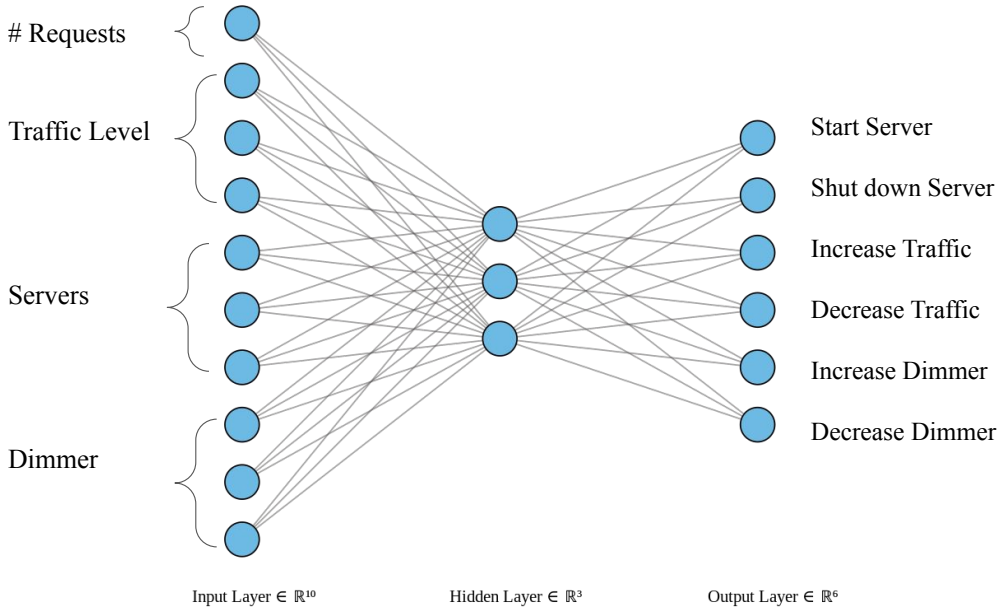


Figure 7.1: Initial ANN for neuro-evolutionary search to evolve.

to handle. Exploring reuse in the context of these neuro-controllers could allow self-* systems to benefit from the power of ANNs while mitigating their weakness of fragility to change.

Some preliminary work explored reusing neuro-controllers during a neuro-evolutionary search. Figure 7.1 shows a basic neuro-controller for the Omnet case study, and Figure 7.2 shows some preliminary results comparing synthesizing a new neuro-controller from scratch versus reusing an existing neuro-controller. The presented results are in line with the other experiments in the thesis, finding that reuse outperforms planning from scratch early in the search, and shows that exploring plan reuse in other planning paradigms such as neuro-controlled systems shows promise.

### 7.3.2 Reusing explanations

Another area increasingly explored by the self-* community is the human in the loop systems. Self-* systems often work in collaboration with humans, and as a result achieve better outcomes when the system communicates more effectively with the human, especially if the system's decisions are difficult for the human to understand and trust. In the domain of plan reuse, systems with humans in the loop could benefit from explanations that communicate why the previously successful plan was

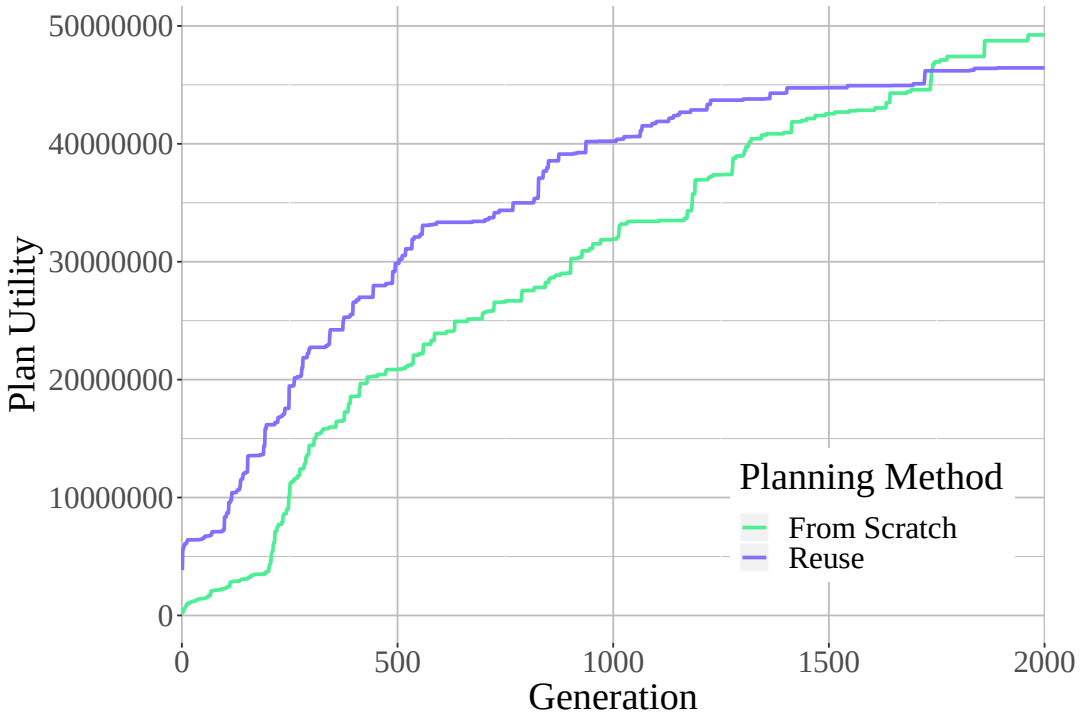


Figure 7.2: The utility of neuro-reuse and planning from scratch plotted during training with a new randomized trace at every generation. Average is taken over 30 trials.

changed in the way that it was. Another possible avenue of work at the intersection of explainability and reuse is reusing existing explanations and applying them to new plans. This could be approached by tracking generational changes in adaptation strategies and providing pointers to existing explanation artefacts in evolved strategies.

### 7.3.3 Integration with self-* infrastructure

An implementation of plan reuse with stochastic search in the Rainbow [15] self-adaptation infrastructure is another direction for future work. While the thesis is evaluated on three case study systems selected to represent a diverse set of self-* systems and planning assumptions, these case studies are simulations that abstract away some implementation details of the surrounding self-* infrastructure. An implementation in Rainbow could facilitate broader investigation in plan reuse by others and might reveal insights on applying plan reuse in a more realistic context.

There are several ways that an integration with the Rainbow infrastructure could be accomplished. One approach to integration with Rainbow is to generate new adaptation strategies using existing adaptation strategies in a Rainbow deployment. This type of integration would require developing an individual representation for a genetic programming planner for the Stitch planning language utilized by Rainbow. Crucially, this would include the applicability guards used by Rainbow to determine when the system is outside of desired conditions and adaptation is necessary. The GP planner would need a specification of the available effectors and probes, and models of the system and its environment to provide a means of estimating the fitness of candidate Stitch adaptation strategies. Apart from having an up-to-date model, this approach to integration also presupposes a means for the system to detect that the current adaptation strategies are no longer sufficient and that replanning is necessary. This could be done for example by a machine learning analysis (i.e., anomaly detection), or by noticing a discrepancy in expected versus obtained utility values.

### 7.3.4 A more rigorous treatment of the unknown

A key concept underlying the thesis is that we seek to enable systems to adapt more effectively in response to unexpected changes. However, what is meant exactly by "unexpected" has been only loosely defined as "that which was not anticipated at design time". While the term has been used broadly in the thesis, and in spite of efforts to evaluate the thesis on a wide range of change scenarios, the kinds of

unexpected changes that have been evaluated have been limited and not precisely quantified. However, we did include unexpected changes from a number of locations in the MAPE-K architecture where changes could occur, including changes to the available adaptation tactics, the effects of tactics, the environment, and the system's utility function. Self-* systems face an infinite number of possible unexpected changes, and yet some kinds of unexpected changes seem more unexpected than others. We should not fault our self-* systems for failing to plan for what would happen if the speed of light should change, even though such a change is conceivable (however unlikely given our current understating of physics). While a number of taxonomies of uncertainty have been proposed [3, 64, 43, 60], these taxonomies are of limited help in understanding the space of possible changes in a way that enables self-* systems to increase their resilience in the most effective way. A possible future avenue of work is developing a framework for understanding the space of unexpected changes and facilitating decision making in determining the space of changes that a system should build robustness towards [14].

## 7.4 Conclusion

Research in self-* systems has enabled these systems to successfully adapt in response to changes that they were designed to handle at design time, however these systems often struggle when confronted with unexpected changes that were not considered at design time. This thesis presented a collection of approaches using stochastic search to reuse existing adaptation knowledge to enable self-* systems to adapt more effectively in response to unexpected changes.

### 7.4.1 Contributions

The contributions of the thesis, first presented in Section 1.3, are restated here.

1. An approach for plan reuse with stochastic search for more effective replanning following unexpected changes.
   (a) A planner using genetic programming and initial population seeding to support reusing existing adaptation strategies.
   (b) A collection of reuse enabling approaches to reduce the evaluation time of existing strategies to facilitate effective plan reuse.

   Publications:

   - Cody Kinneer, Zack Coker, Jiacheng Wang, David Garlan, and Claire Le

Goues. 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*, pages 40–50. ACM, 2018

- Cody Kinneer, David Garlan, and Claire Le Goues. Information reuse and stochastic search: Managing uncertainty in self-* systems. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 15(1):1–36, 2021
- Gabriel A Moreno, Cody Kinneer, Ashutosh Pandey, and David Garlan. Dartsim: An exemplar for evaluation and comparison of self-adaptation approaches for smart cyber-physical systems. In *Proceedings of the 14th International Conference on Software Engineering for Adaptive and Self-Managing Systems*, pages 137–143. ACM/IEEE, 2019

Artifacts:

The source code for the GP planner is publicly available for extension and replication at the following GitHub repository: `https://github.com/squaresLab/sass`. The DARTSim exemplar is available at `https://github.com/cps-sei/dartsim`. Data and analysis code for this thrust is available at: `https://github.com/squaresLab/seams2018-data`, and also at `https://github.com/squaresLab/taas-2018-data`.

2. Techniques for generating reusable repertoires of adaptation strategies to broaden the types of unexpected changes that self-* systems can replan for effectively.
   (a) An approach inspired by chaos engineering for obtaining planning knowledge for a range of change scenarios.
   (b) Analysis approaches for extracting reusable planning components including clone detection and syntactic transformations.

   Publications:

   - Cody Kinneer, Rijnard Van Tonder, David Garlan, and Claire Le Goues. Building reusable repertoires for stochastic self-* planners. In *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, pages 222–231. IEEE, 2020

   Artifacts:

   The source code for the approaches described in this research thrust are available with the GP planner at `https://github.com/squaresLab/sass`. Data and analysis code is available at: `https://github.com/squaresLab/acsos2020-data`.

3. An adversarial extension to support plan reuse to promote the security quality attribute.
   (a) The Observable Eviction Game (OEG), a game theoretic model of system defense laying the foundation for self-* systems that can autonomously adapt in response to unexpected changes in the security landscape.
   (b) A co-evolutionary extension to support reusing adaptation strategies when planning for adversarial situations, enabling self-* systems to replan in the face of unexpected security threats.

   Publications:

   - Cody Kinneer, Ryan Wagner, Fei Fang, Claire Le Goues, and David Garlan. Modeling observability in adaptive systems to defend against advanced persistent threats. In *Proceedings of the 17th ACM-IEEE International Conference on Formal Methods and Models for System Design*. ACM-IEEE, 2019

   Artifacts:

   The source code for the co-evolutionary extension including the Bullseye exemplar is included with the GP planner code at `https://github.com/squaresLab/sass`. Source code for solving the Bullseye exemplar system with Gambit is available at `https://github.com/squaresLab/bullseye-gambit`. Source code for the Observable Eviction Game is available at `https://github.com/squaresLab/oeg-code`. Data and analysis code is available at `https://github.com/squaresLab/sass-coev-data`.

### 7.4.2 Summary

As systems become larger and more complex, the difficulty of planning for the unexpected will only increase. This thesis presented knowledge reuse with stochastic search as an approach for addressing unexpected changes, through three research thrusts: plan reuse with stochastic search and reuse enablers, approaches for constructing reusable repertoires of adaptation strategies for addressing a larger space of unexpected changes, and a co-evolutionary extension for replanning in adversarial environments. The thesis was evaluated on three case study systems from different domains and planning assumptions, with the results finding that in many cases plan reuse resulted in improved planning effectiveness following unexpected changes. In conclusion, plan reuse in self-* systems has the potential to enable the next generation of autonomous systems to quickly respond to unexpected changes.

# Bibliography


[1] Advanced persistent threat groups. `https://www.fireeye.com/current-threats/apt-groups.html`. Accessed: 2018-04.

[2] Comby. `https://comby.dev`, Online. Accessed 13 May 2020.

[3] Jesper Andersson, Rogerio De Lemos, Sam Malek, and Danny Weyns. Modeling dimensions of self-adaptive software systems. In *Software engineering for self-adaptive systems*, pages 27–47. Springer, 2009.

[4] Phillip G. Armour. The five orders of ignorance. *Commun. ACM*, 43(10): 17–20, October 2000. ISSN 0001-0782. doi: 10.1145/352183.352194. URL `http://doi.acm.org/10.1145/352183.352194`.

[5] Chloe Barnes, Aniko Ekart, Kai Olav Ellefsen, Kyrre Glette, Peter Lewis, and Jim Torresen. Coevolutionary learning of neuromodulated controllers for multi-stage and gamified tasks. 08 2020.

[6] Ali Basiri, Niosha Behnam, Ruud De Rooij, Lorin Hochstein, Luke Kosewski, Justin Reynolds, and Casey Rosenthal. Chaos engineering. *IEEE Software*, 33 (3):35–41, 2016.

[7] Ira D Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *Software Maintenance, 1998. Proceedings., International Conference on*, pages 368–377. IEEE, 1998.

[8] Noam Brown and Tuomas Sandholm. Superhuman AI for heads-up no-limit poker: Libratus beats top professionals. *Science*, 359(6374):418–424, 2018.

[9] Javier Cámara, Gabriel A Moreno, and David Garlan. Stochastic game analysis and latency awareness for proactive self-adaptation. In *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 155–164. ACM, 2014.

[10] Tao Chen, Ke Li, Rami Bahsoon, and Xin Yao. Femosaa: Feature-guided and

knee-driven multi-objective optimization for self-adaptive software. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 27(2):5, 2018.

[11] Shang-Wen Cheng and David Garlan. Stitch: A language for architecture-based self-adaptation. *J. Syst. Softw.*, 85(12):2860–2875, 2012. ISSN 0164-1212.

[12] Shang-Wen Cheng, David Garlan, and Bradley Schmerl. Evaluating the effectiveness of the rainbow self-adaptive system. In *2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, pages 132–141, 2009. doi: 10.1109/SEAMS.2009.5069082.

[13] Zack Coker, David Garlan, and Claire Le Goues. Sass: Self-adaptation using stochastic search. In *Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 168–174. IEEE Press, 2015.

[14] David Garlan. The unknown unknowns are not totally unknown. In *Proceedings of the 16th Symposium on Software Engineering for Adaptive and Self-Managing Systems*, 2021. To Appear.

[15] David Garlan, S-W Cheng, A-C Huang, Bradley Schmerl, and Peter Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54, 2004.

[16] Simos Gerasimou, Radu Calinescu, and Giordano Tamburrelli. Synthesis of probabilistic models for quality-of-service software engineering. *Automated Software Engineering*, 25(4):785–831, 2018.

[17] Alicia Grech and Julie Main. *Case-Base Injection Schemes to Case Adaptation Using Genetic Algorithms*, pages 198–210. ECCBR. Berlin, Heidelberg, 2004. ISBN 978-3-540-28631-8.

[18] Mark Harman, Yue Jia, William B. Langdon, Justyna Petke, Iman Hemati Moghadam, Shin Yoo, and Fan Wu. Genetic improvement for adaptive software engineering (keynote). In *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS 2014, pages 1–4, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2864-7. doi: 10.1145/2593929.2600116. URL `http://doi.acm.org/10.1145/2593929.2600116`.

[19] John C Harsanyi and Reinhard Selten. A generalized Nash solution for two-person bargaining games with incomplete information. *Management Science*, 18(5-part-2), 1972.

[20] Scott A. Hissam, Sagar Chaki, and Gabriel A. Moreno. High assurance for dis-

tributed cyber physical systems. In *Proceedings of the 2015 European Conference on Software Architecture Workshops*, ECSAW '15, pages 6:1–6:4, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3393-1. doi: 10.1145/2797433.2797439. URL `http://doi.acm.org/10.1145/2797433.2797439`.

[21] John H Holland. Genetic algorithms. *Scientific american*, 267(1):66–73, 1992.

[22] Yury Izrailevsky and Ariel Tseitlin. The netflix simian army. `https://netflixtechblog.com/the-netflix-simian-army-16e57fbab116`. Accessed: 2020-3-23.

[23] Sushil Jajodia, Anup K Ghosh, Vipin Swarup, Cliff Wang, and X Sean Wang. *Moving target defense: creating asymmetric uncertainty for cyber threats*, volume 54. Springer Science & Business Media, 2011.

[24] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th international conference on Software Engineering*, pages 96–105. IEEE Computer Society, 2007.

[25] Elmar Juergens, Florian Deissenboeck, and Benjamin Hummel. Clonedetective-a workbench for clone detection research. In *Proceedings of the 31st International Conference on Software Engineering*, pages 603–606. IEEE Computer Society, 2009.

[26] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003. ISSN 0018-9162.

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

[28] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic Patch Generation Learned from Human-written Patches. In *International Conference on Software Engineering*, ICSE '13, pages 802–811, 2013.

[29] Cody Kinneer, Zack Coker, Jiacheng Wang, David Garlan, and Claire Le Goues. 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*, pages 40–50. ACM, 2018.

[30] Cody Kinneer, Ryan Wagner, Fei Fang, Claire Le Goues, and David Garlan. Modeling observability in adaptive systems to defend against advanced persistent threats. In *Proceedings of the 17th ACM-IEEE International Conference on Formal Methods and Models for System Design*. ACM-IEEE, 2019.

[31] Cody Kinneer, Rijnard Van Tonder, David Garlan, and Claire Le Goues. Building reusable repertoires for stochastic self-* planners. In *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, pages 222–231. IEEE, 2020.

[32] Cody Kinneer, David Garlan, and Claire Le Goues. Information reuse and stochastic search: Managing uncertainty in self-* systems. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 15(1):1–36, 2021.

[33] Richard Kissel. *Glossary of key information security terms*. Diane Publishing, 2011.

[34] Cristian Klein, Martina Maggio, Karl-Erik AArzén, and Francisco Hernández-Rodriguez. Brownout: Building more robust cloud applications. In *Int. Conf. on Soft. Eng.*, ICSE '14, pages 700–711, 2014. ISBN 978-1-4503-2756-5.

[35] Kostas A Kontogiannis, Renator DeMori, Ettore Merlo, Michael Galler, and Morris Bernstein. Pattern matching for clone and concept detection. *Automated Software Engineering*, 3(1):77–108, 1996.

[36] John R Koza. Genetic evolution and co-evolution of game strategies. In *International Conference on Game Theory and Its Applications*. Citeseer, 1992.

[37] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992. ISBN 0-262-11170-5.

[38] Brian Krebs. Email attack on vendor set up breach at target. `https://krebsonsecurity.com/2014/02/email-attack-on-vendor-set-up-breach-at-target/comment-page-2/`, 2014. Accessed: 2019-11-06.

[39] Jens Krinke. Identifying similar code with program dependence graphs. In *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on*, pages 301–309. IEEE, 2001.

[40] Christian Kroer, Gabriele Farina, and Tuomas Sandholm. Robust stackelberg equilibria in extensive-form games and extension to limited lookahead, 2017.

[41] Christian Krupitzer, Felix Maximilian Roth, Sebastian VanSyckel, Gregor Schiele, and Christian Becker. A survey on engineering approaches for self-adaptive systems. *Pervasive and Mobile Computing*, 17:184–206, 2015.

[42] Marta Kwiatkowska, Gethin Norman, and David Parker. Prism 4.0: Verification of probabilistic real-time systems. In *International conference on computer aided verification*, pages 585–591. Springer, 2011.

[43] Jean-Claude Laprie. From dependability to resilience. In *38th IEEE/IFIP Int.*

*Conf. On dependable systems and networks*, pages G8–G9, 2008.

[44] Sushil J Louis and John McDonnell. Learning with case-injected genetic algorithms. *IEEE Transactions on Evolutionary Computation*, 8(4):316–328, 2004.

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

[46] Richard D. McKelvey, Andrew M. McLennan, and Theodore L. Turocy. Gambit: Software tools for game theory, version 16.0.1. `http://www.gambit-project.org`. Accessed: 2018-02.

[47] Gabriel A Moreno, Javier Cámara, David Garlan, and Bradley Schmerl. Proactive self-adaptation under uncertainty: a probabilistic model checking approach. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 1–12. ACM, 2015.

[48] Gabriel A Moreno, Javier Cámara, David Garlan, and Bradley Schmerl. Efficient decision-making under uncertainty for proactive self-adaptation. In *2016 IEEE International Conference on Autonomic Computing (ICAC)*, pages 147–156. IEEE, 2016.

[49] Gabriel A. Moreno, Javier Cámara, David Garlan, and Bradley Schmerl. Flexible and efficient decision-making for proactive latency-aware self-adaptation. *ACM Trans. Auton. Adapt. Syst.*, 13(1):3:1–3:36, April 2018. ISSN 1556-4665. doi: 10.1145/3149180. URL `http://doi.acm.org/10.1145/3149180`.

[50] Gabriel A Moreno, Bradley Schmerl, and David Garlan. Swim: an exemplar for evaluation and comparison of self-adaptation approaches for web applications. In *Proceedings of the 13th International Conference on Software Engineering for Adaptive and Self-Managing Systems*, pages 137–143. ACM, 2018.

[51] Gabriel A Moreno, Cody Kinneer, Ashutosh Pandey, and David Garlan. Dartsim: An exemplar for evaluation and comparison of self-adaptation approaches for smart cyber-physical systems. In *Proceedings of the 14th International Conference on Software Engineering for Adaptive and Self-Managing Systems*, pages 137–143. ACM/IEEE, 2019.

[52] Héctor Munoz-Avila and Michael T Cox. Case-based plan adaptation: An analysis and review. *IEEE Intelligent Systems*, 23(4):75–81, 2008.

[53] John Nash. Non-cooperative games. *Annals of mathematics*, pages 286–295, 1951.

[54] Bernhard Nebel and Jana Koehler. Plan reuse versus plan generation: A theo-

retical and empirical analysis. *Artificial Intelligence*, 76(1-2):427–454, 1995.

[55] Noam Nisan, Tim Roughgarden, Eva Tardos, and Vijay V Vazirani. *Algorithmic game theory*. Cambridge university press, 2007.

[56] Stefano Nolfi and Dario Floreano. Coevolving predator and prey robots: Do "arms races" arise in artificial evolution? *Artificial life*, 4(4):311–335, 1998.

[57] Ashutosh Pandey, Gabriel A Moreno, Javier Cámara, and David Garlan. Hybrid planning for decision making in self-adaptive systems. In *2016 IEEE 10th International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*, pages 130–139. IEEE, 2016.

[58] Praveen Paruchuri, Jonathan P. Pearce, Janusz Marecki, Milind Tambe, Fernando Ordonez, and Sarit Kraus. Playing games for security: An efficient exact algorithm for solving bayesian stackelberg games. In *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems - Volume 2*, AAMAS '08, Richland, SC, 2008. International Foundation for Autonomous Agents and Multiagent Systems. ISBN 978-0-9817381-1-6.

[59] Mateusz Pawlik and Nikolaus Augsten. Efficient computation of the tree edit distance. *ACM Trans. Database Syst.*, 40(1):3:1–3:40, 2015. ISSN 0362-5915. doi: 10.1145/2699485.

[60] Diego Perez-Palacin and Raffaela Mirandola. Uncertainties in the modeling of self-adaptive systems: a taxonomy and an example of availability evaluation. In *Proceedings of the 5th ACM/SPEC international conference on Performance engineering*, pages 3–14. ACM, 2014.

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

[62] Andres J. Ramirez, David B. Knoester, Betty H.C. Cheng, and Philip K. McKinley. Applying genetic algorithms to decision making in autonomic computing systems. In *Proceedings of the 6th International Conference on Autonomic Computing*, ICAC '09, pages 97–106, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-564-2. doi: 10.1145/1555228.1555258. URL `http://doi.acm.org/10.1145/1555228.1555258`.

[63] Andres J. Ramirez, Betty H.C. Cheng, Philip K. McKinley, and Benjamin E. Beckmann. Automatically generating adaptive logic to balance non-functional tradeoffs during reconfiguration. In *Proceedings of the 7th International Conference on Autonomic Computing*, ICAC '10, pages 225–234, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0074-2. doi: 10.1145/1809049.1809080. URL `http://doi.acm.org/10.1145/1809049.1809080`.

[64] Andres J Ramirez, Adam C Jensen, and Betty HC Cheng. A taxonomy of uncertainty for dynamically adaptive systems. In *Proceedings of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 99–108. IEEE Press, 2012.

[65] Chanchal Kumar Roy and James R. Cordy. A survey on software clone detection research. *School of Computing TR 2007-541, Queen's University*, 115, 2007.

[66] Shouvik Roy, Usama Mehmood, Radu Grosu, Scott A. Smolka, Scott D. Stoller, and Ashish Tiwari. Learning distributed controllers for v-formation, 2020.

[67] Bradley Schmerl, Javier Cámara, Gabriel A. Moreno, David Garlan, and Andrew Mellinger. Architecture-based self-adaptation for moving target defense. Technical Report CMU-ISR-14-109, Institute for Software Research, Carnegie Mellon University, 2014.

[68] Bruce Schneier. Attack trees. *Dr. Dobb's journal*, 24(12), 1999.

[69] Abdullah Sheneamer and Jugal Kalita. A survey of software clone detection techniques. *International Journal of Computer Applications*, 137(10):1–21, 2016.

[70] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

[71] Tyron Stading, Petros Maniatis, and Mary Baker. Peer-to-peer caching schemes to address flash crowds. In *Int. Workshop on Peer-to-Peer Systems*, IPTPS '02, pages 203–213, 2002. ISBN 3-540-44179-4.

[72] Leonardo Trujillo. Genetic programming with one-point crossover and subtree mutation for effective problem solving and bloat control. *Soft. Computing*, 15 (8):1551–1567, 2011. ISSN 1433-7479.

[73] Jakub Černý, Branislav Boýanský, and Christopher Kiekintveld. Incremental strategy generation for stackelberg equilibria in extensive-form games. In *Proceedings of the 2018 ACM Conference on Economics and Computation*, EC '18, pages 151–168, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5829-3. doi: 10.1145/3219166.3219219. URL `http://doi.acm.org/10.1145/3219166.3219219`.

[74] Thomas Vogel and Holger Giese. Adaptation and abstract runtime models. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '10, pages 39–48, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-971-8. doi: 10.1145/1808984.1808989. URL

http://doi.acm.org/10.1145/1808984.1808989.

[75] Ryan Wagner, Matthew Fredrikson, and David Garlan. An advanced persistent threat exemplar. Technical report, Technical Report CMU-ISR-17-100, Institute of Software Research, Carnegie Mellon University, 2017.

[76] Ji Zhang and Betty HC Cheng. Model-based development of dynamically adaptive software. In *Proceedings of the 28th international conference on Software engineering*, pages 371–380. ACM, 2006.