Ph.D. Dissertation Proposal

Search-based Plan Reuse in Self-* Systems

Cody Kinneer
Institute for Software Research
Carnegie Mellon University
Pittsburgh, PA
kinneerc@cs.cmu.edu

April, 2020

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

Thesis Committee

Claire Le Goues (Co-chair)
Carnegie Mellon University

David Garlan (Co-chair)
Carnegie Mellon University

Fei Fang
Carnegie Mellon University

Betty Cheng
Michigan State University

Contents

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 [9, 29, 51].

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 runtime [41], and offline planners [7] prepare a collection or repertoire of adaptation strategies beforehand, which are then selected from at runtime. 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 [12]. 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 [8] 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 [6, 44, 45], 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 [39], and while genetic algorithms have been employed to reuse information for certain limited problem cases [11, 32], 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.

I propose to use 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 unknown unknowns, and (c) reusing abstract strategies in adversarial settings.*

Plans will be 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 will investigate 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 unknown unknowns. This thrust will use ideas from program analysis, namely 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 will explore how coevolution can be applied to reuse plans by refining abstract strategies in this environment.

The thesis will be 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 the Target data breach [50, 27]. Evolution ability will be measured by the number of generations until convergence to a good plan, the improvement in wall-clock time needed for replanning, and the range of unexpected change scenarios that the approach can address.

The expected contributions of the thesis are the following:

1. A planner using genetic programming to reuse existing adaptation strategies for effective replanning in response to unexpected changes.

2. A collection of reuse enabling approaches to reduce the evaluation time of existing strategies to facilitate effective plan reuse.

3. Techniques for generating reusable repertoires of adaptation strategies to broaden the types of unexpected changes that self-* systems can replan for effectively.

4. 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.

5. A coevolutional extension to support reusing adaptation strategies when planning for adversarial situations, enabling self-* systems to replan in the face of unexpected security threats.

6. An empirical evaluation of the approach on three representative case study systems.

7. A discussion of how the technical contributions of the thesis can be implemented in the Rainbow [9] self-* infrastructure.

The remainder of the proposal is organized as follows: Section 2 provides relevant background, including self-* systems, genetic algorithms, and game theory. Section 3 proposes a solution approach for reusing plans using stochastic search. Section 4.1 proposes reuse enabling approaches for reducing the cost of plan reuse. Section 4.2 proposes an approach for identifying generalizable plan fragments to build reusable plan repertoires. Section 4.3 proposes an approach for applying plan reuse in domains including adversarial interactions. Section 5 describes the proposed validation, including a description of the case study systems and claims. Section 6 provides a timeline for the completion of the proposed thesis. Lastly, Section 7 concludes.

2 Review of Literature and Background

This section provides the relevant background for the proposed 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 proposed approach, 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.

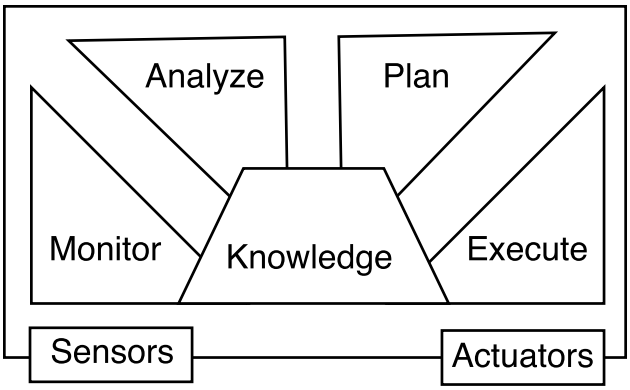


Figure 1: MAPE-K Loop for self-* systems.

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 [18], depicted in Figure 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 proposal focuses on the planning component, which generates an adaptation 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 [41], while offline planners [7] 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" [46]. Under-

standing uncertainty has been a focus of research in self-* systems, including modeling and taxonomizing uncertainty [2, 46, 31, 42], or managing it [33, 5, 34]. Often, uncertainty is categorized by its "anticipation" [2], "prospect" [31], or "level" [42], indicating the epistemic degree of uncertainty. One common demarcation of uncertainty is according to foreseen, foreseeable, and unforeseen types of changes [2, 46, 31], where foreseen changes are those aspects that were considered at design time and 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 according to orders of ignorance [3, 42], 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 [14]. 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 [26, 43]. 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 [25]. GP and how it is used in my thesis approach is explained in more detail in Section 3.

There are many approaches for planning, including those that apply evolutionary approaches. Plato and Hermes [45] 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 distinc-

tion in the proposed work 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 [10] 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. The proposed 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, my proposed research will also address identifying reusable planning components and adversarial environments.

Case-based plan adaption [38] explicitly reuses past plans in new contexts, in which context GAs have been explored directly [11, 32], 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 proposed 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.

An alternative to heuristic planners are exhaustive planners. These planners evaluate every possible plan, which allows them to always find the optimal plan. Models checkers such as PRISM [30] 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.

2.3 Clone-detection

Section 4.2 proposes clone-detection as an approach for building reusable repertoires of adaptation strategies. Clone-detection is analyzing software for duplicate source code [47, 24, 17], 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 [4], or program dependence graphs [28]. Deckard [16] 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 describes proposed 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. Section 4.3 proposes 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" [22].

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) [15].

Game Theory. Self-adaptive systems tasked with promoting security face the challenge of needing to reason about the behavior of adversaries. Game theory provides a framework for reasoning mathematically about interactions between multiple agents, or players [40], and is utilized to reason about the adversarial interactions surrounding security in Section 4.3. At a high level, a game is defined by a set of players, actions available to each player, and utility functions for each player that maps outcomes to utility values. Games provide a way to analyze situations where multiple agents (players) are able to take actions to influence the outcome, and where each agent has their own individual utility function. This is the case when designing self-* systems that consider security as a quality attribute, since security assumes the presence of an attacker, another agent besides the self-* system itself that can take actions and itself adapt in response to the system, and whose interests are opposed to those of the system. One contribution of the proposed thesis is the Observable Eviction Game (OEG) [21]: a game theoretic model for analyzing the interactions between an adaptive system and different types of advanced persistent threat (APT) attackers. This model is referenced in the proposed work described in Section 4.3.

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. I propose 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 proposed 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 the proposed work on plan reuse. The planner is described in terms of a case study system, a cloud-based web server, that will serve as a running example.

The proposed 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. 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 proposed planner to replan. The mechanism for synchronizing the system model with the actual world is outside the scope of the proposed thesis; this may be done manually (likely with less effort than replanning), or automatically [49, 19]. 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 4.1.

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).

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: return best_ind(p)
```

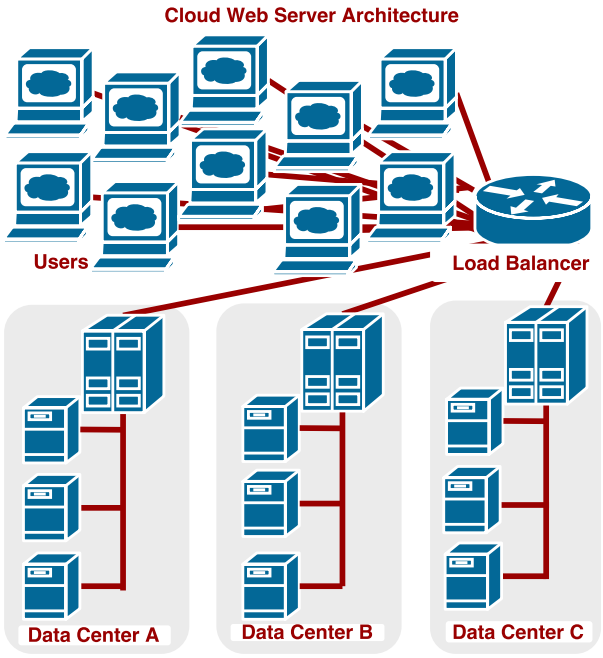


Figure 2: Cloud web server architecture.

3.1 Cloud Web Server

The thesis will be evaluated on three case study systems. The first case study system will serve as a running example for the remainder of the proposal, the remaining case studies are described in Section 5.1. The first case study system is a cloud-based web server with an N-tiered architecture, depicted in Figure 2[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 deliv-

[1]This case study system has been used to evaluate self-* planners in related work [41], work published as a contribution of the proposed thesis [20], and as a standalone exemplar [36]

ered by a brownout mechanism (requests serviced with the 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 [23] 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.

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

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

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

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

New tactics may become available, via new data centers, server types, or even hardware. The use case or environment may also unexpectedly change.

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. Figure 3 gives a Backus-Naur grammar for the plans. Each plan consists of either (a) one of six available *tactics* (described in Section 3.1), 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 4 uses a `try-catch` operator, first attempting to start a new server at data center A. If successful, it attempts to start a server at data center B; if not, it retries the `StartServer A` tactic.

This planning language is a simplified variant of other languages such as Stitch [7]. 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 [30] 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 that take too much time to evaluate). The crossover operator [48] selects a subtree in each of two parent plans (selected via tournament selection [26]) 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 4 shows a plan and its corresponding state tree. Leaf nodes are labeled with their state fitness (profit, in this example); edges with their probability. Left transitions correspond to tactic failure; right-transitions, tactic success. Following the

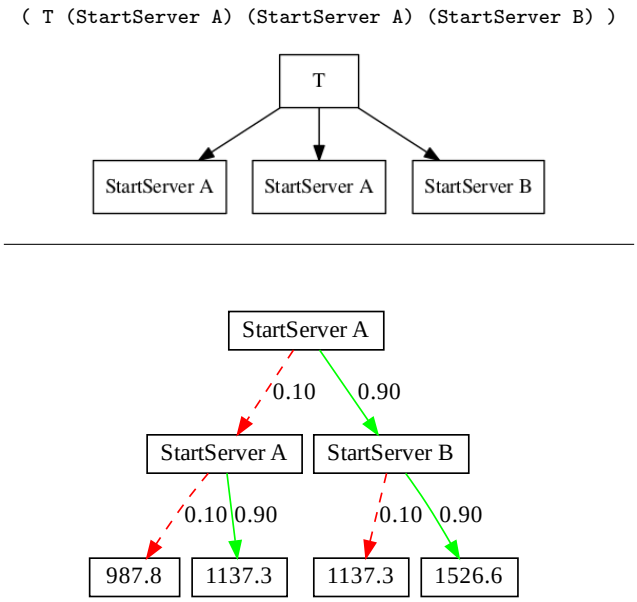


Figure 4: Top: An example plan. Bottom: This plan's system state tree. Dashed red arrows denote tactic failure, while solid green denotes success.

right-hand transitions shows that, if all tactics succeed, profit will be 1526.6, with an 81% probability. Following the left hand transitions shows the expected system state if all tactics fail (1% probability). The weighted sum over all paths (overall fitness) is 1451.14.

The simulator takes into account planning time and tactic latency [35]. 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 penalty thresholds. Figure 5 shows results from this parameter sweep performed on the cloud-based web server case

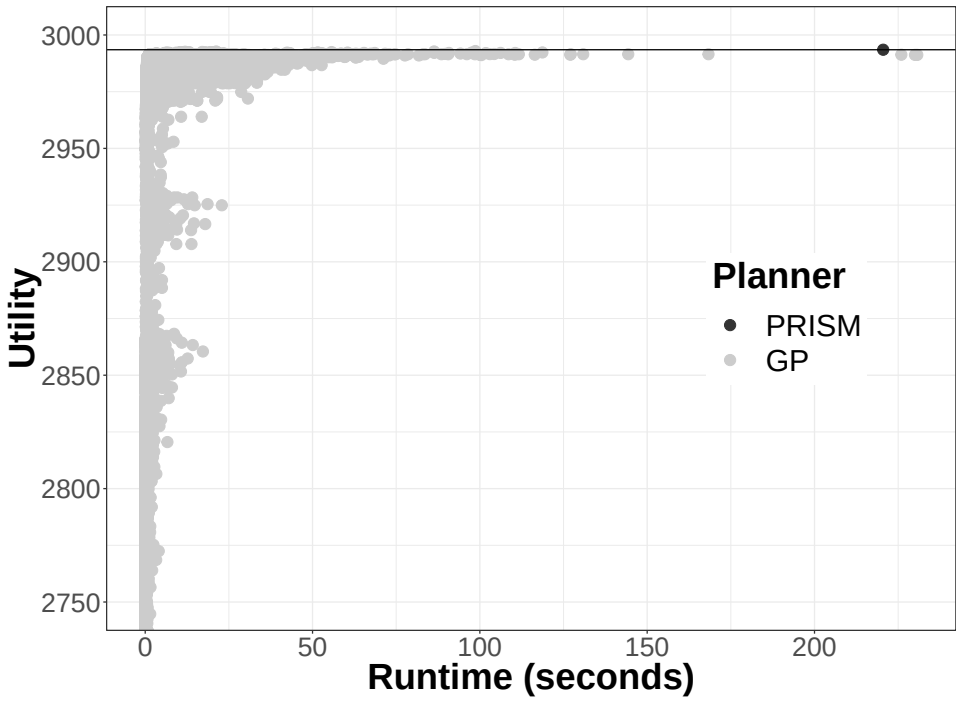


Figure 5: Utility versus planning time for GP parameter configurations. Many configurations produce similar utility results to PRISM, significantly faster.

study. The dark point at the top of Figure 5 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 proposed 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 4 proposes extensions to this planner to address these key challenges.

4 Proposed Work

This section describes the proposed work, three research thrusts which extend on the approach for plan reuse described in Section 3. Section 4.1 proposes several additional heuristic modifications to fitness computation to manage invalid actions and plan size, making plan reuse more effective. Section 4.2 proposes applying program analysis techniques such as clone detection to identify reusable planning components and create reper-

Table 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.

toires of plans to reuse rather than reusing a single source plan. Section 4.3 proposes reusing plans in adversarial settings using co-evolution and reusing strategies obtained from game-theoretic models to allow the application of reuse to quality attributes such as security.

4.1 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 [20]. 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, I propose 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 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 [26] (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.

4.1.1 Preliminary Results

To demonstrate the promise of these reuse enabling approaches, preliminary results from planning for the `Request Spike + New Data Center` scenario with a planning window of 10000 are shown. The proposed reuse enabling approaches were incrementally enabled 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 to determine values that obtained a high utility and low planning time.

Table 2 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` improved utility to 1.044. Without any reuse-enabling techniques, reusing plans by initializing the population with mutated versions of the starting plan resulted in a 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 makes improving planning utility through reuse nontrivial, the proposed enhancements to GP planning can reduce this cost and achieve higher utility than planning from scratch.

4.2 Building reusable repertoires by identifying generalizable plan fragments

Section 4.1 proposed obtaining an improvement in planning utility by randomly trimming existing plans into smaller sub-plans, balancing reused plans with short randomly

Table 2: Improvement obtained by reuse enabling techniques.

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

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, 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, I propose 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 determine common code patterns, or code clones have been applied to source code, such as Deckard [16] or program dependence graphs [28]. 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 [20].

The proposed work includes converting plans from the existing plan language to a source code representation amenable to existing clone detection techniques (Deckard), using the output of this technique to extract common sub-plans, and a comparison of using these common sub-plans to seed the search during replanning versus planning from scratch or the previous replanning approach.

4.3 Plan reuse in an adversarial setting

Applying the insights from the initial work to the security domain presents several challenges. The adversarial nature of the problem requires reasoning about the behavior and adaptation of the attacker in addition to the system, including the attacker's goals and TTPs, as well as the importance of observability and information gathering to both sides. I will investigate applying co-evolution to the self-* planning problem to facilitate reasoning about the behavior of the attacker. Co-evolution is a evolutionary approach where multiple populations of individuals are evolved to solve a problem, and have been applied to evolving strategies in competitive games [25].

Another promising opportunity to leverage plan reuse to defend self-* systems is the refinement of precomputed strategies obtained from game theoretic models [40], which are useful for reasoning about adversarial interactions, but are often difficult to scale to real world problem sizes. For example, the Observable Eviction Game [21], a game theoretic model for deciding between waiting to gather information or attempting to evict an APT attacker, could be used to provide seed strategies for refinement in the search. This could allow the search to leverage the information encoded in the game model while mitigating scalability issues.

4.4 Stretch Goals

There are many promising research directions involving plan reuse and stochastic search to improve the ability of self-* systems to respond to unexpected changes. One important and under investigated subject in self-* systems is the explainability of plans. 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 changed in the way that it was.

Another stretch goal for the proposed thesis is applying insights from plan synthesis with reuse to program synthesis and repair. Preliminary work on plan reuse and stochastic search [8] drew inspiration from work on automated program repair where reusing existing code within programs was found to lead to more effective repair. Since adaptation plans and programs are fundamentally similar, i.e., both specifying instructions to be carried out, and both structured as abstract syntax trees, it is possible that insights produced from plan reuse may be transferable back to program repair.

The final stretch goal of the proposed thesis is an implementation of plan reuse with stochastic search in the Rainbow [9] self-adaptation infrastructure. While the proposed thesis will be 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.

5 Validation

The thesis will be evaluated based on the following three claims described in Section 5.2:

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 will be evaluated on three case study systems:

1. A cloud based web server.
2. The DART team of autonomous aerial vehicles.
3. An enterprise system under attack by advanced persistent threats.

Section 5.1 describes the case study systems in detail. Section 5.2 elaborates on the claims and how they are evaluated for the proposed work.

5.1 Case study systems

The thesis will be 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. The first case study systems is the cloud web server described in detail in Section 3.1, and includes an environment where tactics may fail, and planning is only performed 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 section describes the remaining case studies in detail.

5.1.1 DART

The second case study system, DART [37] is inspired by a scenario from the DART systems project [13], using the same modeling approach and parameters as related work [35]. 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.

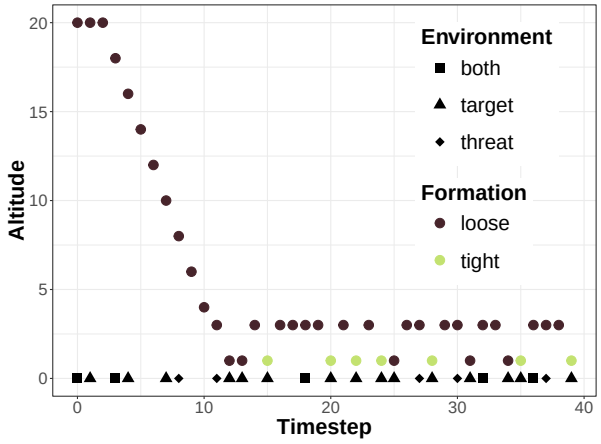


Figure 6: An example trace of the DART team moving through an environment.

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 also decrease the chance that a threat destroys the team, but at the cost of reducing the effectiveness of target detection. In this work, the team starts at a high altitude of 20, and must descend 16 levels before utility gain can occur. This situation could arise if the team was retasked from another mission, or must arrive at the mission area due to air traffic restrictions or to avoid other threats. More generally, this aspect of the case study is indicative of self-* systems that require a specific initialization before utility can be affected by adaptation. Figure 6 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 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 [37].

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.

5.1.2 Observable Eviction Game

The third case study used to evaluate the thesis will be an enterprise system under attack by an advanced persistent threat, inspired by the Target data breach [50] and the Observable Eviction Game [21]. The system consists of an enterprise network containing a web server, a payment server, and off-site point of sale (POS) devices.

Utility. Since this case study captures an adversarial interaction, the utilities of both the attacker and defender must be considered. The defender's utility is the determined by the amount of disruption to the system (by either the actions of the attacker or the defender's countermeasures), and by the utility of the attacker (the defender prefers to prevent the attacker from completing their objectives).

The attacker's utility depends on the attacker's objective, which may be either causing disruption to the system, monetary gain, or gathering intelligence.

Adaptation Tactics. In this scenario, both the system and the attacker may utilize adaption tactics to influence the system and the environment. The defender may re-image servers, change passwords, throttle connections, enable camouflage, or update the POS devices.

The attackers tactics are to conduct spear phishing attacks, gain access to systems by zero day exploits, gain access to the POS devices by a malicious update, install keyloggers, take down compromised machines, or exfiltrate data.

Change Scenarios. The change scenarios considered in this case study are a change in the attackers objectives, the addition of an exploit available to the attacker, and a change in the effectiveness of a defensive tactic.

5.2 Claims

The goal of the proposed 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, I propose to investigate the 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 proposed work along with the three case study systems.

5.2.1 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 heuristic solutions like the one discussed in this proposal 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 generated quickly, is much better than a higher quality plan that cannot be generated in time to avoid the obstacle [41]. Anytime planning approaches (such as GAs) provide flexibility in determining how much time to spend planing, although the issue of deciding when to stop planning is outside of the scope of the proposed thesis. 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 assessed on a case-study basis.

5.2.2 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. 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 proposed work is developing mitigations for this problem, including strategies for speeding up the evaluation time and by identifying the most promising plan fragments to reuse.

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

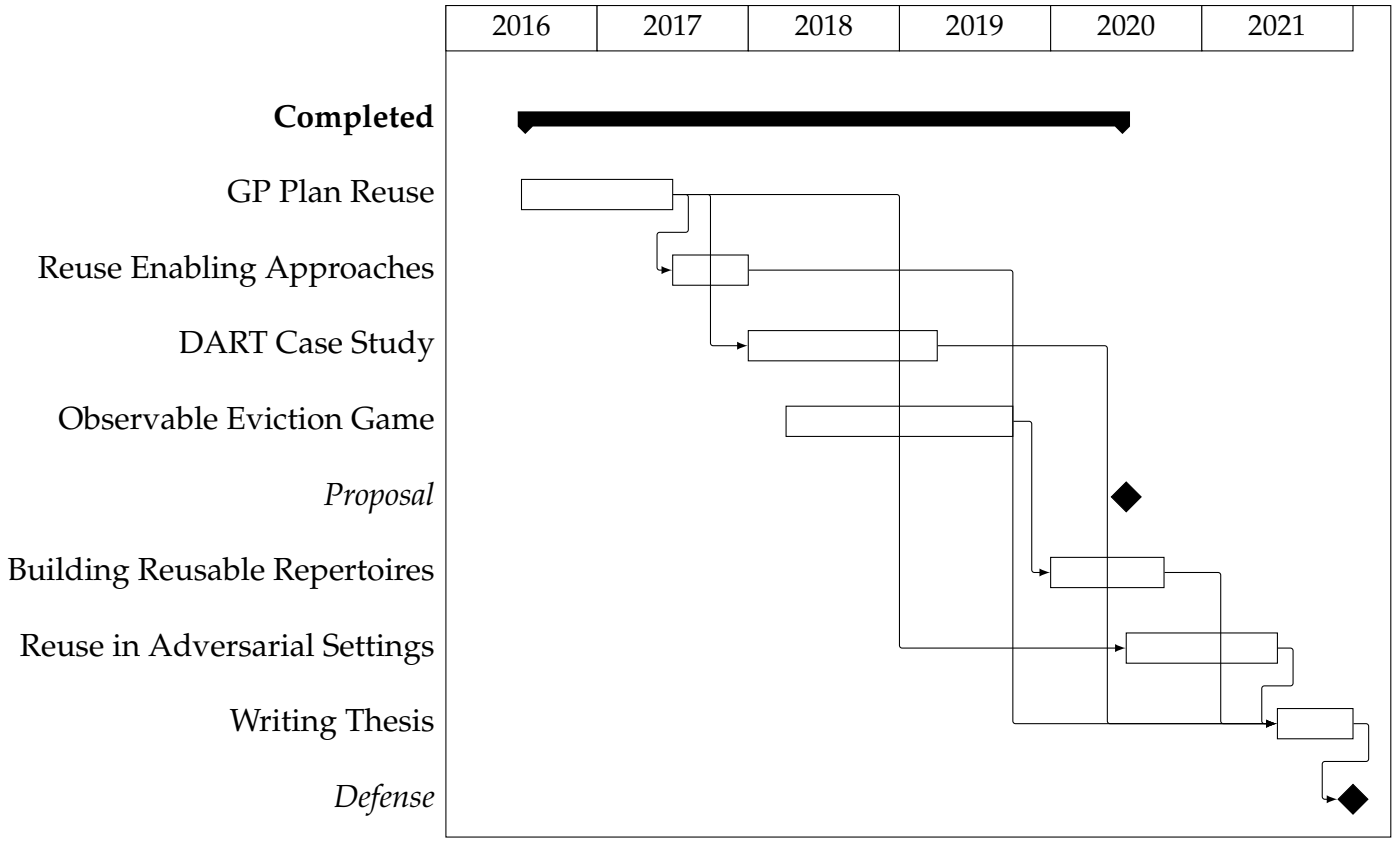
In addition to showing an ability to improve the effectiveness of planning, the approach should also generalize to a range of unexpected change scenarios, including adversarial settings. While it may not be possible to always obtain a significant improvement, the thesis will characterize the types of situations when improvement can be expected. This will be done by performing a survey of taxonomies on unexpected change scenarios, and

describing how the investigated scenarios relate to the space of unexpected changes, as well as describing when large improvements can be expected.

6 Proposed Timeline

This section provides a proposed timeline for the completion of the thesis, including a listing of what has already been accomplished, as well as a roadmap for the remaining work. The GP planner described in Section 3 has already been developed, and has been used to investigate approaches to reduce the evaluation time of candidate plans as described in Section 4.1. This work was evaluated using the cloud web server case study described in Section 3.1, and has been published at SEAMS [20]. An extension of this work applied to the DART case study described in Section 5.1.1 is currently under review.

The remaining two research thrusts described in Sections 4.2 and 4.3 remain to be completed. Work on the Observable Eviction Game, a game-theoretic model of APT defense has been published at MEMOCODE [21], and will provide a foundation for the third case study described in Section 5.1.2, which will in turn be used to evaluate the research thrust described in Section 4.3.



7 Conclusion

As systems become larger and more complex, the difficulty of planning for the unexpected will only increase. I propose to explore knowledge reuse using genetic programming planning as a promising tool for addressing unexpected changes. Future work in plan reuse in self-* systems has the potential to enable the next generation of autonomous systems to quickly respond to changes unforeseen at design time.

References

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

[2] 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.

[3] 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`.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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`.

[13] Scott A. Hissam, Sagar Chaki, and Gabriel A. Moreno. High assurance for distributed 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`.

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

[15] 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.

[16] 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.

[17] 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.

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

[19] 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.

[20] 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.

[21] Cody Kinneer, Ryan Wagner, Fei Fang, Claire Le Goues, and David Garlan. Modeling observability in adaptive systems to defend against advanced persistent threats. In *In proc. of 17th MEMOCODE*. ACM-IEEE, To Appear 2019.

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

[23] 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.

[24] K. A. Kontogiannis, R. Demori, E. Merlo, M. Galler, and M. Bernstein. Pattern matching for clone and concept detection. *Automated Software Engineering*, 3(1):77–108, Jun 1996. ISSN 1573-7535. doi: 10.1007/BF00126960. URL `https://doi.org/10.1007/BF00126960`.

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

[26] 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.

[27] 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.

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

[29] 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.

[30] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *Int. Conf. on Computer Aided Verification*, CAV '11, pages 585–591, 2011.

[31] Jean-Claude Laprie. From dependability to resilience. In *38th IEEE/IFIP Int. Conf. On dependable systems and networks*, pages G8–G9, 2008.

[32] S. J. Louis and J. McDonnell. Learning with case-injected genetic algorithms. *Trans. Evol. Comp*, 8(4):316–328, 2004. ISSN 1089-778X. doi: 10.1109/TEVC.2004.823466.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] H. Muñoz-Avila and M. T. Cox. Case-based plan adaptation: An analysis and review. *IEEE Intelligent Syst.*, 23(4):75–81, 2008. ISSN 1541-1672. doi: 10.1109/MIS.2008.59.

[39] Bernhard Nebel and Jana Koehler. Plan reuse versus plan generation: A theoretical and empirical analysis. *Artificial Intelligence*, 76(1-2):427–454, 1995.

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

[41] 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.

[42] 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.

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

[44] 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.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

[49] 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`.

[50] 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.

[51] 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.