Adaptation Timing in Self-Adaptive Systems

Gabriel A. Moreno

CMU-ISR-17-103
April 2017

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

Thesis Committee:
David Garlan (Chair)
Mark Klein
Claire Le Goues
Sam Malek (University of California, Irvine)

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

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

DM-0004422

Keywords: self-adaptive systems, latency-aware, proactive, probabilistic model checking, stochastic dynamic programming

To Nancy, Tommy and Melissa

Abstract

Software-intensive systems are increasingly expected to operate under changing and uncertain conditions, including not only varying user needs and workloads, but also fluctuating resource capacity. Self-adaptation is an approach that aims to address this problem, giving systems the ability to change their behavior and structure to adapt to changes in themselves and their operating environment without human intervention.

Self-adaptive systems tend to be reactive and myopic, adapting in response to changes without anticipating what the subsequent adaptation needs will be. Adapting reactively can result in inefficiencies due to the system performing a suboptimal sequence of adaptations. Furthermore, some adaptation tactics—atomic adaptation actions that leave the system in a consistent state—have latency and take some time to produce their effect. In that case, reactive adaptation causes the system to lag behind environment changes. What is worse, a long running adaptation action may prevent the system from performing other adaptations until it completes, further limiting its ability to effectively deal with the environment changes.

To address these limitations and improve the effectiveness of self-adaptation, we present *proactive latency-aware adaptation*, an approach that considers the timing of adaptation (i) leveraging predictions of the near future state of the environment to adapt proactively; (ii) considering the latency of adaptation tactics when deciding how to adapt; and (iii) executing tactics concurrently. We have developed three different solution approaches embodying these principles. One is based on probabilistic model checking, making it inherently able to deal with the stochastic behavior of the environment, and guaranteeing optimal adaptation choices over a finite decision horizon. The second approach uses stochastic dynamic programming to make adaptation decisions, and thanks to performing part of the computations required to make those decisions off-line, it achieves a speedup of an order of magnitude over the first solution approach without compromising optimality. A third solution approach makes adaptation decisions based on repertoires of adaptation strategies—predefined compositions of adaptation tactics. This approach is more scalable than the other two because the solution space is smaller, allowing an adaptive system to reap some of the benefits of proactive latency-aware adaptation even if the number of ways in which it could adapt is too large for the other approaches to consider all these possibilities.

We evaluate the approach using two different classes of systems with different adaptation goals, and different repertoires of adaptation strategies. One of them is a web system, with the adaptation goal of utility maximization. The other is a cyber-physical system operating in a hostile environment. In that system, self-adaptation must not only maximize the reward gained, but also keep the probability of surviving a mission above a threshold. In both cases, our results show that proactive latency-aware adaptation improves the effectiveness of self-adaptation with respect to reactive time-agnostic adaptation.

Acknowledgments

First and foremost, I would like to thank my advisor, David Garlan. I truly appreciate that he accepted to take me as a student even under the uncertainty that the newly created SEI Scholars Program posed at the time. His advice has been invaluable. I always enjoyed our meetings because, regardless of whether we discussed high-level concepts or the details of an algorithm, he always had an illuminating perspective that helped. At the same time, he would always ask the right questions, gently forcing me to understand better the issues at hand.

I also want to thank the members of my thesis committee, Mark Klein, Claire Le Goues, and Sam Malek. Their advice has helped me improve this dissertation.

I am grateful to the Software Engineering Institute for giving me the opportunity to pursue doctoral studies. In particular, I want to thank all the people who supported me, navigated all the administrative hurdles, and made this happen: Paul Nielsen, Bob Behler, Linda Northrop, Mark Klein, Kurt Wallnau, Doug Schmidt, Kevin Fall, Ned Deets, Jeff Boleng, and Sagar Chaki. Of course, this would have not been possible without the collaboration of the Institute for Software Research, mainly thanks to Bill Scherlis, Jonathan Aldrich, and Connie Herold.

I am also grateful for having had the opportunity to work with the members of the ABLE group. In particular, I would like to thank Javier Cámara and Bradley Schmerl, who worked with me and co-authored the publications that are the basis of this dissertation.

My brothers, Marcelo and Sergio, deserve special thanks for introducing me to the world of computers. First, they taught me to program in BASIC in a pocket computer that had a 1-line display. Soon after, when I was about 11 years old, they helped me nag my parents for months to buy a computer that was expensive and not even sold in Argentina at the time. We then continued exploring together this fascinating field.

Thanks to my parents, for their love and unconditional support, even after I chose to live in the other hemisphere, 5,000 miles away.

I want to thank my children, Tommy and Melissa, for their patience while I studied and worked on this, and for understanding when I was not available for having some family time. Finally, but most certainly not least, I want to thank my wife, Nancy. First, she left everything to come with me to Pittsburgh for my first graduate school experience at CMU. Not only that, but 11 years later, she was very supportive when I decided to go back to school for a PhD. For all the sacrifices she made, her love and friendship, I am forever grateful.

Contents

List of Figures

List of Tables

Acronyms

API Application Programming Interface.

CE Cross-Entropy.
CS Constraint Satisfaction.

DART Distributed Adaptive Real-Time.
DSL Domain-Specific Language.
DTMC Discrete Time Markov Chain.

ECM Electronic CounterMeasures.
EP-T Extended Pearson-Tukey.

FF Feed Forward.
FIFO First In, First Out.
FNR False Negative Rate.
FPR False Positive Rate.

LPS Limited Processor Sharing.

MAPE-K Monitor-Analyze-Plan-Execute-Knowledge.
MDP Markov Decision Process.
MPC Model Predictive Control.

OS Operating System.

P-NLA Proactive Non-Latency-Aware.
P-NLA-NC Proactive Non-Latency-Aware, No Concurrency.
PCTL Probabilistic Computation-Tree Logic.
PLA Proactive Latency-Aware.
PLA-NC Proactive Latency-Aware, No Concurrency.
PLA-PMC Proactive Latency-Aware with Probabilistic Model Checking.

PLA-SDP Proactive Latency-Aware with Stochastic Dynamic Programming.

PMC Probabilistic Model Checking.

POMDP Partially Observable Markov Decision Process.

RG Reward Gain.

RQV Run-time Quantitative Verification.

SB-PLA Strategy-Based Proactive Latency-Aware.

SDP Stochastic Dynamic Programming.

SLA Service Level Agreement.

TTP Tactics, Techniques, and Procedures.

UAV Unmanned Aerial Vehicle.

List of Terms

adaptation strategy A predefined decision tree built out of adaptation tactics [28].

adaptation tactic An action primitive that produces a change in the system, leaving it in a consistent state [28].

brownout A paradigm for self-adaptation that consists in enabling or disabling optional computations in the system in order to deal with changes in workload [83].

decision period The interval of time between consecutive adaptation decisions, denoted by τ.

dimmer In the brownout paradigm, a parameter that takes values in $[0, 1]$ controlling the proportion of system responses that include the optional computation [83].

discrete-time Markov chain Model for systems with fully probabilistic transitions, equivalent to a Markov decision process with only one possible action [92].

environment state State of the environment defined by the properties of the environment relevant to making adaptation decisions.

latency awareness The consideration of the latency of adaptation tactics when making adaptation decisions.

look-ahead horizon Period of time into the future over which the evolution of the system and the environment is considered when making adaptation decisions. Its length in decision intervals is denoted as H.

Markov decision process Model for sequential decision making under uncertainty [121], and suitable for modeling systems with a mix of probabilistic and nondeterministic behavior [92].

policy A function that prescribes the action that must be taken in each state of a Markov decision process in order to achieve some goal, such as maximizing the expected accumulated reward.

system configuration State of the system defined by the properties of the system that are relevant to making adaptation decisions.

system state See system configuration.

tactic effect The direct effect of the tactic on the structure and/or properties of the system, and not the indirect effect that they may be intended to produce. For example, when adding a new server, the effect is the system having one more active server, and not the reduction of the response time.

tactic latency The time it takes between when a tactic is started and when its effect is produced.

utility A measure of the goodness of the performance of the system with respect to its adaptation goal.

Chapter 1

Introduction

Software-intensive systems are increasingly expected to operate under changing conditions, including not only varying user needs and workloads, but also fluctuating resource capacity and degraded or failed parts [15, 26, 33, 37, 101]. Furthermore, considering the scale of systems today, the high availability demanded of them, and the fast pace at which conditions change, it is not viable to rely mainly on humans to reconfigure systems to maintain optimal performance [37, 42, 97]. Self-adaptation is an approach that aims to address this problem: a self-adaptive system is a system capable of changing its behavior and structure to adapt to changes in itself and its operating environment without human intervention [33].

Most self-adaptive system have some form of closed-loop control that monitors the state of the system and its environment, decides if and how the system should be changed, and performs the adaptation if necessary [15, 36, 81, 127]. Typically, these self-adaptation approaches rely on a set of adaptation tactics they can use to deal with different conditions [54, 68, 87, 129]. For example, adding a server is a tactic that can be used to accommodate increased load in the system, and revoking permissions from a user is a tactic for protecting the system from an insider attack. Furthermore, it is possible, and probably desirable, to have more than one suitable tactic to address the same issue. For instance, reducing the fidelity of the content served by a system (e.g., switching from multimedia to text) is another tactic to manage increased load. Therefore, the self-adaptive system must be able to not only decide when to adapt, but also choose among several applicable tactics.

These adaptation decisions are usually driven by some criteria such as maximizing utility [28, 90, 138, 141, 142], or maintaining a set of invariants [2, 16, 136]. In the first case, a utility function is used to evaluate the outcome of the candidate adaptations along different quality dimensions, such as response time, and operating cost. In the case of invariants, the goal of the adaptation is to satisfy a set of constraints. For example, it may need to keep the response time below some threshold, or ensure that the power consumption is kept below some level.

Current self-adaptive systems tend to be reactive and myopic [88]. Typically, they adapt in response to changes without anticipating what subsequent adaptation needs will be. Furthermore, when deciding how to adapt, they focus on the immediate outcome of the adaptation. In general, this would not be a problem if adaptation tactics were instantaneous, because the system could adapt swiftly to changes, and consequently, there would not be a need for preparing for upcoming environment changes. Unfortunately, adaptation tactics are not instantaneous. Although some

adaptation tactics are fast and can effectively be considered instantaneous, others are not. For example, adapting the system to produce results with less fidelity may be achieved quickly if it can be done by changing a simple setting in a component, whereas powering up an additional server to share system load may take a considerable amount of time. We refer to the time it takes from a tactic is started until its effect on the system is produced as *tactic latency*.[1]

Adapting reactively can result in inefficiencies due to the system performing a suboptimal sequence of adaptations. For example, the system may adapt to handle a transient change, only to have to adapt back to the previous configuration moments later. If the cost of performing those two adaptations is higher than their benefit, the system would be better off not adapting. However, reactive approaches that decide based on immediate outcomes cannot avoid such inefficiencies. This issue is exacerbated when tactics are not instantaneous. First, it may be possible that by the time the tactic completes, the situation that prompted the change has already subsided. Second, it may happen that starting an adaptation tactic prevents the system from reacting to subsequent changes until the tactic completes (e.g., rebooting a server is not possible while the server is being re-imaged). Another limitation of a reactive approach is most evident when the objective is to maintain system invariants. Clearly, a reactive approach results in invariants being violated before the system can react to restore them.

Ignoring tactic latency has negative consequences as well. Consider a situation that can be handled with either of two tactics, a or b. When the adaptation criterion is to restore an invariant, then both tactics are equally good. However, if tactic a is faster than tactic b, it would not be appropriate to consider them as being equally good—restoring invariants faster should be preferred. In a utility-based approach, there is a similar problem. If tactic b is marginally better than a in terms of instantaneous utility improvement, the decision would favor tactic b. However, taking into account the fact that tactic a would start accruing the utility improvement sooner than b, it may very well be that a is better when considering utility accrued over time. In these situations, it is not possible to reason appropriately about adaptation unless the latency of adaptation tactics is considered.

In this dissertation, we present an approach that addresses the limitations of *reactive time-agnostic adaptation* by considering time in self-adaptation as a first-class concern. Instead of being reactive, this solution uses a look-ahead horizon to proactively adapt, taking into account not only the current conditions, but how they are estimated to evolve. In addition, the solution explicitly takes into account tactic latency when deciding how to adapt, improving the outcome of adaptation, both for utility-based adaptation, and for adaptation criteria that involve restoring invariants as quickly as possible. Furthermore, explicit consideration of tactic latency allows the decision to go beyond simply selecting the best adaptation tactic, and further improve the outcome of adaptation by complementing a slow—but better—tactic with a fast tactic that can be executed concurrently to ameliorate the problem while the slower tactic executes.

There are many kinds of systems that would have improved efficiency using this approach. These are some examples:

- *Cloud computing.* One of the advantages of cloud computing is providing elastic com-

[1]We use the term *effect* of a tactic to refer to the changes that the tactic produces on the aspects of the systems that are directly controllable, such as adding a server. It does not include the possible effect on emergent properties, such as the change in response time due to the addition of a server.

puting capacity that can adjust dynamically to the load on the system. One limitation of current approaches is that they assume that the control actions used to make these adjustments are immediate, when in reality they are not [47]. The approach in this thesis could help improve the effectiveness of adaptation for cloud computing. For example, by considering the latency of enlisting additional capacity, it could proactively start the adaptation, or decide that a short workload burst is better handled by another tactic, or perhaps that both tactics are needed concurrently. Furthermore, it could even dynamically decide which provider to use at different times based on their provisioning time. In fact, the decision would not necessarily always favor faster provisioning, if for example, the system can afford, thanks to the proactive adaptation, to wait longer for the provisioning of the new capacity by a cheaper provider.

- *Wireless sensor networks.* In general these systems present a trade-off between the frequency of sensor reading reports and their battery life. Proactive adaptation can help improving the battery life without compromising the mission supported by the sensor network by adapting the reporting frequency ahead of environment changes [113]. For example, in forest fire detection, the reporting frequency can be increased if the temperature is predicted to rise. Furthermore, some adaptations may require updating the firmware of the nodes, an operation that can take over a minute for updating a single node [102].
- *Cyber-physical systems.* Some adaptations that could be used in cyber-physical systems have latency that is due to physics. For example, since different formations in multi-robot teams have different qualities, an adaptation may require switching between them. Doing this has latency because of the time required for the robots to physically move in relation to their teammates. As another example, a GPS may be turned off to save power; however, turning it back on is an adaptation that is not instantaneous because the "time to first fix" may be about a minute [99].
- *Systems with human actuators.* Even though the goal of self-adaptation is to minimize the dependency on humans, self-adaptive systems often rely on humans to perform actions on the physical world. For example, scaling out in industrial control systems may require the connection of a device by a human operator [20]. Adaptation tactics that involve human actions have considerable latency, which must be taken into account when deciding how to adapt.
- *Security.* When a system is under cyber-attack, selecting the appropriate defensive action requires knowing the attacker's tactics, techniques, and procedures (TTPs). Tactics to observe the attacker to gather more information the TTPs being used take time. Meanwhile the attacker may be exfiltrating data, causing harm to the enterprise. Explicitly considering the latency of the observation and the defensive tactics would allow a self-protecting system to consider the consequences of not taking defensive action vs. the consequences of taking inappropriate defensive actions. Also, *moving target defense* is an approach to security whose goal is to change some aspect of the system frequently in order to make it difficult for attackers to exploit knowledge about the system, or to maintain a foothold on the system. One problem of this approach is that it typically focuses exclusively on security, and it can blindly and constantly change the system, without regards to other im-

portant qualities of the system, such as performance. This kind of defense would benefit from reasoning about proactive adaptations in the context of other qualities of the system, and from the consideration of how long the moving target tactics take to execute.

The rest of this chapter introduces a motivating example that will be used throughout the dissertation to present our approach, presents the thesis that this dissertation investigates, and provides a road map for the rest of the dissertation.

1.1 Motivating Example

To illustrate the problem solved by this work and the approaches presented in this thesis, we will use RUBiS, an open-source benchmark application that implements the functionality of an auctions website [126].[2] This application has been widely used for research in web application performance, and various areas of cloud computing [39, 50, 74, 122]. RUBiS is a multi-tier web application consisting of a web server tier that receives requests from clients using browsers, and a database tier. In our version of the system, we also include a load balancer to support multiple servers in the web tier, as shown in Figure 1.1.[3] The load balancer distributes the requests arriving at the website among the web servers. When a client requests a web page using a browser, the web server processing the request accesses the database tier to get the data needed to render the page with dynamic content. The request arrival rate, which induces the workload on the system, changes over time, and we want the system to be able to self-adapt to best deal with this changing environment.[4]

There are two ways the system can deal with changes in the workload induced by the clients. First, as is typical in elastic computing, the system can add/remove servers to/from the pool of servers connected to the load balancer. Second, it can adjust the proportion of responses that include optional content (e.g., advertisement or suggested products) through a control known as a *dimmer*,[5] since not including the optional content in the response to a request reduces the load imposed on the system.

The goal of self-adaptation in this system is to maximize the utility provided by the system at the minimum cost. The utility is computed according to a service level agreement (SLA) with rewards for meeting the average response time requirement in a measurement interval, and penalties for not meeting it [77]. The cost is proportional to the number of servers used. The SLA specifies a threshold T for the average response time requirement. The utility obtained in

[2]Further details about RUBiS are provided in Chapter 8. In addition, a second system used for the validation of the thesis is presented in Chapter 8.

[3]Architecturally, RUBiS is similar to Znn.com, a model problem adopted by the self-adaptive systems research community [30].

[4]RUBiS was not developed as a self-adaptive system, but we added self-adaptive capabilities to it by adding an adaptation layer.

[5]We use a version of RUBiS extended with *brownout* capability [83]. Instead of being limited to a binary choice in which all or no responses include the optional content, brownout uses the dimmer setting as a way to control the proportion of responses that include the optional content, with 1 being the setting in which all responses include the optional content, 0 when no one does (i.e., blackout), and values in between for different levels of brownout.

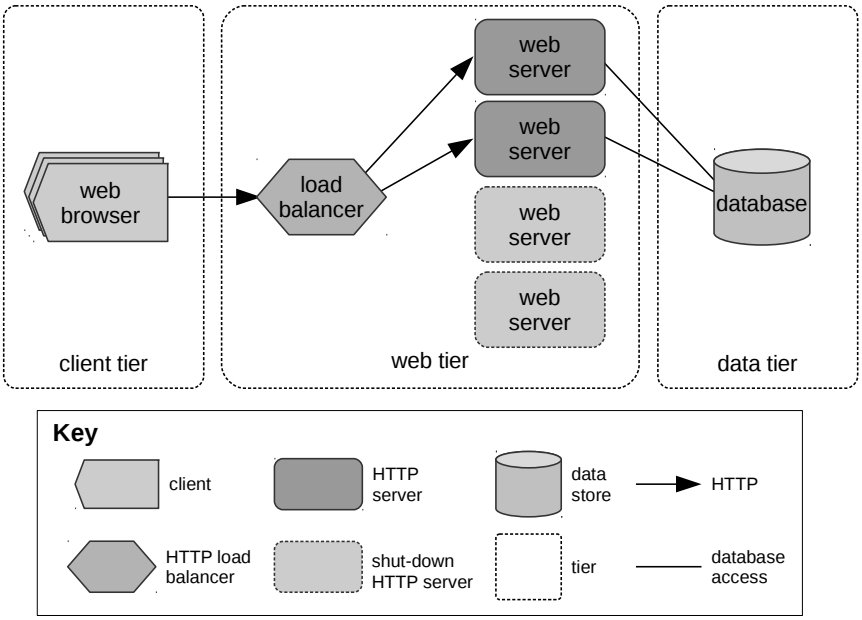


Figure 1.1: RUBiS system architecture.

an interval depends on whether the response time requirement is met or not, as given by[6]

$$U = \begin{cases} \tau a(dR_O + (1-d)R_M) & \text{if } r \leq T \\ \tau \min(0, a - \kappa)R_O & \text{if } r > T \end{cases} \tag{1.1}$$

where τ is the length of the interval, a is the average request rate, r is the average response time, d is the dimmer value, κ is the maximum request rate the site is capable of handling with optional content, and R_M and R_O are the rewards for serving a request with mandatory and optional content, respectively, with $R_O > R_M$.

If cost were not a concern, the sever pool would be configured with all the servers on-line at all times. However, that would not satisfy the secondary goal of minimizing operation costs, which depends on the number of servers being used. Furthermore, there can be events (such as a spike in sales) in which not even the maximum number of servers supported by the pool would be sufficient to meet the response time requirement unless the dimmer is used as well.

To understand the limitations of a reactive time-agnostic approach to self-adaptation, consider how it would handle the following scenario in RUBiS. Assume that the website is in a steady state, meeting the response time requirement with a high dimmer setting, and two out of four servers on-line. Traffic to the website starts to increase. At first, it can still be handled with the current configuration, but at some point, the response time goes beyond the acceptable threshold specified in the SLA. The adaptation manager detects the problem—after clients have already experienced unacceptable response time—and decides to start a new server. Starting a server takes time; meanwhile, traffic continues to increase and the response time continues to get worse. When the new server comes on-line, the response time improves, but still does not meet the

[6]When we introduce equations, any element not previously defined is defined in the prose that follows.

requirement. The adaptation manager detects that, and adds another server. This time, when the server finishes booting, the response time is finally brought back to an acceptable level.

Compare what just happened with how the approach presented in this thesis handles the situation. When the traffic to the website starts to increase, but before it causes the response time to become unacceptable, the adaptation manager detects through its look-ahead approach that the response time will become unacceptable in the near future if the up-trend for the traffic persists. Furthermore, it is able to determine that adding just one server will not be enough, and decides that two servers are needed. Because it is aware that adding a server takes time, it does not wait for the response time to become unacceptable, and instead starts the new servers before that happens. Notwithstanding, moments later, due to a higher than expected increase in traffic, and in spite of having started the available servers in advance, it determines that the response time requirement will go above the acceptable threshold before the servers come on-line. To avoid that, and because it is aware that lowering the dimmer is a low-latency tactic that can executed concurrently with bringing the additional servers on-line, the adaptation manager lowers the dimmer right before the response time gets too high. As soon as the servers come on-line, it restores the dimmer to its highest setting. Thanks to the proactive latency-aware approach and the use of concurrent tactic execution, the response time never goes above the acceptable threshold.

1.2 Thesis

This dissertation shows that:

> **We can improve the effectiveness of self-adaptation over reactive time-agnostic adaptation by (a) explicitly considering the latency of adaptation tactics, (b) adapting proactively, and (c) potentially allowing concurrent execution of adaptation tactics.**

Next, we elaborate on the different elements of the thesis statement.

> **We can improve the effectiveness of self-adaptation...**

Self-adaptation is done to achieve some particular goal even in the face of environment change. Examples of adaptation goals include continuing to satisfy requirements [18], maximizing utility [30], and self-protection [129]. Clearly, a self-adaptation approach is effective if it achieves its goal. Yet, there are different levels of effectiveness. Consider the case, for example, of self-adaptation to satisfy requirements. Ideally, the approach would ensure that the system never fails to meet a requirement. However, that may not be possible, especially if the self-adaptive system reacts to the detection of an unsatisfied requirement. In that case, some time will pass from the time that the system fails to satisfy the requirement until the problem is fixed. If we compare two approaches, the one that takes longer to restore the requirement satisfaction, all else being equal, is less effective.

Timeliness is a desired quality of self-adaption [69, 116, 127]. As such, it ought to be included directly or indirectly in a measure of effectiveness of self-adaptation. In the previous example, the effectiveness could be measured by the amount or proportion of time the system

meets all its requirements. In other cases, timeliness is indirectly measured by a metric relevant to the goal of adaptation. For example, if the amount of sensitive data exfiltrated is a measure of self-protection (less is better), then the longer a self-adaptive approach takes to stop the exfiltration, the worse it will be with respect to that metric. For utility-based approaches, a measure of effectiveness could be the total utility that the system provides over its execution, as this encompasses both how much utility changes and how long the adaptation takes, thus measuring both the impact and the timeliness of the adaptation.

The claim of the thesis is that effectiveness of self-adaption can be improved by using a combination of the following ideas.

...by (a) explicitly considering the latency of adaptation tactics,...

Adaptation tactics take some time to produce their intended effect: that is, they have latency. Different tactics have different latencies. For instance, some tactics involve just changing a property of a component, which can often be done with very low latency. Others take a considerable amount of time. For example, adding a processing node to a Cassandra database takes about 180 seconds [47]. Tactic latency is for the most part ignored by existing self-adaptation approaches, especially when deciding how to adapt.

Ignoring tactic latency has several consequences that negatively affect adaptation effectiveness. These include:

- assuming that the beneficial impact of the tactic will be produced immediately, distorting the projected results. This can lead to misguided adaptation decisions.
- assuming that the tactic will complete before the need for it subsides, which may lead to wasted resources for adaptation, and even preventing the selection of a faster useful tactic.
- not being able to reason about trade-offs between the impact different adaptation tactics provide and the time they take. This can result, for example, in deciding to adapt in a way that takes longer, to get a marginally better result in the end.
- not being able to start proactively an adaptation tactic in time so that it completes by the time its effect is needed. This means that the system will lag with respect to the adaptations needed to deal with changes in the environment.
- preventing the use of other incompatible tactics while a tactic with considerable latency executes (e.g., not being able to remove a server while it is being added). Consequently, an adaptation choice made at some point constrains the possible adaptations in subsequent decisions.

By explicitly considering tactic latency, in what we refer to as *latency-aware adaptation*, our approach avoids these issues, thereby improving adaptation effectiveness.

...(b) adapting proactively,...

Most self-adaptation approaches are reactive: that is, they adapt after detecting that the system configuration is not suitable or not the best for the current environment state. As noted previously when discussing effectiveness, the longer it takes for the system to adapt reactively, the less effective the adaptation is. Even if we ignore the adaptation decision time, reactive approaches have a lower bound on the reaction time (i.e., the time it takes to produce an effect on the system)

imposed by the latency of the adaptation tactics. By adapting proactively and explicitly taking into account tactic latency, we can make the reaction time virtually zero, improving the effectiveness of the system. Proactive adaptation, not only allows the system to start the tactics so that they complete in time, but also to decide *how* to adapt, taking into account how the environment will evolve in the near future. For example, suppose that either tactic a or b can be used to deal with an impending environment change, and tactic a is marginally better. A reactive approach would choose tactic a. Now, if tactic b is predicted to be required in order to deal with a subsequent environment change, and there is an adaptation cost, it would be better to use tactic b in the first place, and avoid the second adaptation cost. Since proactive adaptation uses a look-ahead approach, it is able to make the correct decision in such case.

...and (c) potentially allowing concurrent execution of adaptation tactics.

Current self-adaptation approaches select one adaptation tactic or one adaptation strategy composed of a sequence of tactics. That is, they start the execution of one tactic, wait until it completes, and then start the following one, and so on. Adaptation effectiveness can be improved by supporting concurrent execution of adaptation tactics.[7] There are two reasons why effectiveness is improved. One is the simple reason that parallelization reduces the total amount of time a sequence of computations takes to execute. The most interesting case, though, is when a fast tactic can be used to provide a partial improvement while a slower, but better, tactic is executing. For example, consider a case in which an increase in system load can be handled by adding a server, which takes a considerable amount of time. While the server is being added, the system may fail to meet maximum response time requirements. However, that can be avoided by concurrently executing a fast tactic to reduce the content fidelity. In that way, the response time requirement is met. Once the tactic for adding the server completes, the content fidelity can be restored using another fast tactic. Note that this kind of reasoning is only possible with latency-aware adaptation.

1.3 Solution Approach

Proactive latency-aware adaptation improves self-adaptation effectiveness by integrating timing considerations in a three-pronged approach:

- *latency awareness:* explicitly considers how long tactics take to execute, both to account for the delay in producing their effect, and to avoid solutions that are infeasible when the time dimension is considered.
- *proactivity:* leverages knowledge or predictions of the future states of the environment to start adaptation tactics with the necessary lead time so that they can complete on time, and to avoid unnecessary adaptations.
- *concurrent tactic execution:* exploits non-conflicting tactics to speed up adaptations that involve multiple tactics, and to support long-latency tactics with faster ones that can produce intermediate results sooner.

[7] Not all tactics can be executed concurrently. This is discussed in Section 3.4.

Making an adaptation decision with these characteristics requires solving an optimization problem to select the adaptation path that maximizes a utility function over a finite look-ahead horizon. The decision horizon is necessary to achieve proactivity, so that the system prepares for upcoming needs, and also to account for the delayed effect of adaptation tactics with latency. Additionally, an adaptation action started at given time can constrain the adaptations feasible in the near future. That is the reason why an adaptation path—a sequence of adaptation actions—must be considered to assess the utility that the system could accrue over the look-ahead horizon. Making decisions with such look-ahead requires relying on predictions of the state of the environment over the decision horizon; however, these predictions are subject to uncertainty. Since this is a problem of selecting adaptation actions in the context of the probabilistic behavior of the environment, Markov decision processes (MDP) are a suitable approach. An MDP is a model for sequential decision making when the outcome of taking an action in a given state is uncertain [121]. One of the elements that define an MDP is a probabilistic transition function, which gives the probability of reaching different target states when an action is taken in a given state (more details are provided in Section 3.3). In our context, a state in the MDP refers to the joint state of the system and the environment. This means that constructing the transition function of the MDP requires taking into account the dynamics of both the environment and the system simultaneously. Given all the possible interactions between the different, and possibly concurrent, adaptation tactics, the system, and the environment, constructing the MDP is a complex task. Furthermore, since the predicted behavior of the environment, which is only known at run time, is part of it, the MDP cannot be constructed off-line.

In this thesis, we present two different proactive latency-aware (PLA) solution approaches that involve the construction and solution of an MDP to make adaptation decisions. We refer to these as the main solution approaches, since they solve the full PLA adaptation decision problem. A third solution approach uses the PLA principles to make adaptation decisions, but limits that adaptive behavior to the selection of predefined combinations of adaptation tactics. Although less flexible in terms of the adaption decisions it can make, this solution approach is more scalable than the other two.

One approach, named PLA-PMC, is based on probabilistic model checking (PMC), a formal verification technique used to analyze systems with stochastic behavior [95]. The approach consists of (i) creating off-line formal specifications of the adaptation tactics and the system; (ii) periodically generating at run time a model to represent the stochastic behavior of the environment; and (iii) using a probabilistic model checker at run time to synthesize the optimal strategy that maximizes the expected value of a utility function over the decision horizon by analyzing the composition of the models of the tactics, the system and the environment. From the optimal solution computed by the model checker, we can extract the set of tactics that must be started in order to achieve the adaptation goal (e.g., utility maximization).

One drawback of PLA-PMC is that the model checker has to construct the underlying MDP every time an adaptation decision has to be made because the probabilistic transitions of the MDP depend on the stochastic behavior of the environment, which can only be estimated at run time. Consequently, the overhead of constructing the MDP must be incurred every time an adaptation decision has to be made with PLA-PMC.

The second approach, PLA-SDP, practically eliminates the run-time overhead of constructing the MDP by doing most of that *off-line*. Using formal modeling and analysis, the approach ex-

haustively considers the many possible system states, and combinations of tactics, including their concurrent execution when possible. At run time, the adaptation decision is made by solving the MDP using stochastic dynamic programming (SDP) principles [120], weaving in the stochastic environment model as the solution is computed.

Even though PLA-SDP can make adaptation decisions much faster than PLA-PMC, it does so at the expense of modifiability. It exploits the structure of the adaptation decision problem to achieve its speed, and although it can efficiently solve a class of adaptation decision problems, handling different problems with other characteristics would require redesigning its algorithm. For example, in the scope of this thesis, tactic latencies are assumed to be deterministic. Relaxing that assumption to support probabilistic latencies in future work would be straightforward with PLA-PMC, whereas PLA-SDP would require changes to both the MDP construction part, and the solution algorithm. Another advantage of PLA-PMC is that it is the gold standard for the solution due to the optimality of the solution computed by the model checker. We envision that advances in the field to handle growing classes of adaptation decision problems could be lead by PMC-based solutions, with other solutions, such as SDP-based ones, following behind, striving to solve the same class of problems more efficiently. That is what happened during the development of the approaches in this thesis. Furthermore, there is ongoing work to improve the efficiency of probabilistic model checking [57, 92], so its performance will likely be less of a problem in the future. Hence, both approaches present distinct contributions to address the same problems, and are therefore presented in this thesis.

Both PLA-PMC and PLA-SDP are tactic-based approaches; that is, they make decisions by making adaptation plans combining adaptation tactics. Another kind of approach is the selection of an adaptation *strategy* to achieve the adaptation goal. A strategy is a decision tree built out of tactics [28]. For example, a strategy to reduce the response time in RUBiS, could add a server, wait to check if it reduces the response time to a satisfactory level, and if it does not, reduce the fidelity level—all as part of a single strategy. In this way, an adaptation strategy captures the process a system administrator would follow to repair a problem. Since strategies are pre-planned ways in which tactics can be combined, strategy-based adaptation lacks the flexibility to generate all the solutions that tactic-based adaptation could provide, some of which could be better. One benefit of strategies, though, is that they reduce the solution space, and, consequently, make the adaptation decision faster. This could be desirable for systems in which tactic-based adaptation would not be fast enough due to a large number of tactics available in the system. In addition, there may be cases in which the designers of an adaptive system do not want it to have the latitude to adapt in unforeseen ways that tactic-based adaptation confers, but restrict it to use only strategies that they define. For example, robots are sometimes required to move like humans to match the humans' intuitive expectation of their behavior [75], even though that behavior may be suboptimal or overly restrictive for the robots [125]. For such situations where the use of strategies is desired, we also present SB-PLA, a strategy-based PLA adaptation approach that builds on the ideas of PLA-PMC. In this approach, when an adaptation decision has to be made, the latency of the adaptation tactics involved in a strategy and the predicted evolution of the environment are considered to assess the impact of the applicable strategies. With SB-PLA, systems that cannot use tactic-based adaptation for the aforementioned reasons can still benefit from the improvements that proactive latency-aware adaptation brings.

1.4 Research Claims

The main claim of this thesis is that the effectiveness of self-adaptation can be improved by using the approach presented in this dissertation, and is repeated here for completeness, together with a brief explanation of how that claim is established in this dissertation. In addition, there are two other research claims that aim to demonstrate the practicality of the approach.

Claim 1. *The approach improves the effectiveness of self-adaptation.* This claim, already elaborated in 1.2, is validated by comparing the approach with a self-adaptation approach that lacks the timing aspects of self-adaptation (i.e., it is not proactive, is latency-agnostic, and does not use concurrent tactics) in two different systems with different adaptation criteria.

Claim 2. *The approach is applicable to different kinds of systems.*
Two important aspects that will differ across classes of systems are their adaptation goal and the repertoire of tactics they use. For example, information systems that have to deal with changing workload will have adaptation tactics that allow them to scale the resources they use (e.g., adding servers), or to shape the workload (e.g., removing optional content such as product suggestions), whereas a cyber-physical system may have tactics that, for example affect its physical configuration to best suit the environment conditions. In addition, the adaptation criteria may be different. For example, in an information system, the main trade-off may be between cost and performance, with the adaptation criterion being to maximize a utility function that aggregates both. However, in a cyber-physical system, the adaptation criteria may be to avoid violation of constraints (e.g., the vehicle does not crash), while maximizing some metric related to its mission (e.g., number of objects found). To validate this claim, we applied the approach to two different systems in different domains, each with different kinds of tactics, and different adaptation criteria.

Claim 3. *The approach scales to systems of realistic size.* The size of the space within which the adaptation decision must search for the new system configuration increases with the number of adaptation tactics, the number of parameters that adaptation can control, and the length and granularity of the look-ahead horizon. For the approach to be practical, it has to be able to handle adaptation in a search space with a size representative of what might be encountered in practice. Furthermore, it must be able to carry out the adaptation decisions in a reasonable time, for example, without overrunning a designated decision period. To validate this claim we applied the approach to the two systems mentioned above, and in addition, we ran simulations to artificially increase some of the parameters that define the adaptation search space beyond what our experimental platform supports.

1.5 Contributions

The main contributions of this thesis are:

- a conceptual framework for proactive latency-aware adaptation that describes the PLA self-adaptation loop, how the three elements—proactivity, latency-awareness, and concur-

rent tactic execution—are combined, and defines the PLA adaptation decision problem independent of how it is solved. (Chapter 3)

- three solution approaches that realize the conceptual framework for proactive latency-aware adaptation:
  - PLA-PMC, a solution approach based on probabilistic model checking, which, given the optimality it achieves with exhaustive analysis, serves as the gold standard in terms of solution effectiveness. PLA-PMC not only provides a reference with which to compare other approaches, but also is a solution that can be relatively easily adapted to handle problems that require extensions of PLA. In addition, with its modifiability, PLA-PMC provides a solution approach suitable to lead future extensions of PLA. (Chapter 4)
  - PLA-SDP, a solution approach based on stochastic dynamic programming principles, which exploits the PLA adaptation decision problem structure to make adaptation decisions an order of magnitude faster than PLA-PMC, while retaining optimality (as computed by PLA-PMC) (Chapter 5)
  - SB-PLA, a solution approach that uses the PLA principles to improve adaptation decisions based on adaptation strategies. In this way, PLA can be used in systems for which it is desired to limit the adaptive behavior to a repertoire of predefined and trusted adaptation strategies. Additionally, SB-PLA is more scalable than the other two solution approaches. (Chapter 7)
- support for a variety of forms of utility to drive adaptation decisions. In addition to the often used form of additive utility maximization, this approach allows combining ways in which utility is gained with requirements on the satisfaction of a probabilistic constraint. This makes it possible to express, for example, that the system gains utility as long as it does not fail, and that the probability of failure must be kept above some threshold. This support for a variety of forms for the utility function makes our approach applicable to systems with different adaptation goals. (Chapter 6)

The research presented in this thesis has resulted in the following peer-reviewed publications:

- J. Cámara, G. A. Moreno, and D. Garlan. Stochastic game analysis and latency awareness for proactive self-adaptation. *International Symposium on Software Engineering for Adaptive and Self-Managing Systems* (SEAMS 2014) [22]

  This paper introduced the idea of proactive latency aware adaptation, showing that it can improve the effectiveness of self-adaptation compared to time-agnostic adaptation.

- J. Cámara, G. A. Moreno, D. Garlan, and B. Schmerl. Analyzing Latency-Aware Self-Adaptation Using Stochastic Games and Simulations. *ACM Transactions on Autonomous and Adaptive Systems*. January 2016 [23].

  This paper extended the previous work to support multiple adaptation tactics, and introduced the use of formal modeling and analysis to compute off-line all the possible ways in which the system configuration can be changed through the application of adaptation tactics.

- G. A. Moreno, J. Cámara, D. Garlan, and B. Schmerl. Proactive self-adaptation under uncertainty: a probabilistic model checking approach. *Joint Meeting on Foundations of Software Engineering* (ESEC/FSE 2015) [108]

  This paper presented PLA-PMC, a novel approach that uses probabilistic model checking to solve the PLA adaptation decision problem considering the uncertainty of the environment.

- G. A. Moreno, J. Cámara, D. Garlan, and B. Schmerl. Efficient Decision-Making under Uncertainty for Proactive Self-Adaptation. *International Conference on Autonomic Computing* (ICAC 2016) [109]

  This paper presented PLA-SDP, a PLA solution approach based on stochastic dynamic programing that makes adaptation decisions an order of magnitude faster than the previous approach.

1.6 Dissertation Outline

The rest of this thesis is organized as follows. Chapter 2 presents related work. Chapter 3 introduces the concept of proactive latency-aware adaptation, defines the adaptation decision problem, and gives some background needed for the solution approaches. Chapter 4 presents PLA-PMC, a solution approach based on probabilistic model checking. Chapter 5 describes PLA-SDP, a solution approach that computes the same adaptation decisions as PLA-PMC, but does it much faster. Chapter 6 extends the approach to support additional notions of utility. Chapter 7 presents SB-PLA, a solution approach that uses the principles of PLA but is based on adaptation strategies rather than just tactics. The validation of the thesis is presented in Chapter 8. Chapter 9 has a discussion of the thesis, including an analysis of the contributions of the different elements of our approach, and the rationale for the two solutions proposed. In addition, the limitation of the approach are discussed, and future areas of work are presented. Chapter 10 concludes the thesis with a summary of its contributions and brief recap.

Chapter 2

Related Work

In this chapter we present related work organized in topics relevant for this thesis: self-adaptive systems, proactive adaptation, adaptation latency, model predictive control, MDP-based adaptation, and runtime quantitative verification. For each area, we note the similarities and difference between existing works and ours.

2.1 Self-Adaptive Systems

In the last 15 years, there has been a substantial amount of research in self-adaptive systems, encompassing several adaptivity properties such as self-protecting, self-optimizing, and self-healing, which are usually referred to as *self-** properties. Salehie and Tahvildari, and Krupitzer et al. present two good surveys of the field [88, 127].

A key concept used to engineer self-adaptive systems is that of a feedback loop, which monitors the state of the system and its environment, and adapts the system as needed to achieve the desired self-* properties. Embracing feedback loops for software and making them explicit has been identified as a crucial factor for building self-adaptive systems [15]. One explicit form of feedback loop adopted by the self-adaptive systems community is the Monitor-Analyze-Plan-Execute-Knowledge (MAPE-K) loop [81]. The MAPE elements cover the activities that must be performed in the feedback loop: (i) monitoring the system and the environment; (ii) analyzing the information collected and deciding if the system needs to adapt; (iii) planning how to adapt; and (iv) executing the adaptation. The four activities share a knowledge base or repository that integrates them. The approach presented in this thesis fits in the MAPE-K loop model of self-adaptation. However, for the reasons explained in Chapter 3, we combine the analysis and planning phases into a single phase that we refer to as *adaptation decision*.

In architecture-based self-adaptation, the knowledge includes a model of the architecture of the running system that is maintained at runtime and used to reason about the changes that should be made to the system to achieve the quality attributes linked to the adaptation goals [54]. Architecture-based approaches offer several benefits, including providing the right level of abstraction to describe dynamic change in terms of components and connectors, as opposed to low-level algorithmic details [87]. Additionally, architecture-based self-adaptation is a good fit for the approach described in this thesis because it allows one to use existing architecture

analyses [72, 86, 132] to assess the impact of potential changes in the system and the environment. One example of architecture-based self-adaptation is embodied in the Rainbow framework, which provides reusable infrastructure with customization points to facilitate the implementation of self-adaptive systems (see Section 7.1 for more details) [53].

Although not completely dependent on architecture models, this thesis leverages the concept of architecture-based self-adaptation in that the reasoning required to make adaptation decisions is based on a model of the system at a level of abstraction that is sufficient to analyze the properties that self-adaptation is aiming to control. In some cases, these models are not software architecture models, as in one of the systems used for the validation of the thesis (see Section 8.1.2).

Rainbow, like most self-adaptation approaches, only supports reactive adaptation [88]; that is, it only adapts after the system has already failed to satisfy some requirement. In addition, when deciding how to adapt, the latency of adaptation tactics is ignored, and consequently, the fact that the environment will change while the adaptation is executed is also not considered. Our approach addresses these limitations, aiming to deal with issues proactively, before a requirement is not satisfied, and considering the latency of the different adaptation alternatives when deciding how to adapt.

Even though Rainbow was designed for reactive adaptation, our approach was integrated into this framework mostly using its customization points, except for the support for concurrent execution of adaptation tactics, which required deeper changes to the component that manages the adaptation execution. When used with Rainbow, the architecture models used to make adaptation decisions with our approach are explicitly represented in Acme, an architecture description language [52].

2.2 Proactive Adaptation

Several taxonomies classify adaptation approaches into *reactive* and *proactive* [62, 88, 127]. In reactive approaches the system adapts to deal with changes that have already happened, whereas in proactive approaches the system adapts in anticipation of changes that are going to happen. In its simplest case proactive adaptation could be thought of as reacting to the detection (through prediction, for example) that a change will happen. However, this thesis deals with proactive adaptation under uncertainty, which requires considering different possible future realizations of the environment, in which the change does and does not happen. Thus, simply reacting to a predicted future event is not sufficient.

One of the defining characteristics of autonomic systems[1] is being *anticipatory*, defined as "[being] able to anticipate to the extent possible, its needs and behaviors and those of its context, and [being] able to manage itself proactively" [115]. That goal notwithstanding, the vast majority of the self-adaptive approaches are reactive [88, 127], and in their recent survey, Krupitzer et al. highlight proactive adaptation as a research challenge in the area of self-adaptive systems [88].

One area in which proactive adaptation has received considerable attention is service-based systems [17, 64, 106, 143] because of their reliance on third-party services whose quality of service (QoS) can change over time. In that setting, when a service failure or a QoS degradation

[1] *Autonomic* and *self-adaptive* are terms mostly used interchangeably in the literature, and in some cases the former includes the latter [127].

is detected, a penalty has already been incurred, for example, due to service-level agreement violations. Thus, proactive adaptation is needed to avoid such problems. Hielscher et al. propose a framework for proactive self-adaptation that uses online testing to detect problems before they happen in real transactions, and to trigger adaptation when tests fail [64]. Wang and Pazat use online prediction of QoS degradations to trigger preventive adaptations before SLAs are violated [143]. However, these approaches are limited in that they ignore the adaptation latency, and that their look-ahead is limited, for example by considering only the predicted QoS of services yet to be invoked in a service composition being executed.

Gmach et al. present a proactive/reactive approach to resource pool management [58]. In their approach, proactivity means allocating resources for an upcoming time interval of four hours based on historic workload predictions. A reactive adaptation manager is then used to compensate for prediction errors and change the allocation within the interval. Compared with this thesis, there are several important limitations of their approach: not considering adaptation latency, considering only one period in their look-ahead, and not being able to reason about long running tactics that could prevent others from executing.

Work on anticipatory dynamic configuration by Poladian et al. [118] is the closest to our approach. They demonstrated that when there is an adaptation cost or penalty, anticipatory adaptation outperforms reactive adaptation. Intuitively, if there is no cost associated with adaptation, a reactive approach could adapt at the time a condition requiring adaptation is detected without any negative consequence. However, when there is an adaptation cost, reactive adaptation is suboptimal. By leveraging environment predictions and using a look-ahead horizon, anticipatory adaptation can determine the best adaptation to carry out in order to maximize the sum of utility, taking into account how the environment state will evolve in the short term, and the penalties associated with adapting. Poladian's work, however, is limited by the fact that it ignores adaptation latency, which has the following consequences: (i) it cannot select between a fast and a slow adaptation, (ii) it is not proactive because it cannot start adaptations with the necessary lead time to complete by the time the environment changes, and (iii) it assumes that all configurations are feasible at all times. Furthermore, that work only supports maximization of expected additive utility as the reconfiguration goal, whereas ours support different forms of utility functions, as described in Chapter 6.

2.3 Adaptation Latency

Adaptation latency (i.e., how long the system takes to adapt) is a concern in autonomic computing, and has been proposed and used as a metric to evaluate adaptation approaches [11, 25, 48, 105]. Nevertheless, it is rarely taken into account as a factor in the adaptation decision. As Gambi et al. point out, adaptations are typically assumed to be immediate. So, they pose—but do not address—the research question of how knowledge of adaptation latency can be leveraged to improve the quality of the control exerted by the MAPE loop [47].

Adaptation latency is considered in some very specific situations in some work. Musliner considers adaptation latency by imposing a limit on the time to synthesize a controller for real-time autonomous systems [110]. However, in that work there are no distinct planning and execution phases, and thus there is no consideration of the latency of the different actions the system

could take to adapt. In the area of dynamic capacity management for data centers, the work of Gandhi et al. considers the setup time of servers, and is able to deal with unpredictable changes in load by being conservative about removing servers when the load goes down [49]. Their work is specifically tailored to adding and removing servers to a dynamic pool, a setting that resembles the example introduced in Chapter 1. However, their work cannot reason about other tactics that could be used instead of, or in combination with, tactics to control the number of servers. Zhang et al. propose a safe adaptation approach that can minimize the cost of adaptation, with adaptation duration being one such cost [144]. However, that cost is only considered once all the possible ways of reaching the desired target configuration have been found. That is, adaptation duration is not used when selecting among alternative target configurations. In their work for autonomic security management, Iannucci and Abdelwahed consider the latency of security actions when planning how to adapt to deal with security attacks [71]. However, the latency is only factored into the decision as part of the reward structure in an MDP, penalizing actions with longer latencies, but not actually taking into account how latency affects *when* actions changes the state of the system.

None of these approaches considers adaptation latency systematically as a first-class concern as this thesis does. In contrast to these other works, ours takes into account how the environment changes while the adaptation is carried out, what adaptation tactics would become infeasible during the execution of an adaptation, how utility changes while an adaptation is carried out, and how faster tactics can complement slower tactics when executed concurrently.

2.4 Model Predictive Control

Model predictive control (MPC) is an approach with roots in process control that selects control inputs to optimize forecasts of process behavior [124]. These forecasts or predictions are done using a process model, thus its name. Our approach shares these high-level ideas with MPC: (i) the use of a model to predict the future behavior of the system; (ii) the computation of a sequence of control actions, committing only to the first one; and (iii) the use of a receding horizon [19]. Although MPC has been used in other approaches to self-adaptation [3, 91, 137], our approach differs in the several significant ways. First, it takes into account that control actions executed at a given time may prevent other control actions from being applicable in subsequent time steps, as opposed to assuming that all control actions are applicable at all times. Second, it considers tactic latency during the selection of the adaptation action(s), not just as an adaptation cost, but modeling how the execution of the tactics affects the applicability of other tactics while the tactics execute (over possibly multiple time intervals). Furthermore, our approach is able to decide between fast and slow adaptation tactics. Third, it considers the possible concurrent adaptation tactics during the decision, not just as a way to speed up the execution of the adaptation. Fourth, it considers the transition probabilities of the environment supporting a richer stochastic model of the environment instead of treating the predictions for the environment state at each time interval over the decision horizon independently.

2.5 MDP-Based Adaptation Decisions

In this thesis, Markov Decision Processes (MDP) are used to model the adaptation decision process. There are approaches that use reinforcement learning to gradually learn the optimal policy for the underlying MDP [1, 9]. Their advantage is not requiring the construction of the MDP, which in our case is built out of models of the system and the adaptation tactics provided at design time, combined with models of the environment generated at run time. However, those approaches need time to learn the dynamics of the system, and have to execute possibly inadequate adaptations to learn their effect. Also, if a new adaptation tactic is added to the system after it has learned the underlying MDP, it needs to learn again the underlying model with that new tactic, especially considering how that tactic would behave when used in parallel with other tactics. Our approach does not require learning, and therefore, does not have these issues. But on the other hand, its performance does not benefit from experience, as learning does. In Section 9.4, we propose how learning could be incorporated to our approach in future work.

Naskos et al. use MDPs to make cloud elasticity decisions [111]. Their approach focuses on tactics to add and remove servers, and consequently, it cannot decide between alternative tactics, nor support concurrent tactics. Iannucci and Abdelwahed use MDPs to compute policies to deal with security attacks [71]. The main difference with our use of MDPs is that their work does not consider how the environment evolves over time while the system is adapting, focusing only on how the system state evolves. In addition, they only consider latency to favor faster tactics, since their approach is tailored to dealing with security attacks, in which it is desired to contain or clean the attack as fast as possible.

2.6 Runtime Quantitative Verification

PLA-PMC, one of the solution approaches for adaptation decisions presented in this thesis, involves the use of probabilistic model checking. Calinescu et al. proposed the use of model checking and quantitative verification techniques at run time to ensure the dependability of self-adaptive systems [18]. In their approach, referred to as *runtime quantitative verification* (RQV), information gathered through the self-adaptive system's monitoring capability is used to update parameters in the formal model of the system, which is then used to detect or predict requirements violations. If that is detected, the same quantitative verification techniques can be used to select, for example, the configuration less likely to result in an unsatisfied requirement. One difficulty in achieving their vision is that the underlying verification technique, probabilistic model checking, can be slow for certain systems, especially if the model checker has to be invoked multiple times for each adaptation decision. However, Gerasimou et al. recently showed how the overhead and execution time of this approach can be reduced by combining caching, look-ahead, and near-optimal reconfiguration [56]. They used the PRISM probabilistic model checker [96] in a simulation of a self-adaptive unmanned underwater vehicle.

There are two main differences between that use of verification, and the use of probabilistic model checking in this proposal. One is that for adaptation decision, RQV is used to quantify or verify properties of each possible configuration, one at a time, and that information is then used to select a target configuration outside of the model checking process. In contrast, PLA-

PMC uses the model checker to find the best adaptation by having the model checker synthesize a strategy,[2] whose first action is the adaptation action that must be executed; that is, the model checker is invoked only once per adaptation decision. The second difference is that they do the verification of individual configurations in the context of a snapshot of the environment state. In PLA-PMC, on the other hand, the verification analyzes sequences of adaptation in the context of an evolving environment.

[2]The model checker can synthesize a strategy, which is the resolution of the nondeterminism in the input model that maximizes the expectation of a utility function.

Chapter 3

Proactive Latency-Aware Adaptation

Today most self-adaptation approaches are reactive, making adaptation decisions based on current conditions [88]. Unless there is an adaptation cost, being reactive is not a problem if the system can adapt very quickly, because at any point, the system can rapidly change to best deal with the conditions at that moment. However, as we have already noted, not all adaptation tactics are instantaneous. For example, provisioning a new virtual machine in the cloud can take a few minutes [104]. We refer to the period of time between when a tactic is started and when its effect[1] is produced as *tactic latency*. The problem with tactics that have non-trivial latency is that not all system configurations are possible at all times. For instance, if adding a new server to a system takes two minutes, it is not possible to reach a system configuration with one more server in one minute. The only way to have that additional server on time is to start its addition *proactively*, taking into account the latency of that tactic.

Tactic latency also matters when the system can use tactics with different latencies to deal with the same situation. For example, an alternative to adding capacity with a new server, is to reduce load by reducing the quality of service (QoS); something that can typically be done with a much faster tactic. In a situation like this, considering not only the effect of the tactics on the system, but also their latency when deciding how to adapt can result in more effective adaptations.

Latency-awareness is even more useful when concurrent tactic execution is supported. In that case, it is possible to complement slow tactics with fast ones if they do not interfere with each other. For example, suppose that at some point the tactic to add a server is started because that was deemed appropriate to handle a predicted increase in the request rate to the system. However, the next time the system evaluates its state—but before the tactic to add a server completes—the request rate is worse than was estimated. In this case, the system can reduce the QoS—and the load—right away using a fast tactic.

Another effect of tactic latency is that the execution of a tactic with considerable latency can prevent the use of other incompatible tactics while it executes (e.g., removing a server while it is being added). Consequently, an adaptation choice made at some point constrains the possible adaptations in subsequent decisions.

[1] We refer to the direct effect of the tactic on the structure and/or properties of the system, and not the indirect effect they may be intended to produce. For example, when adding a new server, the effect is the system having one more active server, and not the reduction of its response time.

Proactive latency-aware adaptation, as introduced in Chapter 1, improves self-adaptation effectiveness by considering both the current and anticipated adaptation needs of the system, and taking into account the latency of adaptation tactics. To recap, the key pillars of PLA are:

- *latency awareness:* explicitly considers how long tactics take to execute, both to account for the delay in producing their effect, and to avoid solutions that are infeasible when the time dimension is considered.
- *proactivity:* leverages knowledge or predictions of the future states of the environment to start adaptation tactics with the necessary lead time so that they can complete on time, and to avoid unnecessary adaptations.
- *concurrent tactic execution:* exploits non-conflicting tactics to speed up adaptations that involve multiple tactics, and to complement long-latency tactics with faster ones that can produce intermediate results sooner.

In this chapter, we present an overview of the approach, the definition of the PLA adaptation decision problem, and the elements that the different solution approaches have in common, including the underlying Markov decision process, and the non-interference criteria for concurrent tactics.

3.1 Approach Overview

Our approach fits in the general class of self-adaptation architectures based on explicit closed-loop control such as the monitor, analyze, plan, and execute with knowledge (MAPE-K) loop depicted in Figure 3.1 [81]. The MAPE phases cover the activities that must be performed in the control loop: (i) monitoring the system and the environment; (ii) analyzing the information collected and deciding if the system needs to adapt; (iii) planning how to adapt; and (iv) executing the adaptation. The four activities share a knowledge base or repository that integrates them. These notional elements are realized as follows in our approach.

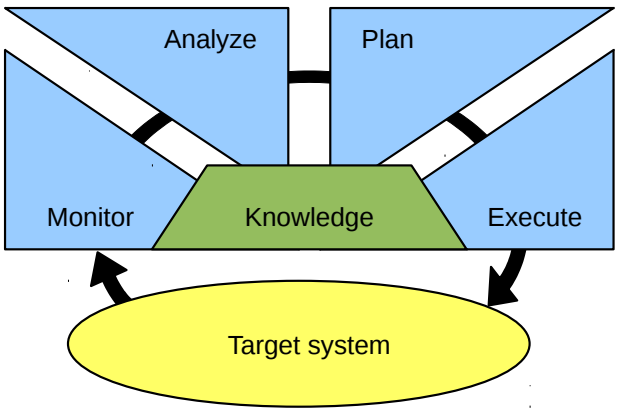


Figure 3.1: MAPE-K self-adaptation loop.

Knowledge model. As in other architecture-based self-adaptation approaches [54], we use an abstract representation of the system that captures important system characteristics and properties

as the knowledge that is used to reason about the possible adaptations. For RUBiS, for example, this model includes the number of servers, the number of active servers (i.e., those connected to the load balancer and able to process requests), the maximum number of servers supported, the current dimmer setting, and the observed average response time. In general, the model must have all the information that is necessary to determine whether adaptation tactics are applicable, and to compute the utility function[2] that drives adaptation decisions.

Because some adaptation tactics have latency larger than the period of the control loop, it is also necessary to keep track, in the model, of the adaptation tactics that are being executed, along with information about the progress they have made (or equivalently, when they are expected to complete). For example, for RUBiS, the model differentiates between active servers and servers that are not active yet. For the latter, the expected time at which they will become active (i.e., when the tactic to add a server will be completed) is kept in the model.

In addition, the model has information about the environment, which, in the case of RUBiS, includes the observed request arrival rate, and estimations of arrival rates in the near future.

Monitoring. Observations of the system and environment are collected, aggregated as needed, and used to update the model. In RUBiS, for example, the request arrival rate at the load balancer is monitored and its average and standard deviation is reflected in the model. In terms of architectural changes, when a server finishes booting and is connected to the load balancer, the monitoring marks the server as active in the model.

Adaptation Decision. Even though MAPE-K has distinct phases for analyzing the system to determine if adaptation is needed, and for planning how to adapt, these are combined into a single activity in our approach, as shown in Figure 3.2. When the goal of self-adaptation is to maximize a utility function, determining whether it is possible to adapt the system to a configuration that will give higher utility—the analysis part—implies finding such a configuration—the planning part. In our approach, the adaptation decision phase is run periodically, at a fixed interval τ.[3] A single computation of the solution of the adaptation decision problem (defined in Section 3.2), simultaneously determines both whether adaptation is required, and what tactics should be used, if needed. The output of the adaptation decision is a (possibly empty) set of adaptation tactics to be executed.

Execution. The execution manager is a component that receives the set of tactics computed by the adaptation decision, and executes them. It executes asynchronously relative to the adaptation decision, so that if it has to execute a tactic with latency larger than the decision period (e.g., adding a server), the adaptation decision can still be made according to its period. Being able to do so allows the approach to complement slow tactics with fast ones if they do not interfere with each other. For example, suppose at some point, only the tactic to add a server is started because it was determined that it was going to be sufficient to handle a predicted increase in the arrival

[2]To make the presentation simpler, we focus on utility maximization in the first chapters of the dissertation. In Chapter 6, we present other forms of utility functions that include constraint satisfaction.

[3]Since the decision is done periodically, we also refer to τ as the *period* of the decision.

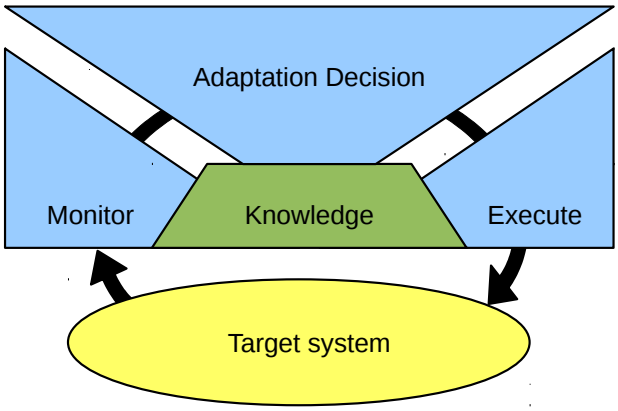


Figure 3.2: PLA self-adaptation loop.

rate. However, in the following decision period—and before the tactic to add server completes—the realization of the environment is worse than it was estimated. In this case, the adaptation decision can instruct the execution manager to execute the tactic to decrease the dimmer value, a fast tactic. The execution manager can execute these adaptation tactics in parallel; thus, it can change the dimmer value right away, without waiting for the other tactic to complete. Further details about the concurrency of tactics is given in Section 3.4.

3.2 Adaptation Decision Problem

In any given system, self-adaptation is done to achieve a particular goal. The goal could be to satisfy requirements such as response time at the minimum cost; to maximize the chargeable fees according to an SLA; or to satisfy certain constraints, such as maintaining the probability of failure below a threshold. Therefore, the self-adaptive system must make the appropriate adaptation decisions to support its goal.

We start with the assumption that the adaptation goal can be encoded as *maximizing aggregate utility*[4] over the execution of the system, where *utility* is a measure that can be taken periodically, at each decision interval, and represents the contribution of the system in that interval to the overall adaptation goal. A simple example is when utility is the amount in dollars that can be charged according to an SLA for a given interval, and the goal is to maximize the sum of the fees that can be charged over the execution of the system. However, there is no need for utility to be expressed in concrete units such as dollars, and it can actually be any measure. In Chapter 6, we present an extension that supports other notions of utility, for example, involving both utility maximization and probabilistic constraint satisfaction.

Making an adaptation decision requires being able to compute the utility that the current and other system configurations can provide in the current and future states of the environment. We assume that utility can be computed as a function of the state of the system and its environment,

[4]Unless otherwise indicated (e.g., with the other notions of utility presented in Chapter 6), *aggregate* is the sum of the utility obtained in each period.

and denote it as $U(c, e)$, where c is a system configuration,[5] and e is an environment state.[6] Note that even if there are many properties that would be needed to define the complete system state (e.g., state of internal variables, open ports, etc.), only those needed to compute the utility function and to determine whether tactics are applicable must be part of the system configuration. In RUBiS, for example, given the number of active servers and the dimmer setting for the system, and the request arrival rate of the environment, we can estimate the response time, and, in turn, the utility that that system configuration achieves in that environment state.

At a high level, the adaptation decision answers the question of what adaptation tactic(s) should be started now, if any, to maximize the aggregate utility that the system will provide in the rest of its execution, considering that the system will keep adapting as necessary in future decision intervals. Poladian et al. showed that reacting to the current situation without looking ahead can result in suboptimal solutions when there is an adaptation cost [118]. We argue a similar case for situations where adaptations have latency, even if there is no adaptation cost. When there is no adaptation cost, tactics do not directly affect utility. Rather, they change the system configuration, which in turn results in a change in utility. If adaptation tactics had no latency, the system could adopt any configuration anytime, and thus no look-ahead would be necessary for optimal adaptation decisions. However, when tactics have latency, it takes some time for the system to adapt to a new configuration. Therefore, the configuration of the system at time t constrains the possible configurations at a later time $t + \tau$ if τ is smaller than the latency of at least one of the adaptation tactics. For instance, if the current configuration has one server at time t, and the latency to add a server is λ, with $\tau < \lambda$, then all system configurations with more than one active server are not feasible at time $t + \tau$. That is, the configuration of the system at any given time constrains the possible configurations at a later time. Consequently, it is not possible to find the best configuration, or the adaptation to get to it, without looking ahead to see which configurations will be needed in the future.

Although the decision approach must look ahead, it is not practical to look too far into the future because of computational complexity, and because the uncertainty of the environment predictions increases as they get further into the future. Therefore, the adaptation decision uses look-ahead with a finite horizon of H decision intervals, and the question it answers is what adaptation tactics should be started now, if any, to maximize the aggregate utility the system will provide over the horizon.[7]

Making an adaptation decision with these characteristics requires solving an optimization problem to select the adaptation path that maximizes the aggregate utility over a finite look-ahead horizon. This requires relying on predictions of the state of the environment over the decision horizon, which in general have uncertainty. Since this is a problem of selecting adaptation actions in the context of the probabilistic behavior of the environment, Markov decision processes (MDP) are a suitable approach. Both PLA-PMC and PLA-SDP use MDPs as the underlying decision model. The next section provides a brief background on MDPs.

[5]The terms *system configuration* and *system state* are used interchangeably.

[6]We do not consider utility at a granularity finer than the decision interval; that is, U is the utility for a decision interval.

[7]The selection of values for parameters H and τ is discussed in Chapter 9.

3.3 Markov Decision Process

A *Markov decision process* (MDP) is a model for sequential decision making under uncertainty [121], and is also suitable for modeling systems with a mix of probabilistic and nondeterministic behavior [92]. An MDP is a tuple $\mathcal{M} = \langle S, s_I, A, \Delta, r \rangle$, where

- S is a finite set of states
- $s_I \in S$ is an initial state
- A is a finite set of actions
- $\Delta : S \times A \to \mathcal{D}(S)$ is a probabilistic transition function (typically partial, since not all actions may be enabled in all states), with $\mathcal{D}(S)$ denoting the set of discrete probability distributions over S, and
- $r : S \to \mathbb{R}_{\geq 0}$ is a reward function mapping each state to a non-negative reward.[8]

In the MDP, the system starts in state s_I, and its state evolves in discrete time steps. When the system is in state s, a set of actions $A(s)$ are enabled. Any action $a \in A(s)$ can be taken, and the next system state, s' is determined by the probabilistic transition function, with $s' = \Delta(s, a)$. When it transitions to state s', the system then accrues reward $r(s')$.

Since the MDP does not specify how the action selection is made, the system is *underspecified*, allowing nondeterministic behavior. A *policy*[9] prescribes how the action is selected in each state, thereby removing the nondeterminism. Policies can be memory-dependent, if the action selection depends on the history (i.e., the sequence of states visited and actions taken so far), or they can be memoryless. In addition, they can be randomized if the action is drawn from a probability distribution over the action set, or deterministic. In this work, we use memoryless deterministic policies. Therefore, a policy is a function $\sigma : S \to A$ that maps states to actions directly.

3.4 Adaptation Tactics and Concurrency

The main approaches presented in this thesis are based on adaptation tactics. An *adaptation tactic* is an action primitive that changes the system [28]. Tactics can change properties of elements in the system, such as changing the dimmer setting in the load balancer in RUBiS, or changing the structure of the system, such as adding a new server. Even though tactics may involve a sequence of system-level operations (e.g., adding a new server involves not only adding the server, but also connecting it to the system's load balancer), tactics are considered atomic operations that leave the system in a consistent state. Although tactics can be combined into higher level constructs called *adaptation strategies*, which are discussed in Chapter 7, tactic-based adaptation gives the most flexibility for deciding how to adapt without having to take into account how to keep the system in a consistent state—something that would be required if adaptation decisions were done in terms of lower level operations.

[8] The reward can depend additionally on the action, with $r : S \times A \to \mathbb{R}_{\geq 0}$, but the simpler definition suffices here.

[9] In the literature for MDPs, a policy is also referred to as an *adversary*, or a *strategy*. We avoid the latter to prevent confusion with the use of the term in the self-adaptive systems literature.

In this work, we assume that tactics have deterministic effect and latency.[10] By *effect*, we mean the effect of the tactic on the structure and settable properties of the elements of the system, and not on emergent properties, such as response time. Furthermore, each tactic has an applicability condition, a predicate over the state of the system that determines if the tactic can be used. Therefore, the only reason a tactic would not produce its intended effect is if it failed to carry out the change on the system.

When making adaptation decisions, we assume the latency of a tactic to be known and deterministic. However, that does not mean that in reality it has to be so. For example, we can use the average latency as the latency of the tactic used to make adaptation decisions. How much of an effect this approximation has on the outcome of the decision depends on the variance of the actual latency in relation to the decision interval. For example, if the latency variance and the decision interval are on the same order of magnitude, the effect of the approximation would be noticeable. But if the variance is only a few seconds, and the decision is made every one minute, for example, the approximation is unlikely to have an effect on the effectiveness of the approach.[11]

One of the reasons PLA can achieve better adaptation effectiveness is that it leverages concurrent tactic execution. By doing so, it can complement a slow tactic with a fast one when necessary, or make multiple changes at once without having to sequence them. The adaptation decision could assume that all tactics can be executed concurrently, and let the execution manager deal with the concurrency. However, in practice tactics may conflict with each other. For example, since tactics are considered atomic operations, it may not be possible to remove a server while it is being added. In such cases, the execution manager would not be able to carry out an adaptation decision that assumed concurrent execution, and would likely have to sequence the tactics. Instead, we assume that for each tactic it is possible to generate a list of conflicting tactics statically, and use that list to constrain the result of adaptation decisions to those that are actually executable. Even though determining which tactics are conflicting is not in the scope of this thesis, there is an approach that can be used to do that. In their work on preemptable adaptation strategies, Raheja et al. use an approximation of rely-guarantee reasoning to determine whether strategies interfere with each other [123]. Using the same idea, a tactic could guarantee that it only modifies some subset of the system, and rely on no other tactic modifying that same subset of the system. Since a tactic already expresses which parts of the system it modifies, it would not be difficult to compute the set of conflicting tactics for a given tactic.

3.5 Environment Model

The goal of the adaptation decision is to decide how to adapt to maximize the utility the system will accrue over the look-ahead horizon. However, utility is a function of both the system configuration and the environment state. Therefore, deciding with a look-ahead horizon requires predicting the near future states of the environment. These predictions are not perfect though, and consequently, they are subject to uncertainty. In Esfahani and Malek's list of sources of

[10] This and other assumptions are discussed in Chapter 9.
[11] This is further discussed in Chapter 9.

uncertainty that affect self-adaptive systems, this corresponds to uncertainty of *parameters over time* [40].

In this work, we assume that the actions of the system do not affect the environment.[12] The environment can be modeled as a stochastic process in which the random variable representing the state of the environment has one realization at each time step, with a time step being equal to the decision period τ. In particular, our approach uses discrete-time Markov chains (DTMCs) to model the probabilistic behavior of the environment. A DTMC is a tuple $\langle S, s_I, P \rangle$, where

- S is a finite set of states
- $s_I \in S$ is an initial state, and
- $P : S \times S \to [0, 1]$ is a transition probability matrix where $\sum_{s' \in S} P(s, s') = 1, \forall s \in S$.

Each element $P(s, s')$ in the transition probability matrix represents the probability that the next state of the process will be s', given that the current state is s. The main difference between an MDP and a DTMC is that in the latter there is no nondeterminism, and no action selection to be made; that is, its behavior is completely probabilistic. Nevertheless, a DTMC can be mapped to an MDP with a single action that is enabled in every state, thus making the model fully probabilistic without having any nondeterminism. This will be relevant later on when we compose the environment DTMC with the system MDP.

Using a DTMC to model the environment, each state represents a realization of the environment, and the probabilistic transitions defined by P occur at discrete time steps, probabilistically determining the environment state in the following decision period. As long as the environment model is encoded in this way, there is no particular requirement for how the model is constructed, or what particular structure the DTMC has. We provide an example here of how the model of the predicted environment behavior over the decision horizon is built for the RUBiS case. A different approach is used in the DART example presented in Section 8.1.2.

In RUBiS, the environment of the system consists of the load created by the users, which is summed up by the request arrival rate. A model of the environment, then, has to encode predictions of the request arrival rate over the decision horizon. Such predictions can be made using using a time series predictor, such as the autoregressive (AR) predictor that is part of the RPS toolkit [35]. The monitoring component measures the requests arrival rate at the load balancer. At the beginning of each decision period, the knowledge model is updated with the average request arrival rate for the previous period. This observation is supplied to the time series predictor so that it can update its internal model. Using the predictor, it is possible to obtain estimations for the average request rate for the next decision period, given the past observations. Since the estimation has an error with a normal distribution, the time series predictor provides the variance associated with the estimation.

To create a DTMC that captures both the prediction of the environment states and its uncertainty, we construct a probability tree like the one partially shown in Figure 3.3, which can be encoded as a DTMC. The root of the tree corresponds to the current state of the environment, each node represents a possible realization of the environment, and its children represent realizations conditioned on the parent, with the edges' label representing the probability of the child realization given that the parent was realized. Creating a small number of branches at each node

[12] We discuss this assumption in Chapter 9.

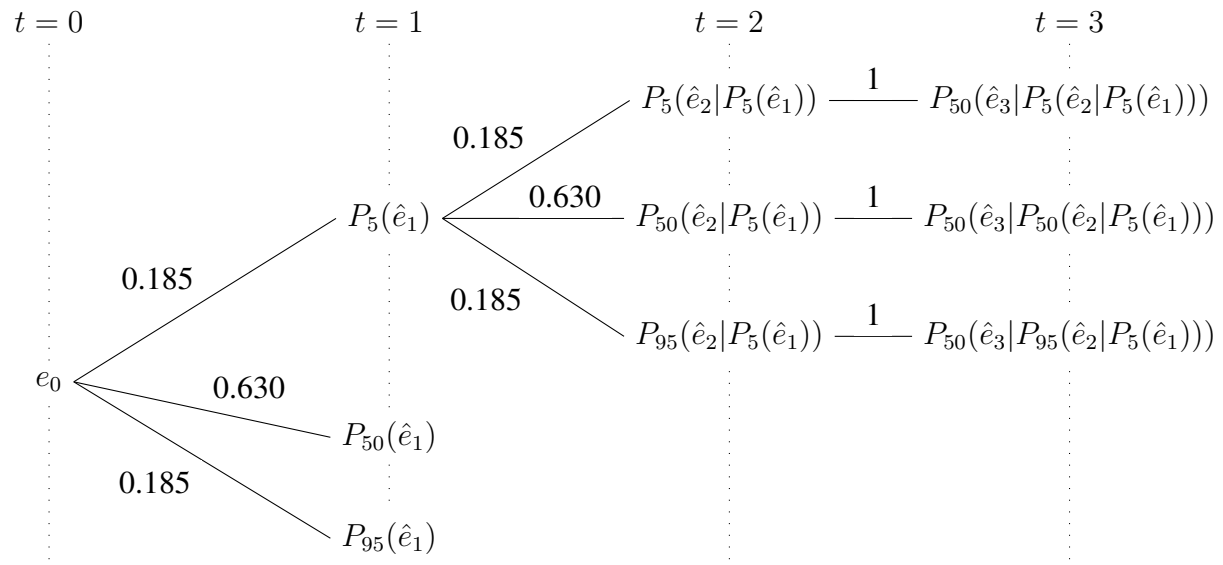


Figure 3.3: Environment probability tree.

requires discretizing the probability distribution of the estimation for the following period. Usually, three-point discrete-distribution approximations are used for constructing probability trees for decision making. For example, we use the Extended Pearson-Tukey (EP-T) three-point approximation [80]. This approximation consists of three points that correspond to the 5th, 50th, and 95th percentiles of the estimation distribution, with probabilities 0.185, 0.630, and 0.185, respectively.

The construction of the probability tree starts with the root, which is the current state of the environment, e_0 in Figure 3.3. Using the predictor, we obtain the distribution for the estimation for the following period, $\hat{e}_1$. Note that the predictor has already seen the past realizations of the environment, up to e_0, so the prediction is implicitly conditioned on the past observations. Using the estimation $\hat{e}_1$, and the EP-T discrete approximation, three children of the root are created. In Figure 3.3, the nodes $P_k(\hat{e}_1)$ represent the kth percentile of the distribution of the estimation $\hat{e}_1$. To continue the expansion of the tree, each child is visited and its children created in the same way. However, the estimation for these children must be conditioned on the parent. This is achieved by cloning the predictor (to avoid disturbing the state of the original predictor), and supplying to it the state of the environment at the parent, as if that state would have actually been the realization of the environment. In that way, when we obtain the prediction for the following period, the prediction will be conditioned on the parent.

In principle, it would be possible to continue the expansion of the probability tree up to a depth equal to the length of the look-ahead horizon. However, the further into the future we get (i.e., the deeper in the tree), the higher the uncertainty of the predictions, and the larger the resulting state space. In their use of probability trees, Poladian et al. found that they could limit the branching depth without much impact on the quality of the solution [118]. We take a similar approach, limiting the branching depth in the tree to two levels (as illustrated in Figure 3.3), and, beyond that, continuing the extension of the branches without any further branching up to a depth equal to the horizon.

3.6 Summary

In this chapter, we have introduced proactive latency-aware adaptation, and defined the PLA adaptation decision problem, for which solution approaches will be presented in chapters 4 and 5. The formulation of the problem, and consequently its solutions, are based on the following assumptions, which are discussed in Chapter 9:

1. The adaptation goal can be expressed with one of the utility forms presented in Chapter 6.
2. Utility can be computed as a function of the state of the system and its environment. This, in turn, requires being able to estimate the system measures of performance, such as response time, that are used to compute utility.
3. Adaptation tactics have deterministic effect on the structure and properties of the system.
4. The actions of the system through adaptation tactics do not affect the evolution of the environment.

In addition, we have shown how the PLA adaptation decision fits in the self-adaptation loop, and provided background on concepts that are common among the approaches, including Markov decision processes, our model of adaptation tactics, and environment modeling.

Chapter 4

Probabilistic Model Checking Approach

In this chapter, we present PLA-PMC, one of the approaches for proactive latency-aware adaptation.[1] PLA-PMC is based on probabilistic model checking, a formal verification technique used to analyze systems with stochastic behavior [95]. The approach consists of (i) creating offline formal specifications of the adaptation tactics and the system; (ii) generating periodically, at run time, a model to represent the stochastic behavior of the environment; and (iii) using a probabilistic model checker at run time to synthesize the optimal policy that maximizes the expectation of a utility function over the decision horizon by analyzing the MDP that results from the composition of the models of the tactics, the system and the environment.

The key idea is to leave the adaptation decisions in the model underspecified through nondeterminism, and have the model checker resolve the nondeterministic choices so that accumulated utility is maximized. Thanks to the use of formal specification and verification, it is straightforward for the approach to deal with the infeasibility of adaptations due to the latency of tactics, or conflicts between them. Furthermore, the same mechanism allows the adaptation decision to select multiple adaptation tactics to execute in parallel when they do not interfere with each other. In addition, the use of probabilistic model checking naturally handles the uncertainty of the environment.

The following section provides a brief background on probabilistic model checking, and the rest of the chapter describes the formal models used, and the adaptation decision using probabilistic model checking.

4.1 Probabilistic Model Checking

Probabilistic model checking is a set of techniques that enable the modeling and analysis of systems that exhibit stochastic behavior, allowing quantitative reasoning about probability and reward-based properties (e.g., resource usage, time, etc.). These techniques use state-transition systems augmented with probabilities to describe the system behavior.

Probabilistic model checking approaches employing formalisms that support the specification of nondeterminism, such as Markov decision processes, also enable the synthesis of policies guaranteed to achieve optimal expected probabilities and rewards [93]. Reasoning about policies

[1] Much of the material in this chapter is adapted from our original publication about the approach [108].

is a fundamental aspect of model checking MDPs, which enables checking for the existence of a policy that is able to optimize an objective expressed as a quantitative property in an extension of probabilistic computation-tree logic (PCTL) [13]. In MDPs, PCTL allows expressing properties such as $\mathsf{P}_{>p}[\mathsf{F}\ \phi]$, which states that under all resolutions of the nondeterminism, the probability of eventually satisfying the state formula ϕ is greater than p. Extensions to PCTL allow reasoning about reward-based properties [46]. For instance, the property $\mathsf{R}^{r}_{\geq x}[\mathsf{F}\ \phi]$ is true if the expected reward r accumulated until formula ϕ is satisfied is at least x. Further extensions to the reward operator enable the quantification of the maximum accrued reward r along paths that lead to states satisfying the state formula ϕ, which are expressed as $\mathsf{R}^{r}_{\mathsf{max}=?}[\mathsf{F}\ \phi]$ [46]. A typical example of a property employing the reward maximization operator is $\mathsf{R}^{\mathsf{time}}_{\mathsf{max}=?}[\mathsf{F}\ \mathsf{empty\_battery}]$, meaning "maximum amount of time that a cell phone can operate before its battery is fully discharged," where time is a reward function.

In our approach, we use the PRISM probabilistic model checker [96]. In addition to model checking MDPs with the characteristics previously described, PRISM provides a modeling language to express MDPs, avoiding the need to input them in terms of the tuple described in Section 3.3. A model in PRISM consists of one or more *modules* (a.k.a. *processes*). Modules have *variables* that represent its state. The behavior of a module is specified with *commands* [94] like:

$$[action]\ guard \text{ -> } p_1 : u_1 \text{+ } \ldots \text{ + } p_n : u_n;$$

where $guard$ is a predicate over the model variables, including variables in other modules. If the guard is true, the process can make one of the transitions represented by each update u_i, with the $+$ operator separating the different alternatives. An update assigns new values to the module's variables (using the primed name of the variable to refer to the post-state). Even though only variables of the module can be updated, the new values assigned to them can be functions of variables in any module. Each update has an assigned probability $p_i \in [0, 1]$. Multiple commands with overlapping guards introduce local nondeterminism.

The complete model is the parallel composition of its modules. So, for an MDP, when multiple commands are simultaneously enabled, even across multiple modules, the choice of which one is executed is nondeterministic. Composed modules synchronize (i.e., make transitions simultaneously) on shared actions, which are listed inside the square brackets at the beginning of the command.[2] A transition that specifies an action can only be taken if all the modules specifying that action can make a transition labeled with that action too.[3]

Our approach to decision-making leverages the capability of probabilistic model checking to synthesize policies that maximize expected reward, since it allows us to (i) deal with the uncertain behavior of the environment, and (ii) find optimal policies based on a reward function that is easily mapped to maximizing utility.[4] In the following section, we show how similar

[2]This list can be empty if a command does not synchronize on any action.

[3]By default, all modules are composed in parallel, synchronizing on all of their common actions. Although not used in this thesis, PRISM allows other forms of composition, such as renaming or hiding actions. For more details, see the PRISM manual (`http://www.prismmodelchecker.org/manual/`).

[4]In general, the terms *reward* and *utility* are interchangeable, although with the extensions presented in Chapter 6, utility is a broader notion that can impose constraints on how reward is gained.

properties that refer to utility-based rewards are employed for decision-making in the context of proactive latency-aware adaptation.

4.2 Adaptation Decision Overview

The overall approach to solve the PLA adaptation decision problem introduced in Section 3.2 using probabilistic model checking is to analyze the MDP model that results from the parallel composition of processes representing the behavior of the environment and the system, starting at the current time until the end of the decision horizon. These models are abstractions that contain only the properties of the system and the environment that are necessary to compute the value of the utility function, and to keep track of how the system changes when tactics are applied. As noted earlier, the key idea is to leave the decision to execute adaptation tactics underspecified in the model through nondeterministic behavior. That is, how the decision of whether to start the execution of tactics is made is not encoded in the model, but is included as a nondeterministic choice. Then, PRISM is used to synthesize a policy, resolving the nondeterminism in the model so that the expected accumulated utility over the horizon is maximized. This policy indicates which tactics must be used and when.

The following sections elaborate on the overall structure of the model used to make adaptation decisions; describe the models of the environment, system, and tactics; and provide more details about how the model checker is used with these models to solve the adaptation decision problem.

4.3 Formal Model

The model used to make adaptation decisions describes the behavior of the adaptive system in the context of the predicted behavior of the environment over the look-ahead horizon. As depicted in Figure 4.1, it is composed of modules (or equivalently, concurrent processes) for the environment, the adaptation tactics, and the system. The orchestration of these processes (i.e., when each one is allowed to make certain transitions) is critical to get the right behavior. However, it is as important to leave enough nondeterminism in the scheduling of the processes to give the model checker the freedom to decide when to use the adaptation tactics. The orchestration is accomplished via a module, *clock*, that controls the passing of time, and through the synchronization of modules on shared actions (the connectors in the figure).

The overall behavior of the model is as follows. The execution of the model is done at the granularity of evaluation periods, so one unit of model time corresponds to τ in real-world time. Time 0 in the model represents the beginning of the look-ahead horizon (i.e., the current time in the controlled system). At the beginning of each evaluation period in the execution of the model, the system has a chance to proactively adapt. Once the system has started the execution of a tactic (or passed up the opportunity to do so), the environment updates its state for the current period by taking a probabilistic transition according to its model. After that, the utility that the system provides for the period is computed, and accumulated. Then, time is advanced, and the process is repeated until the end of the horizon is reached.

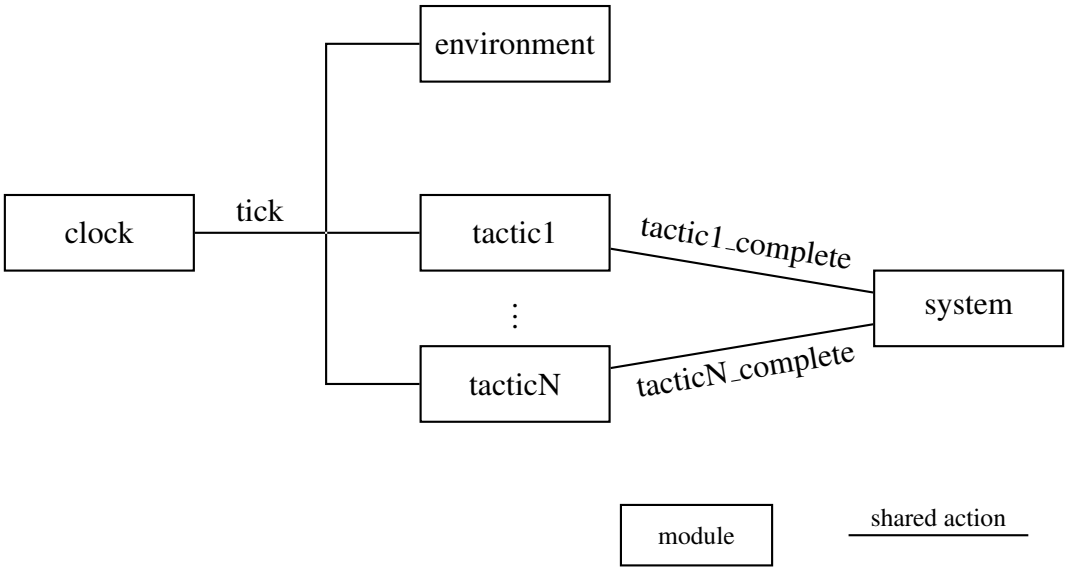


Figure 4.1: Module composition in adaptation decision model.

The specification of the clock module, called clk,[5] is shown in Listing 4.1. The state variables are defined and initialized in lines 2-3, with the constant HORIZON being the number of periods in the decision horizon. The behavior of the module is specified with the commands in lines 5-6. In each period, the module takes two transitions. First, the command labeled with the action tick advances the time. However, since the environment and the tactics share the same action, and a module can only execute a labeled command when all modules sharing the same label execute their corresponding command synchronously, clk will only be able to advance the time when the tactics and environment modules are ready to do so. After that happens, clk takes another transition labeled tack, with which the utility accumulation synchronizes.[6] This ensures that neither the system nor the environment change when the utility calculation is done. The reward structure util (lines 9-11) defines the reward function that will be maximized by the model checker, which in this case is the accumulation of the utility for the periods in the decision horizon.[7] Although not shown in the listing, periodUtility is a formula that encodes the system utility function for a single period. The utility function may need to estimate emergent properties (e.g., response time in the case of RUBiS), which depend on the state of the system and the environment. For example, the utility function for RUBiS uses queuing theory [84] to estimate the response time (see Appendix A). Since the utility is computed as a function of the environment and system state, their evolution over the decision horizon has to be modeled. The following sections describe their models.

[5] clock is a reserved word in PRISM.

[6] This means that the utility for the period is computed right after time is advanced, which is different than the conceptual schedule described before. This change allows a reduction of the state space without affecting the resulting decision.

[7] The utility function is translated with a large positive shift constant because PRISM does not support the modeling of negative rewards. This does not affect the result of the optimization.

```
1  module clk
2      time : [0..HORIZON] init 0;
3      readyToTick : bool init true;
4
5      [tick] readyToTick & time < HORIZON -> 1 : (time' = time + 1) & (readyToTick'=false);
6      [tack] !readyToTick -> 1 : (readyToTick'=true);
7  endmodule
8
9  rewards "util"
10     [tack] true : UTILITY_SHIFT + periodUtility;
11 endrewards
```

Listing 4.1: Clock module and reward structure.

4.4 Environment Model

Recall from Section 3.5, that we want to encode the model of the predicted, but uncertain behavior of the environment as a DTMC. More specifically, for PLA-PMC we want to model the evolution of the environment over the decision horizon as a PRISM module, so that it can be composed with the model of the system and tactics as shown in Figure 4.1. Including a DTMC in the PRISM model for an MDP is straightforward since, as noted before, a DTMC can be thought of as an MDP having no nondeterminism (i.e., no action selection), with only fully probabilistic behavior.

Each time an adaptation decision has to be made, the DTMC for the environment is generated, incorporating the latest environment predictions (see Section 3.5). Generating the PRISM encoding that represents the DTMC for the environment model is straightforward. Listing 4.2 shows a fragment of the specification in PRISM of the probability tree shown in Figure 3.3. Each node of the environment DTMC is assigned a unique index in the range $[0..N-1]$, where N is the number of nodes. A variable s in the env module with the same range represents the state of the environment as an index to a node in the DTMC. The transitions out of each node can be encoded directly as commands in PRISM. For example, the command in line 4 represents all the transitions out of the root node (s=0). The action tick is used to synchronize the transitions of the environment with the transitions of the clock and the system. The mapping from state s to the value of the state is encoded in the formula starting in line 9, using the conditional operator.[8] In this formula, constants such as P5_E_1 represent the value of the nodes of the probability tree, which, for the case of the RUBiS environment, is the request rate. In cases in which the environment is multi-dimensional, the values of the different dimensions corresponding to the nodes in the DTMC can be encoded using multiple formulas akin to the one starting in line 9.

4.5 System and Tactic Models

In addition to the environment state, the system configuration is also needed to compute the value of the utility function the adaptation decision is maximizing. This section presents how the evolution of the system state, as it is changed by adaptation tactics, is modeled. Unlike the

[8] An expression of the form (condition ? a : b) is a if condition is true, and b otherwise.

```
1  module env
2  s : [0..N−1] init 0;
3
4  [tick] s = 0 −> 0.185 : (s' = 1) + 0.63 : (s' = 2) + 0.185 : (s' = 3);
5  [tick] s = 1 −> 0.185 : (s' = 4) + 0.63 : (s' = 5) + 0.185 : (s' = 6);
6  ...
7  endmodule
8
9  formula stateValue = (s = 0 ? E_0 : 0) +
10                       (s = 1 ? P5_E_1 : 0) +
11                       (s = 2 ? P50_E_1 : 0) +
12                       ...
```

Listing 4.2: Environment module in PRISM.

environment model, the model of the system and its tactics does not change at run time except for a few initialization constants; and consequently, it can be constructed off-line. For better modularity, the system and each adaptation tactic are modeled as separate PRISM modules. Each tactic module synchronizes with the system module on an action that represents the completion of the adaptation tactic. Since the tactics are modeled as concurrent processes that synchronize only when they all have had a chance to execute, their ordering is nondeterministic. This, and the nondeterminism in the decision to start each tactic (described in Section 4.5.2), give sufficient flexibility to the model checker so that it can decide how to best schedule the adaptation tactics to achieve the adaptation goal.

4.5.1 System Model

The system model only has to keep track of the configuration information that is needed as input to the utility function and to evaluate the applicability conditions of the different tactics. In the case of RUBiS, this information includes the number of active servers, and the value of the dimmer. Emergent properties, such as the response time in RUBiS, are not part of the system state, since they are estimated as mentioned in Section 4.3. Note that the system model does not need to specify the behavior of the system. For example, it does not need to model the processing of requests in RUBiS. The only behavior that must be specified in the system model is how its state changes as a result of using of adaptation tactics.[9]

The specification of the system module for RUBiS is shown in Listing 4.3. Lines 2-3 define the two variables that capture the system configuration. They are initialized with constants ini_$*$ that represent the state of the system at the time that the adaptation decision is invoked; that is, at the beginning of the decision horizon. Although the actual dimmer property of the system can range in the continuous interval $[0, 1]$, it must be discretized to get a finite and manageable state space. In this example, the dimmer setting is discretized into a small number of integer levels (DIMMER_LEVELS in line 3), which can be converted to the corresponding continuous value when needed. Lines 5-8 have commands that capture how the system state is updated when each

[9]It is also possible to include system behavior that is not triggered by adaptation tactics. However, this would only make sense if such behavior affected adaptation decisions (e.g., changing the applicability conditions of adaptation tactics).

of the adaptation tactics completes. Since each command is synchronized with the completion of the corresponding tactic, the associated state updates can take place only when the tactic completes. For example, the command in line 5 is labeled with the action AddServer_complete, which is also shared by the module for the tactic to add a server, as will be shown later. In that way, the two have to synchronize on that action. When the tactic completes, the system module updates its state by increasing the number of servers. The sys module includes similar commands with labeled actions for other tactics. The suffix _start is used for the actions of tactics that are instantaneous, but this is just a convention that we have adopted.

```
1 module sys
2   servers : [1.. MAX_SERVERS] init ini_servers;
3   dimmer : [1.. DIMMER_LEVELS] init ini_dimmer;
4
5   [AddServer_complete] servers < MAX_SERVERS -> 1 : (servers' = servers + 1);
6   [RemoveServer_start] servers > 1 -> 1 : (servers' = servers - 1);
7   [IncDimmer_start] dimmer < DIMMER_LEVELS -> 1 : (dimmer' = dimmer + 1);
8   [DecDimmer_start] dimmer > 1 -> 1 : (dimmer' = dimmer - 1);
9 endmodule
```

Listing 4.3: System module in PRISM.

Even though we use RUBiS as an example to make the presentation concrete, the same pattern is used for different systems; that is, the system module consists of the representation of the system state, and the commands that model how it is affected by adaptation tactics (the models of the systems used for the validation are included in Appendix A and Appendix B).

4.5.2 Tactic Models

The responsibilities of each tactic module include determining if the tactic's applicability conditions are met; nondeterministically starting the tactic or passing the opportunity to do so; keeping track of the progress of the tactic (if it has latency); and synchronizing with the system module when the tactic completes. Again, we use tactics from the RUBiS example to present the approach, but the same pattern can be used for different tactics.

Tactics with latency. Listing 4.4 shows the model for the tactic to add a server. The state of the tactic is defined in lines 5-6. The variable AddServer_state is used to keep track of whether the tactic is executing or not (it is greater than 0 when the tactic is executing), and if it is, how much progress it has made. Tactic progress is tracked at the granularity of decision periods. For that reason, the upper bound of AddServer_state is the constant AddServer_LATENCY_PERIODS defined in line 2 by rounding up the latency of the tactic to the nearest decision interval boundary. Thus, when greater than zero, AddServer_LATENCY_PERIODS indicates how many periods the tactic has executed for. As was the case with the system state, the state of AddServer_state is initialized with a constant that represents its state at the time the adaptation decision is invoked. This is needed because the tactic may already be in progress when the adaptation decision is carried out, and if that is the case, the fact that the tactic is running must be taken into account

to avoid making decisions inconsistent with the state of the system.[10] This is the reason why the knowledge model needs to keep track of tactic execution.

```
1  formula AddServer_used = AddServer_state != 0;
2  const int AddServer_LATENCY_PERIODS = ceil(AddServer_LATENCY / PERIOD);
3
4  module AddServer
5      AddServer_state : [0.. AddServer_LATENCY_PERIODS] init ini_AddServer_state;
6      AddServer_go : bool init true;
7
8      // tactic applicable, start it
9      [AddServer_start] sys_go & AddServer_go // can go
10             & !AddServer_used // tactic has not been started
11             & AddServer_applicable
12         -> (AddServer_state' = 1) & (AddServer_go' = false);
13
14     // tactic applicable, but don't use it
15     [] sys_go & AddServer_go // can go
16             & !AddServer_used // tactic has not been started
17             & AddServer_applicable
18         -> (AddServer_go' = false);
19
20     // pass if the tactic is not applicable
21     [] sys_go & AddServer_go
22             & !AddServer_used // tactic has not been started
23             & !AddServer_applicable
24         -> 1 : (AddServer_go' = false);
25
26     // progress of the tactic
27     [] sys_go & AddServer_go
28             & !AddServer_used & AddServer_state < AddServer_LATENCY_PERIODS
29         -> 1 : (AddServer_state' = AddServer_state + 1) & (AddServer_go' = false);
30
31     // completion of the tactic
32     [AddServer_complete] sys_go & AddServer_go
33             & AddServer_state = AddServer_LATENCY_PERIODS // completed
34         -> 1 : (AddServer_state' = 0) & (AddServer_go' = true); // so that it can start again at this time if needed
35
36     [tick] !AddServer_go -> 1 : (AddServer_go' = true);
37 endmodule
```

Listing 4.4: PRISM model of a tactic with latency.

The variable AddServer_go is for internal bookkeeping, and has to do with the orchestration of the modules. At the beginning of each period this variable is true, as it is the sys_go predicate, which is simply an alias to readyToTick from the clock module. This means that the tactic has an opportunity to execute one of its enabled behaviors. After doing that, AddServer_go is set to false to force the tactic to wait until the next period by leaving only the transition labeled with tick enabled (line 36).

The guards for the different commands in the tactic module also check whether the tactic is already being used (predicate AddServer_used defined in line 1), and if the tactic is applicable. For the latter, the predicate AddServer_applicable, which will be explained later, is used.

[10] When a tactic is in progress, it cannot be started again. If that were needed, for example to support starting the addition of a server while another one is being added, multiple copies of the tactic would have to be included in the model.

When the tactic is enabled and applicable, the two commands starting in lines 9 and 15 are enabled with identical guards. These commands correspond to starting the execution of the tactic, and to passing up the chance to start it, respectively. Since these commands have no probability specified on their update portion, the model represents a nondeterministic choice between them. This is the key idea in this approach; that is, leaving the decision to start a tactic underspecified through nondeterminism, and then have the model checker resolve the nondeterminism in a way that maximizes the expected utility over the decision horizon.

When the tactic is enabled, but it is not applicable, it passes (lines 21-24). The commands starting in lines 27 and 32 model the progress of the tactic and its completion, respectively. The latter must synchronize with the system module on the action AddServer_complete, causing the system to reflect the change caused by the completion of the tactic in its state. Note that after the completion of the tactic, AddServer_go is left true[11] to make it possible for the tactic to start again in the same period.

Tactics with no latency. For tactics with negligible latency, the model can be simplified since there is no need to track their progress. Listing 4.5 shows the model of the tactic to increase the dimmer level in RUBiS. In this case, the state of the tactic is reduced to the two boolean variables in lines 2-3. The variable IncDimmer_go is true if this module has not yet used its chance to make a decision in the current period. IncDimmer_used is true if the tactic was used in this period.

If the tactic is enabled and applicable, the two commands starting in lines 5 and 10 are enabled, and since they do not specify a probability in their update part, they are chosen nondeterministically. Again, this underspecification is later resolved by the model checker in a way that maximizes the accumulated utility. The command starting in line 15 passes the turn for this tactic to do something when the tactic is not applicable. After either of these three commands has executed, the only enabled command is the one labeled with tick in line 19, making the tactic process synchronize with the rest of the tactics and the clock before moving on to the next period.

Applicability conditions. The previous tactic models relied on predicates with the _applicable suffix to determine when an adaptation tactic can be applied. These predicates include two classes of conditions. First, they ensure that the system is in a state in which the tactic can be applied, regardless of what other tactics are being executed. For example, this ensures that the tactic to add a server is not used when the system is already using the maximum number of servers. The second class of condition ensures that the tactic does not conflict with other tactics already being used. Listing 4.6 shows the definition of the applicability predicates for the two tactics presented previously. The first lines define the concurrency rules, with the $*$_compatible predicates indicating whether the tactic can be run given the other possibly interfering tactics that could be in use. For example, the tactic AddServer can be used unless the tactic RemoveServer is being used (line 2). This allows the adaptation decision to select non-conflicting tactics to execute concurrently (e.g., decreasing the dimmer value while a server is being added), while

[11]This part of the update is actually not needed in the PRISM model since AddServer_go is already true as required by the command's guard. However, we include it explicitly to show that it is intentional, since it breaks the pattern of the other commands.

```
1  module IncDimmer
2      IncDimmer_go : bool init true;
3      IncDimmer_used : bool init false;
4
5      [IncDimmer_start] sys_go & IncDimmer_go
6              & IncDimmer_applicable
7          -> (IncDimmer_go' = false) & (IncDimmer_used' = true);
8
9      // tactic applicable but not used
10     [] sys_go & IncDimmer_go
11             & IncDimmer_applicable
12         -> (IncDimmer_go' = false);
13
14     // pass if the tactic is not applicable
15     [] sys_go & IncDimmer_go
16             & !IncDimmer_applicable
17         -> 1 : (IncDimmer_go' = false);
18
19     [tick] !IncDimmer_go -> 1 : (IncDimmer_go' = true) & (IncDimmer_used' = false);
20 endmodule
```

Listing 4.5: PRISM model of a tactic with no latency.

avoiding the concurrent execution of conflicting tactics (e.g., adding a server while one is being removed).

```
1  // tactic concurrency rules
2  formula AddServer_compatible = !RemoveServer_used;
3  formula IncDimmer_compatible = !DecDimmer_used;
4
5  // applicability conditions
6  formula AddServer_applicable = servers < MAX_SERVERS & AddServer_compatible;
7  formula IncDimmer_applicable = dimmer < DIMMER_LEVELS & IncDimmer_compatible;
```

Listing 4.6: Tactic applicability predicates in PRISM.

4.6 Adaptation Decision

The adaptation decision can be made after the environment model has been constructed as described in Section 4.4. The input to the probabilistic model checker is the composition of the modules previously described. We also have to specify the property of the model that must hold under the policy we want the model checker to generate. In this case, the desired property is to maximize the accumulated utility over the look-ahead horizon. In PCTL extended with rewards, this property is expressed as

$$R^{\mathsf{util}}_{\mathsf{max}=?}[\mathsf{F}\ \mathsf{end}]$$

where util is the reward structure specified in the model (Listing 4.1, lines 9-11), and end is a predicate that indicates that the end of the look-ahead horizon has been reached by the clock module.

The policy synthesized by PRISM resolves the nondeterminism in the model, replacing nondeterministic choices with choices based on the state of the system and the environment. Because the behavior of the environment remains stochastic, it is not possible to extract from the policy what adaptation tactics should be used at all time steps in the horizon, since that decision depends on the future realizations of the environment. That notwithstanding, the choices made by the policy at time 0 are deterministic because they are made before the environment takes any probabilistic transition. Because these choices are exactly the ones that should be enacted at the current time in the controlled system (recall that time 0 in the model corresponds to the current time), it is sufficient to extract these from the policy and ignore future choices. The set of tactics extracted from the synthesized policy are handed off to the execution manager (see Figure 3.2), thus completing the adaptation decision phase.

4.7 Summary

In this chapter, we have presented PLA-PMC, an approach for proactive latency-aware adaptation under uncertainty that uses probabilistic model checking to make adaptation decisions. The approach uses a finite look-ahead horizon to find the adaptation that maximizes the expected utility accumulated over the decision horizon in the context of the uncertainty of the environment. The advantages of using probabilistic model checking are that (i) the adaptation decision is optimal over the horizon because the model checker selects the policy through a combination of mathematical models and exhaustive search; and (ii) it takes into account the stochastic behavior of the environment. Furthermore, the modular specification of tactics as separate processes combined with the use of tactic compatibility predicates allows the approach to deal naturally with the (in)feasibility of adaptations due to the latency of tactics, and the conflicts (or lack thereof) between them.

Chapter 5

Stochastic Dynamic Programming Approach

In Chapter 4, we presented an approach that uses probabilistic model checking to solve the proactive latency-aware adaptation decision problem. The probabilistic model checker takes as input a formal specification of the adaptive system and its stochastic environment, which is internally translated into an MDP, and solved. The solution to the MDP is the set of tactics that have to be started in order to achieve the adaptation goal (e.g., utility maximization). Using MDPs in this way, it is possible to reason about latency and uncertainty. However, the probabilistic transitions of the MDP depend on the predicted behavior of the environment, which can only be estimated at run time, and with a short horizon. Consequently, the overhead of constructing the MDP must be incurred at run time, every time an adaptation decision has to be made, so that the latest predictions of the environment behavior can be incorporated.

In this chapter we present PLA-SDP, an approach that practically eliminates the run-time overhead of constructing the MDP by doing most of that work *off-line*.[1] Using formal models, the approach exhaustively considers the many possible system states and combinations of tactics, including their concurrent execution when possible. At run time, the adaptation decision is made by solving the MDP through stochastic dynamic programming, weaving in the stochastic environment model as the solution is computed. Our experimental results (detailed in Chapter 8) show that this approach reduces the adaptation decision time by an order of magnitude compared to the PLA-PMC, while producing exactly the same results.

Figure 5.1 shows an overview of the elements of PLA-SDP and how they are used for making adaptation decisions. The strategy to reduce the time it takes to make an adaptation decision at run time is to avoid the run-time overhead of constructing the MDP. Since the model of the environment needed to build the complete MDP is not known until run time, constructing the complete MDP off-line is not possible. However, we can separate the aspects of the MDP that are known before run time—namely the system model and how adaptation tactics affect the system state—from the environment model. In that way, the system MDP can be constructed off-line. The system MDP is encoded as reachability predicates that specify whether a system configuration can be reached from another system configuration with the use of one or more

[1] This chapter is based on our original publication on the approach [109].

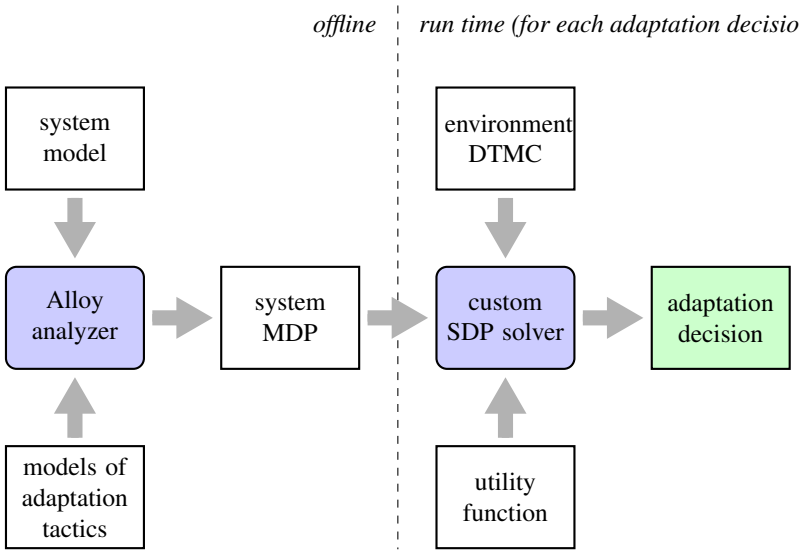


Figure 5.1: Elements of PLA-SDP.

adaptation tactics. These predicates are formally defined in Section 5.1. In order to make run-time adaptation decisions, it is necessary to compute the list of system configuration pairs that satisfy these predicates for that particular system with its adaptation tactics. This explicit representation of the predicates is computed from specifications of the system and its adaptation tactics using the Alloy analyzer [76], a tool that analyzes these specifications exhaustively to find all the possible configuration pairs that satisfy these predicates. This computation, described in Section 5.2, is done off-line and results in the system MDP. When an adaptation decision has to be made at run time, a stochastic model of the predicted evolution of the environment over the decision horizon is generated and encoded as a DTMC, as described in Section 3.5. To make the decision, an algorithm based on the principles of stochastic dynamic programming computes the solution to the adaptation decision problem, weaving the environment DTMC into the system MDP as the solution is computed.

The rest of this chapter is organized as follows. In Section 5.1, we present the mathematical formulation of the adaptation decision. Section 5.2 explains how the reachability predicates that define the system MDP are computed off-line using formal specification and analysis. Section 5.3 presents the algorithm that implements the mathematical optimization problem defined in Section 5.1, which is used to make adaptation decisions at run time.

5.1 Adaptation Decision

As outlined in Section 3.2, the adaptation decision is formulated as a discrete-time sequential decision problem with finite horizon, and its solution determines what adaptation tactics, if any, should be started at the current time to maximize the aggregate utility the system will provide over the decision horizon. A new adaptation decision is made at regular intervals of length τ, and each decision itself is the solution of a discrete-time finite horizon decision problem, in which time is discretized into intervals of length τ, with a horizon of H intervals.

In discrete-time MDPs there is no built-in consideration of the time actions and state transitions take; therefore, we have to model the evolution of the system state in a way that considers the latency of adaptation tactics. Furthermore, since tactics with different latency can be executed concurrently and not even start at the same time, it is not possible to combine multiple tactics into a single MDP action, since each of these tactics could complete, and thus cause a state transition in the MDP, at different times.[2] In order to handle this, we avoid mapping adaptation tactics directly to MDP actions, and focus instead on state transitions, which, although triggered by adaptation tactics, can be due to one or more of them. Time is considered at the granularity of the decision period τ, and the latency of tactics is approximated to multiples of τ. However, the model must also deal with state changes that take a very short amount of time compared to τ (e.g., a fast tactic), which cannot be appropriately modeled with a transition in the MDP that takes an interval of length τ to complete.

In order to account for both fast tactics and tactics whose latency is not negligible, the evolution of the self-adaptive system in this decision problem is modeled considering two kinds of configuration changes or transitions due to adaptation: *immediate*, and *delayed*. Immediate transitions are the result of either the execution of tactics with very low latency (e.g., changing the dimmer value), or the start of a tactic with latency (e.g., adding a server). In the latter case, the transition is immediate because the target state is a configuration in which a new server is being added, but the addition has not been completed yet. Delayed configuration changes are due to adaptation as well, but also require the passing of time for the transition to happen. This is the case, for example, with the addition of a new server, transitioning from a state in which the server is being added to a state in which the addition has completed.

Any solution to the adaptation decision problem will follow the same pattern of immediate and delayed transitions as shown in Figure 5.2 for $H = 3$. The interval $t = 1$ corresponds to the interval of length τ starting at the current time, interval $t = 2$ starts τ later, and so on. To simplify the presentation, we start with two assumptions, which will be relaxed later: (i) the evolution of the environment over the decision horizon is known deterministically; and (ii) each tactic is either instantaneous or its latency is approximately τ. The state of the environment in interval t is e_t.[3] State c_0 represents the configuration of the system when the adaptation decision is being made. At that point, an immediate transition takes the system from c_0 to c_1 right before the first interval starts. The negligible time that this transition takes is denoted as ϵ, and it is shown disproportionately large in the figure so that it can be drawn. The passage of time causes the configuration to change from c_1 to c_1'. For example, if c_1 is a configuration in which the addition of a server has been started, c_1' is one with the server addition completed. After that, another immediate transition resulting in c_2 takes place, then the second interval starts, and so on. For the purpose of considering the utility accumulated over the decision horizon, the inter-

[2] Although there are MDP variants that consider time, such as continuous-time MDP [59], and timed MDP [78], those models associate timing properties with actions, which can be taken one at a time. Therefore, they cannot be used directly to model the timing aspects of concurrent adaptation tactics as is needed in our decision problem.

[3] In general, the state of the environment can change during a decision interval. However, for the abstraction in this decision problem, we do not consider state changes in the environment and the system within a decision interval. Therefore, a metric representative of the state of the environment throughout the interval is used (e.g., the average request arrival rate).

mediate configuration that precedes each immediate transition is ignored, and the utility accrued in interval t is $U(c_t, e_t)$.

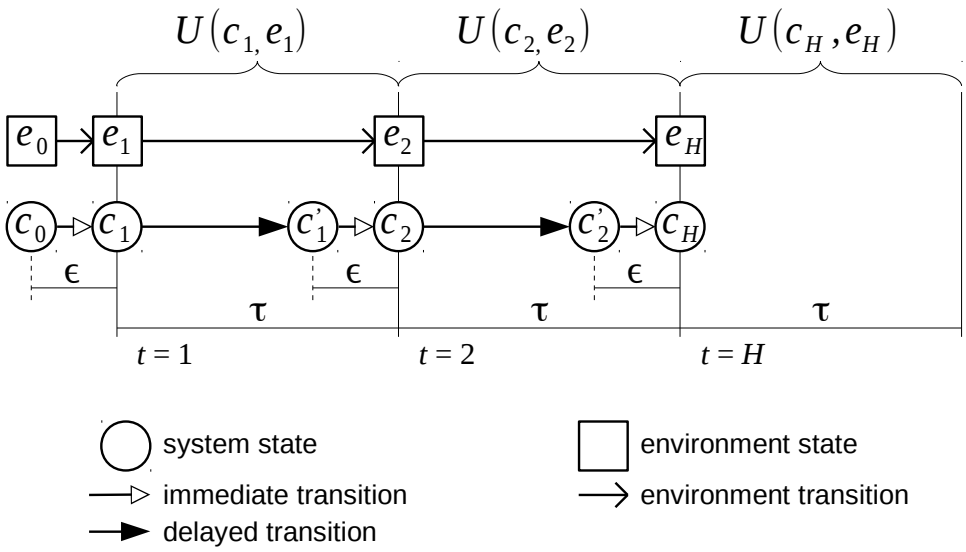


Figure 5.2: Pattern of adaptation transitions in adaptation decision solution.

In general decision problems, the solution is found by considering all the actions that are applicable in each state, and the result is a policy that maps states to actions. However, in our setting there are two reasons why finding and expressing the solution directly in terms of actions is not practical. First, our approach supports concurrent execution of tactics, which means that more than one tactic (or action) can be started simultaneously, resulting in a single transition to a configuration with the combined effect of the tactics. Second, there can be tactics with latency longer than the decision interval, which means that once the tactic has started, it is possible to have transitions that are exclusively due to the passage of time (e.g., transitioning from a state in which 2τ remain to complete the tactic, to one in which τ remains).[4]

Instead of dealing directly with actions, we use predicates over pairs of states that indicate whether configuration c' can be reached from configuration c. These reachability predicates are:

- $R^I(c, c')$, which is true if configuration c' can be reached with an immediate transition from c with the use of none, one or more tactics; and
- $R^D(c, c')$, which is true if configuration c' can be reached with a delayed transition from c in one time interval.

[4]Even though in the models used by PLA-PMC the decision is made at the level of adaptation tactics, the complexity of the decision problem is hidden by PRISM. The MDP that PRISM builds, and then solves, ends up having multiple states to represent the different combinations of tactics that could be started simultaneously, and the different ways in which they could be phased in time. Furthermore, the generated MDP has a varying number of transitions that happen without advancing the clock, to model the parallel composition of the tactic modules. Here, we avoid modeling the problem in terms of adaptation actions so that transitions in the resulting MDP happen following a predefined pattern in time.

A third helper predicate, used for a more compact notation, is true if c' can be reached from c in one time interval through a delayed transition followed by an immediate transition:

$$R^T(c, c') \equiv \exists c'' : R^D(c, c'') \wedge R^I(c'', c')$$

Defining these predicates is not trivial due to the possible interactions between different tactics, which requires exploring all the possible combinations of tactics, including all the different ways in which they can be phased in time, given that some of them have latency. In our approach, we use formal models and an analysis tool to compute these predicates off-line (as explained in Section 5.2), reducing the burden on the run-time decision algorithm.

These predicates define the transition matrix for the system portion of the adaptation MDP. Therefore, a solution like the one shown in Figure 5.2 is feasible only if the following holds

$$R^I(c_0, c_1) \wedge R^T(c_t, c_{t+1}), \forall t = 1, \ldots, H-1$$

To find the solution, let us refer to the set of all system configurations as C. This set contains all the configurations that are unique with respect to the properties relevant to computing the utility function, but not emergent or derived properties. In RUBiS, for example, these properties include the number of active servers, and the dimmer value (but not response time, for example, because it is an emergent property). Later on, this set will be extended to capture the state of running tactics in the system configuration. Let us also define sets of configurations that can be reached from a given configuration using different kinds of transitions:

$$C^T(c) = \left\{c' \in C : R^T(c, c')\right\}$$
$$C^I(c) = \left\{c' \in C : R^I(c, c')\right\}$$

With the assumption of a deterministic environment (i.e., the state of the environment is a function of time), the solution C^* to the adaptation decision problem can be found using *dynamic programming* [98]. Dynamic programming is an approach to solve complex decision problems that can be divided into smaller problems such that the solution of a subproblem is a partial solution to the complete problem. In the adaptation decision problem, for example, deciding which adaptation tactic to execute at the beginning of the decision horizon so that the utility accumulated over the whole horizon is maximized requires knowing the maximum utility that can be obtained in the remainder of the horizon once the first action has been taken, which is an optimization problem in itself. To solve a problem with dynamic programming, one has to identify base cases that are easily solvable. For a decision problem like ours, the utility that can be obtained in each of the possible states at the end of the horizon does not depend on anything other than the system configuration and the environment state in that interval because there are no further decisions or accumulation of utility considered beyond the end of the horizon. Therefore, it makes sense to consider those as the base cases, working backwards throughout the decision horizon. That is, the value of a configuration $c \in C$ in the last interval of the horizon is the utility that c provides in that interval. The value of a configuration c in interval $t < H$ is the utility that c provides in that interval, plus the maximum value that can be obtained from the configurations reachable from c in interval $t + 1$.

The formulation of the adaptation decision problem with dynamic programming is as follows:

$$v^H(c) = \hat{U}(c, e_H), \quad \forall c \in C \tag{5.1}$$

$$v^t(c) = \hat{U}(c, e_t) + \max_{c' \in C^T(c)} v^{t+1}(c'), \quad \forall c \in C,\ t = H-1, \ldots, 1 \tag{5.2}$$

$$C^* = \arg\max_{c' \in C^I(c_0)} v^1(c') \tag{5.3}$$

where $v^H(c)$ represents the value that each configuration has at the end of the horizon—the base cases—and $v^t(c)$ is the value that will be obtained from time step t onwards if the system is in configuration c at time t and optimal decisions are made in the remainder of the horizon. The configuration that the system should adapt to in order to maximize the utility accumulated over the decision horizon is the one that has the maximum value at time $t = 1$.

The utility function used to solve the adaptation decision problem is the decision utility function $\hat{U}$, which has some differences with respect to U, the one used to measure the utility of the system. The main difference is that the emergent system properties that are relevant for computing the utility function are not measured but estimated. For example, for RUBiS, the response time used here is not the measured response time of the system, since $\hat{U}$ is used to compute the utility that the system *would* attain under a certain configuration and state of the environment. Therefore, emergent properties like these have to be estimated. For example, the response time in RUBiS can be estimated using queuing theory [84], or performance analysis tools, such as layered queuing network solvers [100, 112], or queuing Petri net solvers [85]. In addition, $\hat{U}$ can encode other decision preferences that, despite not being considered by the measurement utility function, are desirable for the system. For instance, in a case in which all the configurations of RUBiS would exceed the response time threshold, the measurement utility function would choose the one with the smallest number of servers, since it would see no benefit in having more. This is not the right decision because removing resources from an overloaded system would cause the backlog of requests to increase at a higher rate, making the recovery of the system in subsequent decisions even more unlikely. If it is desired to avoid such behavior, $\hat{U}$ can be defined to favor the configuration with the most servers in such a case.

The result of $\arg\max$ in (5.3) is actually a set, so we can pick any configuration $c^* \in C^*$. However if $c_0 \in C^*$, we can avoid adapting, since no configuration change would render any improvement. Since the actions in our setting are assumed to be deterministic, given the source and target of a single transition, it is possible to determine the actions that must be taken, as we explain later in Section 5.2.2. Therefore, once c^* is found, the set of tactics that must be started to reach it from c_0 can be determined.

5.1.1 Stochastic Environment

We model the evolution of the environment over the decision horizon as a discrete-time Markov chain (see Section 3.5). The set of environment states is denoted by E, and the probability of transitioning from state e to state e' is given by $p(e'|e)$. The most straightforward way to take into account the stochastic evolution of the environment would be to create the joint MDP of

the system MDP and the environment Markov chain,[5] and then find its solution. However, this would require creating the transition probability matrix over the joint state space $C \times E$, and evaluating many joint states that would never be reachable. Keeping in mind that this would have to be done every time an adaptation decision has to be made in order to incorporate the latest environment predictions, doing this has a couple of drawbacks. First, the full joint MDP would have to be created for every decision so that the latest environment predictions could be incorporated. Second, evaluating the utility for a pair of system configuration and environment state may involve some extensive computation (e.g., invoking a layered queuing network solver to estimate response time), so doing that for unreachable joint states is a waste of resources and time.

To reduce the running time of the adaptation decision, we avoid creating the joint MDP, and instead weave the environment model into the predefined MDP of the system as needed. Referring to Figure 5.2, we can see that the system and the environment make a transition almost simultaneously at the beginning of each interval. For example, at the start of the interval at $t = 1$, the system transitions from c_0 to c_1 (the alternative target configurations are not shown), and the environment transitions from its current state e_0 to e_1 deterministically. On the other hand, with a stochastic environment both the system and the environment have several possible target states at each interval. Figure 5.3 depicts these two kinds of transitions interleaved. First, the system takes a deterministic transition, and then the environment takes a probabilistic transition.

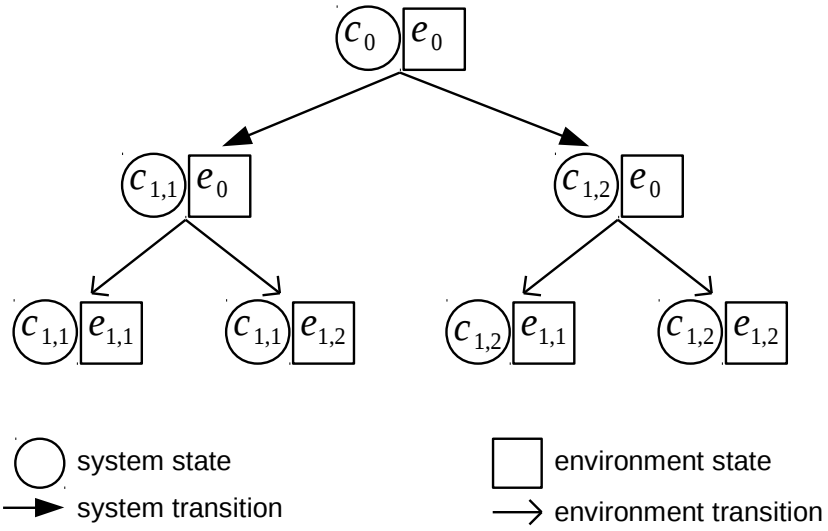


Figure 5.3: System and environment transitions.

Using the principles of stochastic dynamic programming [120], the adaptation decision problem with a stochastic model of the environment can be solved as follows:

[5] A discrete-time Markov chain can be turned into an MDP by assuming there is a single action applicable in every state.

$$v^H(c,e) = \hat{U}(c,e), \quad \forall c \in C, e \in E_H \tag{5.4}$$

$$v^t(c,e) = \hat{U}(c,e) + \max_{c' \in C^T(c)} \sum_{e' \in E_{t+1}} p(e'|e) v^{t+1}(c',e') \quad \forall c \in C, e \in E_t, \tag{5.5}$$

$$t = H-1, \ldots, 1$$

$$C^* = \arg\max_{c' \in C^I(c_0)} \sum_{e' \in E_1} p(e'|e_0) v^1(c',e') \tag{5.6}$$

Note that instead of evaluating all environment states in E for each time interval, only those that are feasible are considered. E_t is the set of environment states feasible in time interval t. When a probability tree is used to model the environment, E_t is the set of nodes of depth t. This greatly reduces the number of calculations of accumulated utility that have to be made, avoiding evaluating system/environment state pairs that are not feasible.

5.1.2 Handling Latency

When a tactic has latency, the adaptation decision has to be able to determine when the tactic is going to complete, so that its effect on the system configuration can be accounted for at the right time. In addition, while a tactic executes, it can prevent other incompatible tactics from starting, which affects the decision. So far, we assumed that if a tactic had latency, it was roughly equal to one time interval, but in reality it can be of any length, and span multiple intervals. Since we use a Markov model, there is no history in the model to allow us to directly keep track of the progress of the tactic. Consequently, we have to extend the state space in the model to keep track of the progress of tactics with latency. However, given that decisions are made at regular intervals over the decision horizon, it is only necessary to keep track of the progress of the tactic at the granularity of the time interval, as in PLA-PMC.

For RUBiS, for example, the configuration of the system with the properties relevant for computing the utility function can be captured by a tuple (s,d), where s is the number of active servers, and d is the discretized dimmer value. In order to keep track of the progress of the tactic to add a server, we extend the configuration with another component, so that the full configuration tuple is (s,d,p_{add}), where $p_{add} \in \left\{0, \ldots, \lceil \frac{\lambda}{\tau} \rceil\right\}$ is the number of time intervals left until the tactic completes, with 0 indicating that the tactic is not being executed.[6] Every delayed transition enabled by R^D decreases the progress tracking component of the configuration tuple for all the executing tactics. That is, if there are N tactics with latency, and the configuration tuple has components $p_1, \ldots, p_N$ to represent the progress of the tactics with latency, then

$$\forall c = (\ldots, p_1, \ldots, p_N), c' = (\ldots, p'_1, \ldots, p'_N) \in C :$$
$$R^D(c,c') \implies \forall i = 1, \ldots, N : p'_i = \max(0, p_i - 1)$$

For example, the start of the tactic to add a server implies an immediate transition in the model to a configuration with p_{add} equal to its maximum value. This transition is enabled by R^I. It

[6] If a tactic cannot execute concurrently with itself, a single value can track its progress. If multiple instances of a tactic can be executed concurrently, one progress component per instance is needed.

is then followed by a sequence of delayed transitions enabled by R^D that decrease the value of p_{add} until it reaches 0, and when that happens, the number of servers in the configuration tuple is increased.

5.2 Computing Reachability Predicates

The predicates R^I and R^D determine which system configurations can be reached from other configurations through adaptation; that is, they specify which transitions are feasible in the system MDP. Defining these predicates by extension, or trying to express them in propositional logic can be a daunting and error prone task due to all the possible combinations of tactics, all their possible phasings (i.e., how their executions overlap in time), and all the possible system states that must be taken into account. Instead, we use formal models and analysis to compute the reachability predicates. Specifically, we use Alloy [76] to formally specify system configurations and adaptation tactics, and to compute the reachability predicates.[7] Alloy is a language based on first-order logic that allows modeling structures—known as signatures—and relationships between them in the form of constraints. Alloy is a declarative language, and, in contrast to imperative languages, only the effect of operations—tactics in our case—on the model must be specified, but not how the operations work. The Alloy analyzer is used to find structures that satisfy the model. By deferring combining the system and environment states until run time in our approach, these predicates are independent of the environment state, and thus can be computed off-line. Hence, the overhead of using formal methods to compute the predicates is not incurred at run time.

The support for concurrent tactics in the adaptation decision is handled by these predicates. The adaptation problem formulation (5.4)-(5.6) is agnostic with regard to concurrent tactic execution, since it only cares about state reachability, regardless of whether that requires concurrent tactics or not. On the other hand, to correctly determine whether a configuration can be reached from another configuration when computing the predicates, it is necessary to consider whether tactics can be executed concurrently or not. To that end, we rely on a *compatibility predicate* for each tactic that indicates whether it can be run, considering the other tactics that are executing. This is the same approach used in PLA-PMC to express in the model whether a tactic can be run or not (see Section 4.5.2). According to the concurrency model described in Section 3.4, we require that two tactics are allowed to execute concurrently only if they affect disjoint subsets of the properties of the configuration state. From the formal model perspective, this requirement makes the tactics serializable, since the state resulting from the serial application of any combination of compatible tactics would be the same as if they were applied in parallel.

5.2.1 Delayed Reachability

To compute R^D, we use Alloy to find all the pairs $(c, c') \in \mathrm{CP} \times \mathrm{CP}$, such that $R^D(c, c')$, where CP is the configuration space extended with tactic progress. To achieve that, we introduce first other necessary pieces of the model, starting with the definition of the configuration space for the

[7] Our initial use of Alloy to compute the reachability predicates was published in [23], and later improved in [109].

system. Listing 5.1 shows the declarations that define the configuration space for RUBiS. The sets S, and D represent the different numbers of active servers, and dimmer levels, respectively. The elements of these sets are not numbers, but rather abstract elements. However, lines 1-2 specify that these are ordered sets. Thus, we can refer to their first and last elements, for example, with SO/first and SO/last. Also, we can get the successor and predecessor of an element e with SO/next[e] and SO/prev[e]. The signature C defines the set of all possible configurations, each having a number of active servers s, and a dimmer level d. In the Alloy model, we distinguish between plain system configurations, C, and configurations extended with tactic progress, CP. The reason we do this is for the model to be more modular, keeping concerns separated, so that it is easier to generate the model in the Alloy language for different systems, and/or different sets of tactics. For example, new tactics can be added to the model without modifying C. Note, however, that when the adaptation decision problem is solved, CP in this model corresponds to C in the formulation presented in Section 5.1.

```
1  open util/ordering[S] as SO
2  open util/ordering[D] as DO
3  sig S {} // the different number of active servers
4  sig D {} // the different dimmer levels
5
6  // each element of C represents a configuration
7  sig C {
8      s : S, // the number of active servers
9      d : D // dimmer level
10 }
```

Listing 5.1: Alloy model: configurations.

Listing 5.2 shows the elements needed to represent tactic progress. The declaration of the set of all the tactics T as **abstract** in line 3 indicates that all of its elements must be elements of one of the signatures that extends it. For each of the tactics with no latency, a singleton set extending T is declared (line 4). Since it is necessary to tell tactics with latency apart, the abstract subset LT is declared (line 5), and a singleton subset of it is declared for each of the tactics with latency (line 6, only AddServer in our example). The different levels of progress of each tactic are represented by the elements of an ordered set. For example, TPAS (lines 1 and 8) contains the levels of progress of the tactic to add a server, with TPASO/first indicating that the tactic has just started, TPASO/last indicating that the tactic execution has completed, and the elements in between representing intermediate progress. The ordered sets that represent the levels of progress of tactics are subsets of an abstract set TP. The signature CP extends C, adding a mapping p from tactics with latency, LT, to the tactic progress, TP (line 12). The facts in lines 14-15 constrain p to be a function over LT. Additionally, we require that the function maps each tactic to a progress in its corresponding class (line 16). Lastly, the fact in line 19 requires that all elements of CP are different. The predicate equals (lines 21-23) is true if the two configurations are equal with respect to the fields of C. The predicate equalsExcept (lines 25-27) is similar, except that it supports excluding one field of C from the comparison.

Note that even though we are using the RUBiS example to describe the models, they can be easily adapted to model other systems, since special effort has been devoted to making as many

```
1  open util/ordering[TPAS] as TPASO // tactic progress for adding server
2
3  abstract sig T {} // all tactics
4  one sig IncDimmer, DecDimmer, RemoveServer extends T {} // tactics with no latency
5  abstract sig LT extends T {} // tactics with latency
6  one sig AddServer extends LT {} // tactic with latency
7  abstract sig TP {} // tactic progress
8  sig TPAS extends TP {} // one sig for each tactic with latency
9
10 // configuration extended with the progress of each tactic with latency
11 sig CP extends C {
12     p: LT -> TP
13 } {
14     ~p.p in iden // p maps each tactic to at most one progress
15     p.univ = LT // every tactic in LT has a mapping in p (p.univ is the domain of p)
16     p[AddServer] in TPAS // restrict each tactic to its own progress class
17 }
18
19 fact uniqueConfigs { all disj c1, c2 : CP | !equals[c1, c2] or c1.p != c2.p}
20
21 pred equals[c, c2 : C] {
22     all f : C$.fields | c.(f.value) = c2.(f.value)
23 }
24
25 pred equalsExcept[c, c2 : C, ef : univ] {
26     all f : C$.fields | f=ef or c.(f.value) = c2.(f.value)
27 }
```

Listing 5.2: Alloy model: configurations extended with tactic progress.

parts of the model as possible be system-independent. For example, in Listing 5.2, lines 19-27 do not depend at all on the specific fields that define the configuration space of a system. In addition, it is straightforward to extend or modify the model for other systems. For instance, if another tactic with latency were added, lines equivalent to lines 1, 8, and 16 would need to be added.

Now that we have all the basic elements in the model, we can present the predicates that determine the reachability in one time interval. For each tactic with latency, a predicate like addServerTacticProgress, shown in Listing 5.3, is needed. This predicate is true if according to the tactic, the post-state c' can be reached in one time interval from the pre-state c. If the tactic is running (i.e., its progress is not in the last state), the predicate requires that in the post-state, the progress of the tactic is the next one (line 3). If it reaches the last level of progress, then the configuration has one more server in the post-state (line 5), reflecting the effect of the completion of the tactic. In addition, it is as important to ensure that if the tactic is not running, it stays in that state (line 10), and does not have an effect (line 11). We also need to require that nothing else changes (lines 15-16). Line 15 requires that every field of C other than s, the field affected by this tactic, stays the same, whereas line 16 ensures that the progress of all the other tactics has not been changed in the post-state.

Finally, the predicates for the progress of each tactic with latency have to be put together to define progress, a predicate equivalent to R^D (line 1-3, Listing 5.4). When the system has more than one tactic with latency, their predicates have to be composed to reflect the effect that all of them would have on the state. All the progress predicates are serializable, because they

```
pred addServerTacticProgress[c, c' : CP] {
    c.p[AddServer] != TPASO/last implies { // tactic is running
        c'.p[AddServer] = TPASO/next[c.p[AddServer]]
        c'.p[AddServer] = TPASO/last implies {
            c'.s = SO/next[c.s] // tactic effect
        } else {
            c'.s = c.s // no finished yet, maintain state
        }
    } else { // tactic is not running
        c'.p[AddServer] = TPASO/last // stay in not running state
        c'.s = c.s
    }

    // nothing else changes other than s and the progress of this tactic
    equalsExcept[c, c', C$s]
    (LT − AddServer) <: c.p in c'.p
}
```

Listing 5.3: Alloy model: tactic progress predicate

either correspond to tactics that can execute concurrently, for which serializability is required; or they correspond to incompatible tactics. In the latter case, only one of them could be in a state in which it can affect the configuration, whereas the rest would have no effect, making them serializable as well. Therefore, all of the progress predicates can be combined using sequential composition [66]. In the example of Listing 5.4, we assume there is another tactic with latency, rebootServerTacticProgress, and show an example of sequential composition.

```
pred progress[c, c' : CP] { // is c' reachable from config c in one evaluation period?
    some tc : CP | addServerTacticProgress[c, tc] and rebootServerTacticProgress[tc, c']
}

sig Result {
    c, c' : CP
} {
    progress[c, c']
}

// this reduces the number of unused configurations
// each cp in CP is either in a pair in a result, or an intermediate one needed for that pair
fact reduceUsedConfigs {
    all cp : CP | {some r : Result | r.c = cp or r.c' = cp
            or (addServerTacticProgress[r.c, cp] and rebootServerTacticProgress[cp, r.c'])
        }
}

pred show { }
```

Listing 5.4: Alloy model: delayed transition predicate.

To compute R^D with Alloy, we define a signature Result that represents a pair of configurations for which progress holds (Listing 5.4, lines 5-9). Alloy requires that a scope (i.e., cardinality, either exact or as a bound) be provided for the different sets in the model. In general, the scope can be determined by the number of values that each of the fields that define the system state space can take. In the case of RUBiS, the scope can be computed based on the maximum

number of servers for the system, and the number of dimmer levels. In addition, the scope for the sets that define the number of progress levels for the tactics with latency has to be specified. For each tactic with latency, the scope of its corresponding progress set is the number of time intervals needed for the execution of the tactic plus one, to denote the state in which the tactic is not running. In addition, we specify a scope of 1 for the signature Result, so that in a valid instance of the model, there is only one pair (c, c') that satisfies $R^D(c, c')$. The following command shows how the Alloy analyzer is run to obtain all model instances, and thus all the pairs, that satisfy $R^D(c, c')$.

run show **for exactly** 3 S, **exactly** 3 TPAS, **exactly** 5 D, **exactly** 1 Result, 3 C, 3 CP

The **run** command finds all the instances that satisfy the model and the predicate specified as its first argument, which in this case always holds. The remainder of the command, the **for** clause, specifies the scope for the analysis. In this case, it is stating that the maximum number of servers is 3, the latency for the tactic to add a server is 2 (+1 in the scope), and there are 5 dimmer levels. The scope after that indicates that each instance should have exactly one result. The bound for C and CP depends on the number of tactics with latency used in the model. If we look at the sequential composition of these tactics in line 2 of Listing 5.4, we observe that it requires up to three distinct elements of CP; that is, one more than the number of tactics with latency. And, since each distinct element of CP could be associated with a different element of C, we need at most the same number of elements for the latter. In this example, the value for the upper bound of their scope is 3, because there are two tactics with latency. When run with this command, Alloy generates all the instances that satisfy the model. The output is read using Alloy's API, and used to generate a simple encoding of R^D as a lookup table suitable for use at run time when a decision has to be made.

Even though we are only interested in the instance of the Result signature, which corresponds to a pair in R^D, Alloy can generate many instances that have the same instance of Result if they vary in something else that is valid in the model. For example, if the progress predicate can be satisfied with fewer elements of CP than were specified in the scope, then there are free elements of CP that can take any value. In such a case, Alloy will generate all possible combinations of them. To reduce the number of such instances that result in multiple solutions with the same pair for R^D, we include the fact in lines 13-17, which requires that elements of CP are used in the progress predicate. This considerably reduces the amount of time needed to compute all the solutions.

5.2.2 Immediate Reachability

In order to compute R^I, we define a predicate for each tactic that checks whether the tactic is applicable, and if so, it reflects the effect of the tactic on the post-state. However, we cannot simply compose them sequentially, as we do for R^D, because it is necessary to consider cases in which a tactic is not used even if its applicable. The approach we take to deal with this problem is to model a trace of configuration states such that each element of the trace is related to its predecessor by either the application of a tactic, or the identity relation. Even though this

trace consists of a sequence of tactics, it represents the simultaneous start of those tactics, whose combined effect on the system state is the result of their sequential composition. Using the Alloy analyzer we can find all possible traces that satisfy this model, and the set of all pairs formed by the first and last state of each trace is the relation R^I.

Listing 5.5 shows a portion of the model to compute R^I defining the model of a trace. In addition to computing R^I, we also need to compute for each pair in that relation the (possibly empty) set of tactics that have to be started for the immediate transition represented by the pair to hold. This is used to determine which tactics have to be started once the solution to (5.6) is found. To accomplish that, the elements of the trace have not only the configuration state, but also a set of tactics that have been started to arrive at that particular state in the trace (lines 3-6). The fact traces defines what a valid trace is. Line 9 states that at the beginning of the trace no tactic has been started at this time. The remainder of the fact specifies that every trace element is the same as its predecessor, or is related to its predecessor by one of the predicates for the tactics. These predicates specify tactic applicability and how the state is affected when the tactic is started, which largely depends on whether the tactic has latency or not.

```
1  open util/ordering[TraceElement] as Trace
2
3  sig TraceElement {
4      cp : CP,
5      starts : set T // tactics started
6  }
7
8  fact traces {
9      let fst = Trace/first | fst.starts = none
10     all e : TraceElement − last | let e' = next[e] | {
11             equals[e, e'] and equals[e', Trace/last]
12         } or addServerTacticStart[e, e'] or removeServerTactic[e, e'] or decDimmerTactic[e, e'] or incDimmerTactic[e, e']
13 }
```

Listing 5.5: Alloy model: traces for immediate reachability.

For each tactic with latency there is a predicate that models the start (but not the effect, which is delayed) of the tactic. An example is shown in Listing 5.6. These predicates relate trace elements instead of configuration states, since they have to maintain the state of starts in the trace elements, and use it to determine tactic compatibility. The predicate addServerTacticStart first checks that the tactic is compatible (line 7) using the predicate addServerCompatible. This predicate (lines 1-4) checks that this tactic has not already been started in the trace, and that it is compatible. In this case, this tactic is not compatible with the tactic RemoveServer, so it ensures that the latter has not been started. Also, the predicate holds only if the tactic is applicable, which in this case means that the number of servers has not reached its maximum, or last value (line 8). In the post-state, the tactic is added to the set of tactics started (line 10), and the progress of the tactic is set to the first level (line 12). In addition, the predicate ensures that nothing else changes (lines 15-16).

For instantaneous tactics, the predicate follows the same pattern, except that the effect of the tactic on the post-state is included (e.g., an increase of the dimmer value), and no tactic progress state is affected. Listing 5.7 shows the predicates for the tactic to increase the dimmer. The

```
1  pred addServerCompatible[e : TraceElement] {
2      e.cp.p[AddServer] = TPASO/last
3      !(RemoveServer in e.starts)
4  }
5
6  pred addServerTacticStart[e, e' : TraceElement] {
7      addServerCompatible[e]
8      e.cp.s != SO/last
9
10     e'.starts = e.starts + AddServer
11     let c = e.cp, c'=e'.cp | {
12         c'.p[AddServer] = TPASO/first
13
14         // nothing else changes
15         equals[c, c']
16         (LT − AddServer) <: c.p in c'.p
17     }
18 }
```

Listing 5.6: Alloy model: predicates for tactic start.

compatibility predicate holds if the tactic has not already been used in this trace, and if the tactic DecDimmer, with which it is not compatible, has not been started. In line 12, the predicate incDimmerTactic reflects the effect of the tactic on the system configuration in the post-state. In this case, it reflects an increase in the dimmer setting.

```
1  pred incDimmerCompatible[e : TraceElement] {
2      !(IncDimmer in e.starts)
3      !(DecDimmer in e.starts)
4  }
5
6  pred incDimmerTactic[e, e' : TraceElement] {
7      incDimmerCompatible[e]
8      e.cp.d != dimmer/last
9
10     e'.starts = e.starts + IncDimmer
11     let c = e.cp, c'=e'.cp | {
12         c'.d = c.d.next
13
14         // nothing else changes
15         equalsExcept[c, c', C$d]
16         c'.p = c.p
17     }
18 }
```

Listing 5.7: Alloy model: predicates for instantaneous tactic.

Similarly to what is done for computing R^D, Alloy is used to find all the possible instances that satisfy the model. In this case, the command used to run the Alloy analyzer is the following.

run show **for exactly** 3 S, **exactly** 3 TAPS, 5 D, 3 C, 3 CP, 3 TraceElement

The scope for the properties of the system configuration and tactic latency are the same as ex-

plained before. However, the scope for C, CP, and TraceElement is one more than the maximum number of tactics that could be started concurrently.[8]

The model presented thus far would generate different solutions for different permutations of tactics in the trace. Since the order of the tactics within a trace does not matter, because in the end they will be used as a set of tactics to be started concurrently by the adaptation manager, it is a waste of time to generate those permutations. To avoid that, we impose an arbitrary order on the tactics requiring that they follow that order if they appear in a trace. This does not prevent any feasible trace from being generated, because if two tactics are compatible, the order in which they are applied does not matter. If they are not compatible, only one of them can appear in the trace. Notwithstanding, the one that is first in the order imposed does not trump the other because a trace that has latter but not the former is also valid. Listing 5.8 shows a modified version of the trace model to accomplish this. Lines 1-8 define an arbitrary order for the tactics. The predicate validOrder is used in line 21 to ensure that the new tactic started by an element does not have any of the tactics previously started by the trace as a successor.

```
1  open util/ordering[T] as TO
2
3  fact tacticOrdering {
4      TO/first = AddServer
5      AddServer.next = RemoveServer
6      RemoveServer.next = IncDimmer
7      IncDimmer.next = DecDimmer
8  }
9
10 // holds if the tactics started in e are not successors of t
11 pred validOrder[t : T, e : TraceElement] {
12     all s : e.starts | !(s in t.nexts)
13 }
14
15 fact traces {
16     let fst = Trace/first | fst.starts = none
17     all e : TraceElement − last | let e' = next[e] | {
18         equals[e, e'] and equals[e', Trace/last]
19     } or (
20         (addServerTacticStart[e, e'] or removeServerTactic[e, e'] or decDimmerTactic[e, e'] or incDimmerTactic[e, e']) and
21             (let s = e'.starts − e.starts | all t : s | validOrder[t, e]))
22 }
```

Listing 5.8: Alloy model: avoiding unnecessary permutations in traces.

When Alloy is run to analyze the model presented in this section, it finds all the traces that satisfy the model. Through its API we can iterate over all the traces it produces. For each trace, the first and last element correspond to pairs of R^I. In addition to encoding R^I as a simple lookup table for its use at run time, a map that associates pairs in R^I to their corresponding set of tactic starts is also constructed. This map can be used to determine the tactics that have to be started to realize the immediate transition from the current configuration to the next optimal configuration found with (5.6).[9]

[8]The scope could be the total number of tactics plus one, which is easier to determine than the maximum number of tactics that could be started concurrently. The result would be the same, except that this off-line analysis would be slower because it would generate duplicate solutions that differ only in free elements that do not affect the result.

[9]The complete Alloy models for the two systems used for the validation of the thesis are included in Appendix C

5.3 Algorithm

The algorithm in Figure 5.4 implements the mathematical formulation in (5.4)-(5.6). The function DECIDE takes the current system configuration c_0 and the model of the environment E, and returns the configuration c_{best}, which is the configuration reachable through an immediate adaptation that maximizes the expected value to be accumulated over the decision horizon.

To avoid duplication, (5.4) and (5.5) are implemented by the same loop in lines 2-10. Basically, this loop iterates over the decision horizon starting from the end, and has nested loops to iterate over system configurations and environment states. Inside all these nested loops, a call to the function EXPECTEDVALUE computes the expected value for a given pair of system and environment states. The parameter V has all the values v computed up to that point. The algorithm has one optimization over the mathematical formulation. Line 4 avoids computing the expected utility for configurations that are not reachable from the current configuration, when $t = 1$, because at that point it is simple to determine whether a configuration is reachable through the use of R^I.

The function EXPECTEDVALUE computes the expected utility that can be accumulated from time t until the end of the decision horizon if the system is in configuration c at time t, given that the environment is e. For $t = H$, this expected value is just $\hat{U}(c, e)$, since that is the end of the decision horizon. For other values of t, however, we must add the maximum expected value that can be accumulated afterwards, given the stochastic environment behavior. This is accomplished by lines 34-43, which implement the second term of (5.5).

Finally, lines 12-27 implement (5.6), finding the configuration c_{best} that can be reached immediately and maximizes the expected utility to be accumulated over the decision horizon. Note that the algorithm gives preference to the current system configuration, c_0, if it happens to be one of the optimal solutions, avoiding unnecessary adaptations.

5.4 Speedup Evaluation

To compare the running time of the run-time decision of PLA-SDP with PLA-PMC, we measured the adaptation decision time with both solutions in a simulation of RUBiS (see Section 8.1.1 for more details).[10] Since the adaptation decision time increases with the size of the state space, the maximum number of servers used in RUBiS was varied between 10 and 100, resulting in increasing sizes of the state space. The results are shown in the box plot in Figure 5.5, with each box summarizing the statistics of the multiple decisions made in a run of the simulation, with the median, 1st and 3rd quartiles represented by the box, and the range by the bar.[11] These results show that the adaptation decisions with PLA-SDP are much faster than with PLA-PMC, with an average speedup of 27.9, and with much less variance. Despite computing the solution to the adaptation decision much faster, PLA-SDP produces exactly the same results as PLA-PMC,

and Appendix D.

[10] The adaptation decision code is exactly the same used with the real system; only the managed system is simulated.

[11] The first decision in each run with PLA-PMC took longer, probably because the model checker had not been cached yet. To avoid these outliers, the first data point for each run was not used for the plot.

```
1: function DECIDE(c_0, E)
2:     for t = H downto 1 do
3:         for all i ∈ C do
4:             if t ≠ 1 ∨ R^I(c_0, i) then
5:                 for all e ∈ E_t do
6:                     v_t(i, e) ← EXPECTEDVALUE(i, t, e, E, V)
7:                 end for
8:             end if
9:         end for
10:    end for
11:
12:    c_best ← c_0
13:    v_0(c_0, e_0) ← −∞
14:    for all j ∈ C do
15:        v' ← 0
16:        if R^I(c_0, j) then
17:            for all e' ∈ E_1 do
18:                v' ← v' + p(e_0, e')v_1(j, e')
19:            end for
20:            if v' >= v_0(c_0, e_0) then
21:                if j = c_0 ∨ v' > v_0(c_0, e_0) then
22:                    v_0(c_0, e_0) ← v'
23:                    c_best ← j
24:                end if
25:            end if
26:        end if
27:    end for
28:    return c_best
29: end function

30: function EXPECTEDVALUE(c, t, e, E, V)
31:    if t = H then
32:        v ← 0
33:    else
34:        v ← −∞
35:        for all j ∈ C do
36:            v' ← 0
37:            if R^T(c, j)) then
38:                for all e' ∈ E_{t+1} do
39:                    v' ← v' + p(e, e')v_{t+1}(j, e')
40:                end for
41:                v ← max(v, v')
42:            end if
43:        end for
44:    end if
45:    v ← Û(c, e) + v
46:    return v
47: end function
```

Figure 5.4: PLA-SDP adaptation decision algorithm.

which was confirmed in these runs and is also shown in Chapter 8. The decision speed of the approach is further analyzed in Chapter 8 as well.

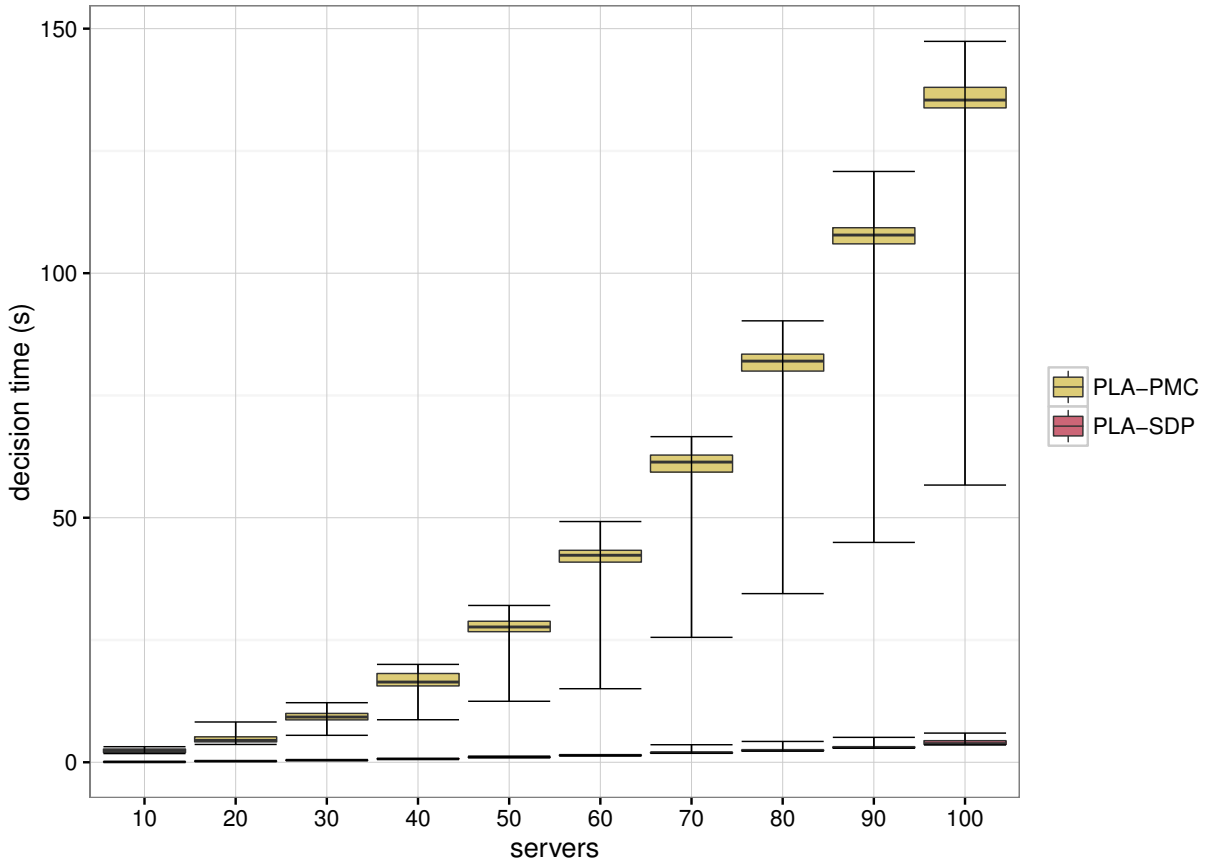


Figure 5.5: Adaptation decision times with PLA-PMC and PLA-SDP.

5.5 Summary

In this chapter we presented PLA-SDP, an approach for proactive latency-aware adaptation that makes adaptation decisions faster than PLA-PMC while producing the same results. This is achieved by keeping the system and environment components of the adaptation MDP separate as long as possible, combining them efficiently when an adaptation decision has to be made. The system MDP is difficult to build due to the large number of possible combinations of tactics, system states, and the (in)compatibility of certain tactics. However, because of this separation, the system MDP does not require information about the environment, which is known only at run time. Therefore, it can be built off-line using formal specification in Alloy. The probabilistic model of the environment is updated at run time, and is combined with the system MDP as the adaptation decision is solved using stochastic dynamic programming. Our experimental results show that this approach is more than twenty times faster than using probabilistic model checking at run time to make adaptation decisions, while preserving the same effectiveness. Another

advantage of PLA-SDP over PLA-PMC is that it is not constrained to a particular language (PRISM, in the case of PLA-PMC) to implement the decision utility function $\hat{U}$. This is because the algorithm in Figure 5.4 can be implemented in any general purpose language,[12] and thus allows the decision utility function to be implemented with the full flexibility of the language, even invoking third party tools such as performance prediction tools.

[12]The results presented in this chapter were obtained with the implementation of PLA-SDP written in C++.

Chapter 6

Support for Alternative Notions of Utility

In previous chapters, the adaptation goal has been presented as the maximization of utility accumulated over the execution of the system, and approximated by only considering the decision horizon. Even though that kind of goal is sufficient for many kinds of self-adaptive systems, there are others with different adaptation goals that cannot be coerced into the maximization of a sum. For example, if reward (or utility) can only be gained as long as a constraint has always been satisfied, whether it can be collected or not in an individual period depends on what has happened in previous periods. Even though this seems trivial as a rule for accounting rewards, it is not that simple for making adaptation decisions. For example, in PLA-SDP, the adaptation decision MDP is solved by backward induction, evaluating states in reverse chronological order. Therefore, either the algorithm requires a modification to deal with this, or the representation of state has to be expanded to include the necessary history.

In this chapter we present an approach to support alternative notions of utility, allowing us to express adaptation goals as a combination of how reward is accumulated and a requirement on constraint satisfaction. The different combinations allow us to implement a variety of adaptation goals with an extension to PLA-SDP. One of the possible combinations is used for one of the case studies described in Chapter 8, and other brief examples illustrating how other combinations can be used are provided in this chapter.

6.1 Adaptation Goal Composition

We define the adaptation goal as a reward maximization problem subject to keeping the probability of satisfying a constraint above a lower bound. The basic building blocks used to formulate the goal are the following functions, which determine the reward gain and constraint satisfaction during a single decision period with system configuration c in environment e:

- $g(c, e)$ is the reward that the system gains
- $s(c, e)$ is the probability of satisfying a constraint[1]

For example, $g(c, e)$ could be the utility RUBiS attains by serving requests, and $s(c, e)$ could be

[1]For cases in which constraint satisfaction can be determined with certainty, the range of s can be $\{0, 1\}$. The approaches in this chapter can be used without modification.

the probability of the system being available (i.e., the probability of satisfying the constraint of not being down).

These functions refer to individual decision periods with no relation to what happens in other periods. However, the adaptation goal must be formulated over the whole decision horizon. Let us start with how reward is gained over the decision horizon, although that can itself be modified by the constraint. We define three reward gain (RG) forms relative to the satisfaction of the constraint, which are summarized in Table 6.1. RG1 is the reward accumulation over the decision horizon regardless of the constraint satisfaction, which is the adaptation goal we have used as an example in previous chapters. In the other two forms, reward gain depends on the satisfaction of the constraint. In RG2, the reward is only gained in a given period if the constraint is satisfied in the period. Since $s(c, e)$ is a probability, RG2 maximizes the expected reward gained over the horizon. RG3 represents the case in which the reward is gained in a period as long as the constraint has been satisfied up to the current period (inclusive) within the decision horizon. The product in RG3 represents the probability of having satisfied the constraint in all the periods up to period t, and the whole equation is the expected reward gained over the decision horizon.

Table 6.1: How reward is gained, relative to the constraint satisfaction.

When reward is gained	**maximize** $_{c_1,\ldots,c_H}$
RG1: *regardless of* the constraint satisfaction	$\sum_{t=1}^{H} g(c_t, e_t)$
RG2: *only when* the constraint is satisfied	$\sum_{t=1}^{H} s(c_t, e_t) g(c_t, e_t)$
RG3: *as long as* the constraint has been and is satisfied	$\sum_{t=1}^{H} \left(\prod_{i=1}^{t} s(c_i, e_i) \right) g(c_t, e_t)$

In addition to how reward is gained, it is also possible to impose a requirement on the probability of satisfying the constraint, although this is optional. Table 6.2 shows two different forms for the constraint satisfaction (CS) requirement. Form CS1 requires that the probability of satisfying the constraint be no less than a bound P in each period, whereas CS2 requires that the probability of satisfying the constraint over all periods be no less than P.

Here are a few examples of how these reward gain and constraint satisfaction requirements can be used to formulate different adaptation goals.

Table 6.2: Constraint satisfaction requirements (zero or more).

Bound on the probability of	**subject to**
CS1: satisfying the constraint *in each period*	$\forall t \in \{1, \ldots, H\}\ s(c_t, e_t) \geq P$
CS2: satisfying the constraint *over all periods*	$\prod_{t=1}^{H} s(c_t, e_t) \geq P$

Example 1. A system gets reward only if it is available, and it is desired to keep the probability of being available in each evaluation period over some threshold P. The fact that the system is unavailable in one period does not prevent it from being available afterwards (i.e., it can repair itself). The problem formulation in this case is {RG2, CS1}, with the constraint being *being available*.

Example 2. A robot gets reward as long as it can operate (i.e., its battery has charge), and it is desired to keep the probability of being able to operate over some threshold P. If the robot runs out of battery, it cannot recover (e.g., it cannot reach a recharging station). The problem formulation in this case is {RG3, CS2}, with the constraint being *having battery charge*.

Example 3. A drone gets reward only if it is not detected in a segment of the route it is flying. However, being detected in a segment does not mean it cannot get reward in subsequent segments. Nevertheless, it is desired to keep the probability of being undetected during the entire mission over some threshold P. The problem formulation in this case is {RG2, CS2}, with the constraint being *not being detected*.

Example 4. A system gets reward in every period even if it is slow to respond. However, it is desired to keep the probability of meeting the response time requirement in each period over some threshold P. The problem formulation in this case is {RG1, CS1}, with the constraint being *meeting the response time requirement*.

Example 5. A drone gets reward as long as it does not crash. The problem formulation in this case is {RG3}, with the constraint being *not crashing*.

6.2 PLA-SDP Formulation Extension

We now show how to extend the formulation of the stochastic dynamic programming adaptation decision to support these types of adaptation goals. In particular, we focus on the most difficult

combination, {RG3, CS2}, since the other forms are either subsumed by this one or require minor modifications, as described later.

With an adaptation goal of the form {RG3, CS2}, the optimization problem is the following:

$$\underset{c_1,\ldots,c_H}{\text{maximize}} \quad \sum_{t=1}^{H} \left(\prod_{i=1}^{t} s(c_i, e_i) \right) g(c_t, e_t) \tag{6.1}$$

$$\text{subject to} \quad \prod_{t=1}^{H} s(c_t, e_t) \geq P \tag{6.2}$$

The key idea to find a policy for the adaptation MDP that is a solution to this optimization problem is to avoid transitions in the policy that would result in a violation of constraint (6.2). However, achieving this requires keeping track of the probability of satisfying the constraint for each partial solution to the problem throughout the backward induction. In the following extended stochastic dynamic programming formulation of the adaptation decision problem, (6.4) and (6.8) keep track of that probability, and (6.5) and (6.11) filter out partial solutions that would violate (6.2).

$$v^H(c, e) = s(c, e)g(c, e), \quad \forall c \in C, e \in E_H \tag{6.3}$$

$$S^H(c, e) = s(c, e), \quad \forall c \in C, e \in E_H \tag{6.4}$$

For $t = H - 1, \ldots, 1$, and $\forall c \in C, e \in E_t$

$$C_t^T(c, e) = \left\{ c' \in C^T(c) \,\middle|\, s(c, e) \sum_{e' \in E_{t+1}} p(e'|e) S^{t+1}(c', e') \geq P \right\} \tag{6.5}$$

if $C_t^T(c, e) \neq \emptyset$

$$\hat{c}^t(c, e) \in \underset{c' \in C_t^T(c,e)}{\arg\max} \sum_{e' \in E_{t+1}} p(e'|e) v^{t+1}(c', e') \tag{6.6}$$

$$v^t(c, e) = s(c, e) \left(g(c, e) + \sum_{e' \in E_{t+1}} p(e'|e) v^{t+1}(\hat{c}^t(c, e), e') \right) \tag{6.7}$$

$$S^t(c, e) = s(c, e) \sum_{e' \in E_{t+1}} p(e'|e) S^{t+1}(\hat{c}^t(c, e), e') \tag{6.8}$$

otherwise

$$v^t(c, e) = -\infty \tag{6.9}$$

$$S^t(c, e) = 0 \tag{6.10}$$

Finally,

$$C_0^I = \left\{ c' \in C^I(c_0) \;\middle|\; \sum_{e' \in E_1} p(e'|e_0) S^1(c', e') \geq P \right\} \tag{6.11}$$

$$C^* = \arg\max_{c' \in C_0^I} \sum_{e' \in E_1} p(e'|e_0) v^1(c', e') \tag{6.12}$$

It is possible that $C_t^T(c, e) = \emptyset$ for some t, which means that there is no policy that satisfies the constraint satisfaction requirement. In that case, assuming that the system cannot stop, it is better to do the best possible. That requires solving the problem again, this time making C_t^T have the configuration that results in the highest probability of satisfying the constraint when no configuration meets that requirement. That is, if $C_t^T(c, e) = \emptyset$ in (6.5), we can fall back to a best-effort approach and compute it as

$$C_t^T(c, e) = \arg\max_{c' \in C^T(c)} s(c, e) \sum_{e' \in E_{t+1}} p(e'|e) S^{t+1}(c', e') \tag{6.13}$$

Analogously, if $C_0^I = \emptyset$ in (6.11), we can compute a best-effort solution with

$$C_0^I = \arg\max_{c' \in C^I(c_0)} \sum_{e' \in E_1} p(e'|e_0) S^1(c', e') \tag{6.14}$$

Note that it is not the same to directly compute the best-effort solution because it will enable paths that would be discarded by the first solution. So, if the first problem has a solution, it may not be the same as the best-effort solution.

The filtering done in (6.5) filters out the target configurations that would violate the probability bound *in expectation* over the environment distribution. This filtering could have other forms. For example, the bound could be applied to the minimum probability of satisfaction, as follows.

$$C_t^T(c, e) = \left\{ c' \in C^T(c) \;\middle|\; s(c, e) \min_{e' \in E_{t+1}} S^{t+1}(c', e') \geq P \right\} \tag{6.15}$$

The other forms of adaptation goals can be formulated based on the ones we have already presented. RG2 can be implemented as RG1 (which is the original formulation presented in Chapter 5) by replacing $g(c, e)$ with $g'(c, e) = s(c, e)g(c, e)$. CS1 can be implemented as CS2, except that the tracking of the probability of satisfying the constraint in (6.8) has to be replaced with

$$S^t(c, e) = s(c, e) \tag{6.16}$$

In addition to these extensions, there are other simple transformations that can be used to deal with other adaptation goals, and do not require changing the implementation of the decision algorithm. Changing the sign of $g(c, e)$ turns the goal into minimization. If instead of maximizing a sum as in RG1, the maximization of a product is required, that can be achieved by taking the logarithm of $g(c, e)$.

6.3 Summary

This chapter presented other forms of utility that combine different forms of reward gain with requirements on probabilistic satisfaction of a constraint, and showed how the PLA-SDP approach can be extended to accommodate these richer goal formulations. As shown in the examples, these cover a variety of adaptation goals for different kinds of self-adaptive systems.

Chapter 7

Strategy-based Proactive Latency-Aware Adaptation

Both PLA-PMC and PLA-SDP solve the adaptation decision problem selecting the adaptation tactics that must be executed, possibly concurrently, to achieve the adaptation goal. Even though the decision considers sequences of tactics over the look-ahead horizon, it only commits to the ones that must be started at the current time. Another approach is the selection of an adaptation *strategy* to achieve the adaptation goal. A strategy is a predefined decision tree built out of tactics [28]. For example, a strategy to reduce the response time in RUBiS, could add a server, wait to check if it reduces the response time to a satisfactory level; if it does not, add another server, wait and check; and if it still has not reduced the response time to the desired level, decrease the dimmer. In this way, an adaptation strategy captures the process a system administrator might follow to repair a problem [27, Ch. 4]. A strategy-based adaptation decision consists in selecting the best strategy to achieve the adaptation goal from a repertoire of strategies.

Tactic-based adaptation is as, or more, effective than strategy-based adaptation. Intuitively, the former has the flexibility to generate the same solutions the latter could generate. Since strategies are pre-planned ways in which tactics can be combined [28], strategy-based adaptation lacks the flexibility to generate all the solutions that tactic-based adaptation could provide, some of which could be better. Despite this disadvantage, strategy-based adaptation may be desired for either *scalability*, or what we loosely refer to as *trust*.

Scalability. PLA-SDP has to iterate multiple times over the system configuration space when making an adaptation decision.[1] In systems with a large adaptation-relevant configuration space, the performance of the PLA-SDP will suffer. In addition, adaptation tactics with latency also contribute to the size of the computational state space due to the need to keep track of their progress. Strategy-based approaches do not suffer these scalability issues because the decision time is mainly driven by the number of strategies and their structure, regardless of the size of the system configuration space (Cheng shows this for reactive strategy-based adaptation [27, Ch. 4], and we present a detailed analysis for using strategies in PLA in Section 7.4).

[1] In this chapter we focus on PLA-SDP to compare the scalability of tactic-based and strategy-based approaches, since PLA-SDP scales better than PLA-PMC (see Section 5.4).

Trust. Dependability and predictability are two important factors that affect trust in automation [14, 128]. According to Schaefer et al., "dependability refers to the consistency and effectiveness of a behavior" [128]. For that reason, system administrators may desire to limit the actions of the system to strategies they have tried before and are known to have worked reasonably well, as opposed to letting the system come up with untested arbitrary adaptation plans, even if that means not attaining the most effective adaptation. Furthermore, since these strategies are known in advance, it is possible to analyze off-line what problems they can solve and how well [129]. Predictability means matching the system administrator's expectation [128]. Dragan et al. make an interesting distinction between predictability and the sometimes correlated concept of legibility [38]. Predictable behavior matches the system administrator's expectation of what a system should be doing to achieve a known goal. For example, if the goal is to reduce the response time of the system, the system administrator may expect the system to add more servers. Legibility, on the other hand, is an inference in the opposite direction, meaning that the system administrator can infer the goal from the observed behavior. Using strategies can help both predictability and legibility, since strategies are used to "capture routine human adaptation knowledge as explicit adaptation policies" [27]. Therefore, the system behavior using a strategy will match the system administrator's expectation. Legibility is supported with strategy-based adaptation because it is easier for the system administrator to identify a behavior from the label of the strategy being used, and associate that to the goal of the strategy.[2] Furthermore, the internal structure of a strategy can also shed light into the behavior of the system, allowing, for example, a system administrator to understand the conditions that triggered some adaptive behavior.

To date, strategies have only been used in a reactive way and without considering the latency of adaptation. By bringing latency into the calculations needed for the selection of a strategy, we can obtain much better ways to predict the outcomes of a strategy. In addition, by making the selection of strategies and their execution proactive with look-ahead, we can address the limitations of reactive adaptation discussed in previous chapters.

In this chapter we present SB-PLA, a strategy-based proactive latency-aware approach. SB-PLA improves strategy-based adaptation, and allows reaping the benefits of PLA for systems in which strategy-based adaptation is desired or needed. Section 7.1 gives some background on strategies, the approaches that use them, and the limitations of those approaches. Section 7.2 describes how SB-PLA works. In Section 7.3, we compare the effectiveness of SB-PLA to a strategy-based adaptation that does not use the proactive latency-awareness principles. Section 7.4 shows how the strategy-based approach scales better than PLA-SDP.

7.1 Background

The use of strategies for self-adaptation was proposed by Cheng et al. [29], and became a key aspect of the Rainbow framework [53]. Rainbow provides a reusable infrastructure to implement architecture-based self-adaptive systems. Figure 7.1 shows the self-adaptation loop embodied

[2]Even though tactics also have labels, they represent primitive adaptation actions that can be used to achieve different goals, making it more difficult to infer the adaptation intent from their label. For example, the tactic to decrease the dimmer can be used as the sole action to deal with increased load, or just temporarily, while a new server is being added.

by the Rainbow framework, which resembles the MAPE-K loop. The target system and its environment are monitored through *probes*. The monitoring information is aggregated by *gauges*, and used to update models within the *Model Manager*. Although different kinds of models are supported by Rainbow, the primary models used for adaptation decisions are architectural models specified in *Acme*, an architecture description language [52]. The *Architecture Evaluator* analyzes the architecture model to determine if there is a need to adapt, and if that is the case, the *Adaptation Manager* selects the adaptation strategy most suitable to deal with the current conditions. The strategy is then executed by the *Strategy Executor* using *effectors* to carry out the changes to the system dictated by the strategy.

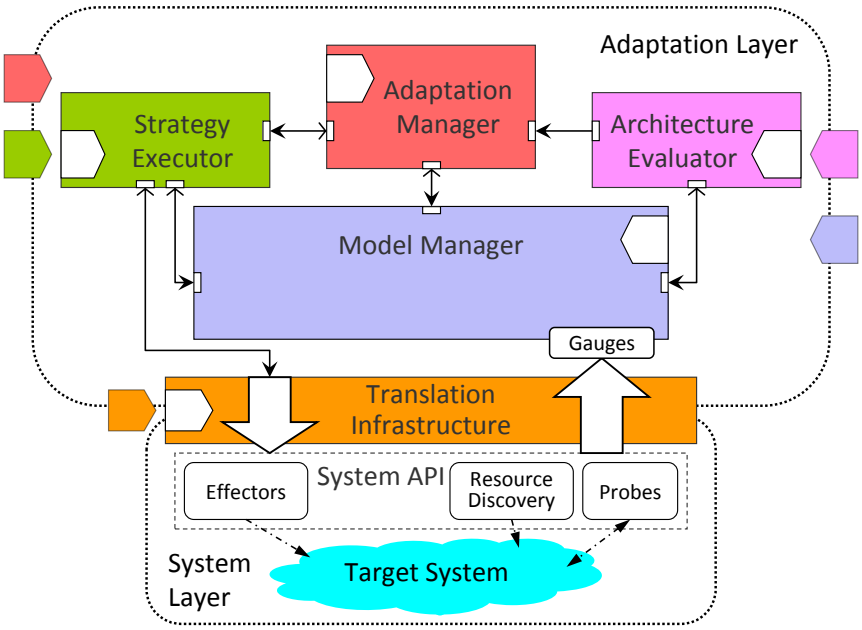


Figure 7.1: The Rainbow self-adaptation framework [54].

Rainbow has several customization points, depicted by the cutouts in Figure 7.1 [54]. The *Translation Infrastructure* supports the definition of mappings to bridge the gap between the architectural abstractions in the models and concrete elements in the target system. For example, when an effector is invoked by the Strategy Executor, the architectural element to be affected must be translated to the corresponding element in the target system. The Model Manager is customized with architectural models of the target system, and the Architecture Evaluator is tailored with the constraints that self-adaptation must maintain for the system. These constraints are expressed as predicates over the architectural model. The Adaptation Manager is customized with a repertoire of adaptation tactics and strategies, and a utility function that is used to select which strategy to use when a constraint has been violated. The Strategy Executor is tailored with operators that change the target system through the effectors.

As noted above, one of the several customizable aspects of Rainbow is the repertoire of strategies it can use to adapt the target system when it is not performing as desired. These strategies are defined in the *Stitch* language [28]. Stitch allows defining tactics, which then are composed into strategies. When an adaptation decision has to be made, there may be situations in

which multiple strategies from the repertoire are applicable; in that case, strategy selection takes into account the impact that the applicable strategies would have on the system along different qualities of interest, such as response time and cost. A utility function combines these impacts according to a business context (e.g., response time considered more important than cost) in order to produce a final score for each strategy.

Listing 7.1 shows the Stitch code defining a tactic to set the dimmer value to its minimum level. The **condition** block is a predicate over the model that defines when the tactic is applicable. In this case, the condition is always true because this tactic is idempotent, and because the writer of the tactic considered that it was reasonable to invoke it at any time. The **action** block defines what the tactic does when it is executed, using operators that can change the system. In this example, the operator setDimmer defined over the model M takes a component of the system as the first argument, which must be a load balancer, and the value that its dimmer must be set to as the second argument (line 6). The load balancer of the system is the component M.LB0, and the minimum value for the dimmer is M.DIMMER_MARGIN, a constant defined in the system model. The **effect** block defines a predicate that represents the expected outcome of the tactic and it is used to check whether the tactic completed successfully.

```
1  tactic TSetMinDimmer() {
2    condition {
3      true;
4    }
5    action {
6      M.setDimmer(M.LB0, M.DIMMER_MARGIN);
7    }
8    effect {
9      M.LB0.dimmer == M.DIMMER_MARGIN;
10   }
11 }
```

Listing 7.1: Sample tactic in Stitch.

In Stitch, a strategy is encoded as a decision tree with condition-action-delay nodes [28]. Listing 7.2 shows a sample strategy to reduce the response time as fast as possible encoded in Stitch. Predicates over the model can be used to specify conditions and guards. For example, the predicate between square brackets in line 1 is the applicability condition for the strategy, indicating in this case that the strategy is applicable when the average response time is above the response time threshold (see [28] for more details about predicates in Stitch). Each node in the tree has a label (e.g., t1 in line 2) followed by a condition. If the condition is true, the action following the condition after the operator $->$ is executed. If more than one node is enabled by its condition, then one of them is chosen nondeterministically. The action can be either a tactic, or the keywords **done** to terminate the strategy successfully, or **fail** to terminate the strategy with a failure (e.g., if the adaptation goal was not achieved). The special condition **success** is true if the previous tactic was successful, and **default** is true if no other node is enabled. The children of each node in the tree are defined within curly braces (e.g., nodes t1a and t1b are children of node t1).

The strategy in Listing 7.2 has three top-level nodes. In node t1, if the dimmer is not already at the minimum (!isDimmerMin), the tactic to set the dimmer to its minimum setting is executed

```
1  strategy FastReduceResponseTime [ M.LB0.avgResponseTime > M.RT_THRESHOLD] {
2    t1: (!isDimmerMin) -> TSetMinDimmer() {
3      t1a: (success) -> done;
4      t1b: (default) -> fail;
5    }
6    t2: (isDimmerMin && canAddServer) -> TAddServer() @[180000 /*ms*/] {
7      t1a: (success) -> done;
8      t1b: (default) -> fail;
9    }
10   t3: (default) -> fail;
11 }
```

Listing 7.2: Sample strategy in Stitch.

(TSetMinDimmer). If this tactic succeeds, the strategy terminates successfully (line 3); otherwise, it terminates with failure (line 4). Node t2 adds a server with tactic TAddServer if the dimmer is already at its minimum setting and it is possible to add a server. This node has a delay specified with @[180000 /*ms*/], indicating that after the tactic is started, Rainbow will wait up to 3 minutes for the effect of the tactic to be observed. If neither t1 nor t2 are applicable, node t3 makes the strategy terminate as having failed.

When adaptation is needed, a strategy is selected by first filtering out all the nonapplicable strategies based on their applicability conditions, and then selecting the one with the higher score. To compute the score of a strategy, each tactic defines an impact vector that specifies how different utility dimensions are affected by the tactic (e.g., cost is increased by one unit, and response time is decreased by 500 ms). Starting at the leaves of the strategy's tree, the impacts are aggregated working towards the root using the probability of selecting each child node to compute the expected impact vector of the parent node. Unless otherwise specified, each child is assumed to have equal probability of being chosen (see [28] for mode details).

This strategy selection method has two limitations. First, the computation of the impact of the strategy is largely independent of the context.[3] For example, the same impact of adding a server is assumed to be independent of the traffic arriving at the system, and also independent of the number of servers already active in the system. The second limitation is that the latency of the adaptation tactics is ignored. Even though a delay can be specified for a tactic in a strategy, this delay is only used during the execution of the strategy, but it is not taken into account when a strategy is being selected. Recent work has addressed the first limitation by using probabilistic impact models that allow evaluating each strategy in the context of the current state of the system and environment, and taking into account the stochastic nature of some tactics (e.g., a tactic could fail with some probability) [21]. However, this evaluation is also time-agnostic. For example, when the utility impact of a strategy is computed, it is assumed that the environment will not change during the execution of the strategy, despite the fact that it can include multiple tactics with latency. A strategy that is the most appropriate for a snapshot of the environment, may not be the best one when the evolution of the environment is considered. For example, a strategy that adds a server and then decreases the dimmer level if necessary may not be the most appropriate if the request arrival rate is already predicted to keep increasing while the server is being added. In that case, decreasing the dimmer first may be the best course of action. By addressing these

[3]The initial filtering of strategies based on their applicability condition is context dependent, though.

limitations, the impact of executing an adaptation strategy can be better estimated, reflecting how the utility accrued by the system will change as the environment and the system change during the execution of the tactic. This, in turn leads to a better selection of the most appropriate adaptation strategy, thus improving the effectiveness of the adaptation.

7.2 Approach

SB-PLA improves over previous strategy-based adaptation approaches by (i) taking into account the latency of the different adaptation tactics when computing the utility a strategy provides; and (ii) taking into account how the environment is expected to evolve during the execution of the strategy. To accomplish that, it builds on an approach to encoding the decision tree of a strategy as a discrete-time Markov chain (DTMC) [129], and on PLA-PMC for modeling tactic latency in the context of an evolving environment.

The overall SB-PLA approach consists in

1. filtering out inapplicable strategies (as is done currently)
2. scoring each applicable strategy using probabilistic model checking to compute the utility each is expected to accrue over the decision horizon
3. selecting the strategy with the higher score

In the following sections we describe the strategy scoring approach first (7.2.1), and then how a the overall strategy-based adaptation decision works (7.2.2).

7.2.1 Strategy Scoring

The score of a strategy is computed as the expected utility the system would accrue over the decision horizon if the strategy was executed. To do that, we use probabilistic model checking to compute the score, analyzing a formal model of the self-adaptive system composed with the model of the environment, as in PLA-PMC. However, unlike in PLA-PMC, the model checker is not used to make the adaptation decision. Instead, the decision is made outside of the model checker using the scores computed with it.

The main difference with PLA-PMC is that instead of modeling multiple tactics with non-deterministic starts, which are then resolved by the model checker, the SB-PLA model includes only the strategy being scored, which is started deterministically at the beginning of the decision horizon. That is, there is no nondeterminism in the model to be resolved, and thus it is a DTMC and not an MDP.

The structure of the model for scoring a strategy is depicted in Figure 7.2. The modules for the clock, the environment, and the system are identical to those used for PLA-PMC. However, instead of having separate modules for each tactic, there is a single module that encodes the strategy, including the execution of the tactics it uses. The system module synchronizes with the strategy module on actions that represent the completion of tactics within the strategy, so that the effect the tactics can be reflected on the system as they complete.

The encoding of a strategy in PRISM builds on the pattern used by Schmerl et al. [129], enhanced with latency-awareness. Listing 7.3 shows the PRISM model of the strategy shown in

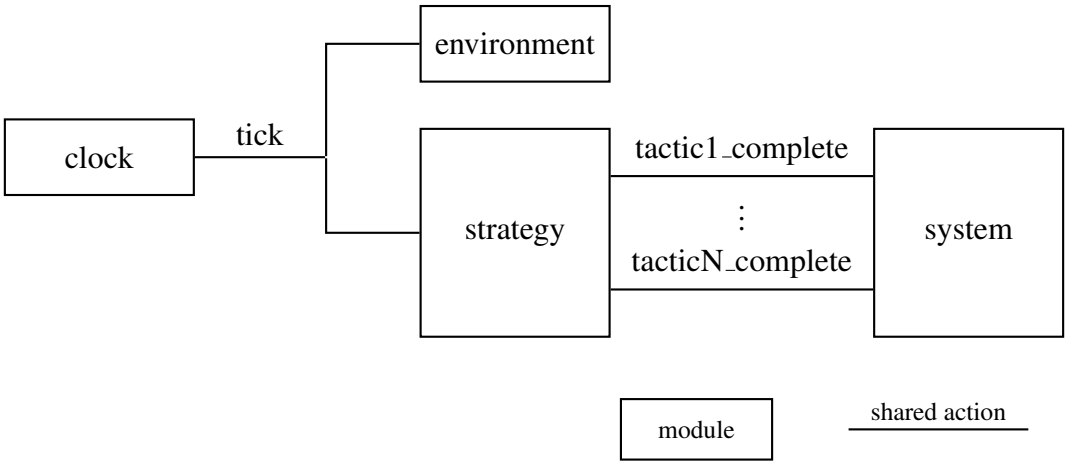


Figure 7.2: Module composition in strategy scoring model.

Listing 7.2. The state variable node indicates the node in the strategy that is being evaluated or executed, with node=0 indicating the root node. For the purpose of computing utility, whether a strategy finishes with **done** or **fail** is irrelevant, since the system will accrue utility regardless (although different amounts). Therefore, all the leaf nodes of the strategy are represented by a single value of the node variable. In this example, this value is 3, and the helper predicate formula leaf (line 1) is defined to easily identify the leaf node.

The different branches of a node are represented by commands with the following pattern:

sys_go & node=*nodeId* & *branchCondition* −> 1: (node'=*nextNodeId*);

The predicate sys_go is true when it is the turn for the system and its adaptation strategy to execute actions that do not involve the passage of time. For example, as long as sys_go is true, the strategy can execute multiple instantaneous tactics. Only when the strategy has completed, or needs the passage of time for a tactic with latency to make progress, does it have to synchronize with the clock on the tick action (lines 17 and 18 respectively). The node=*nodeId* conjunct of the guard indicates which node these branches correspond to. Lines 8-10 represent the branches of the root node of the strategy, thus the conjunct node=0. The *branchCondition* conjunct is the condition that the branch has in the Stitch encoding of the strategy. In the case of the **default** branch, the condition is the conjunction of the negation of the conditions of all the other branches for that node (line 10). The update portion of these commands sets the variable node to the index of the node that corresponds to the child for that branch, *nextNodeId*, as in the update of the commands in lines 8-10.

The encoding of the action part of a strategy node depends on whether the tactic used in the action has latency or not. If the tactic is instantaneous, as is the case for TSetMinDimmer() in Listing 7.2, line 2, the command in its PRISM encoding follows the pattern:

[*tacticAction*] sys_go & node=*nodeId* −> 1: (node'=*nextNodeId*);

where *tacticAction* is the label for the action that synchronizes with the system module upon the execution of the tactic (see lines 5-8 in Listing 4.3, for example). The value *nextNodeId* in the update portion of the command corresponds to the node that follows in the strategy after

```
1  formula leaf = (node=3);
2
3  module Strategy
4    node : [0..3] init 0;
5    exec : [0.. MAX_LATENCY] init 0; // remaining tactic execution
6    tacticRunning : bool init false; // tactic with latency running
7
8    [] sys_go & node=0 & (!isDimmerMin) -> 1: (node'=1);
9    [] sys_go & node=0 & (isDimmerMin & canAddServer) -> 1: (node'=2);
10   [] sys_go & node=0 & !(!isDimmerMin) & !(isDimmerMin & canAddServer) -> 1: (node'=3);
11
12   [MinDimmer_start] sys_go & node=1 -> 1: (node'=3);
13
14   [] sys_go & node=2 & !tacticRunning -> 1: (tacticRunning'=true) & (exec'=TAddServer_LATENCY); // tactic start
15   [AddServer_complete] sys_go & node=2 & tacticRunning & exec=0 -> 1: (tacticRunning'=false) & (node'=3); // tactic
         completion
16
17   [tick] leaf -> 1: true; // strategy finished
18   [tick] exec > 0 -> 1: (exec'=exec-1); // tactic progress
19
20   // prevent all other tactics from executing
21   [RemoveServer_start] false -> true;
22   [DecDimmer_start] false -> true;
23   [IncDimmer_start] false -> true;
24   [MaxDimmer_start] false -> true;
25 endmodule
```

Listing 7.3: Sample strategy in PRISM

the execution of the tactic. Line 12 in Listing 7.3 is an example of this encoding. For this particular strategy, all paths in its tree have a leaf immediately after the first tactic, and thus, they transition directly to the leaf node (e.g, node'=3 in line 12). In cases with longer paths, it would be necessary to model an intermediate node to transition to after the tactic is executed, but before the selection of the next child node, much like the root node (node=0) models the branch selection at the beginning of the strategy.

For tactics with latency, the PRISM encoding consists of three parts: starting the tactic, allowing the tactic to make progress, and handling the completion of the tactic. The starting of the tactic is done through a command with the following pattern:

```
[] sys_go & node=nodeId & !tacticRunning -> 1: (tacticRunning'=true)&
                                                (exec'=tacticLatency);
```

The boolean variable tacticRunning is true only if a tactic with latency is already running.[4] Therefore, this variable has to be false in the guard for the command to start a tactic. The update portion of the command sets this variable to **true** and the variable exec to the latency in periods of the tactic being started. In line 14, the constant TAddServer_LATENCY represents the latency for the tactic *AddServer*. The exec variable represents the number of periods left before the running tactic completes. Since it would be possible to have more than one tactic with latency in the strategy, the range of exec is defined up to the maximum of the latencies of all the tactics (line 5).

[4] Stitch does not support concurrent tactic execution; thus a boolean variable is sufficient.

The progress of the tactic is handled by line 18, regardless how many tactics with latency the strategy has. This command decreases exec each time the model clock ticks until the tactic execution completes.

The completion of the tactic is modeled with a command of the form

[*tacticAction*] sys_go & node=*nodeId* & tacticRunning & exec=0 −>
1: (tacticRunning'=**false**) & (node'=*nextNodeId*);

That is, when the remaining execution time of the running tactic reaches zero, the tactic has completed. At that point, the strategy synchronizes with the system module on the action *tacticAction* to reflect the impact of the tactic completion on the system state. As in the case with a tactic with no latency, the update portion of the command sets node to the node that follows in the encoding of the strategy tree.

When the execution of the strategy reaches a leaf node, the module stays in line 17, allowing the clock module to make progress freely. This allows the environment evolution and the accounting of utility to continue throughout the decision horizon even if the strategy finishes before the end of the horizon.[5]

At the end of the encoding of the strategy in PRISM (lines 21-24), there is a block of commands with false guards that synchronize on all the tactic actions that are not used by the strategy. This is needed to prevent the system module from executing the commands associated with those tactics, which would be unconstrained otherwise. For example, without these commands, the system could arbitrarily remove a server if the server count is greater than one, even though the strategy does not use that tactic. These commands prevent that from happening.

To compute the score of a strategy, the module of the strategy has to be composed with the other modules shown in Figure 7.2. Operationally, this means concatenating the PRISM module for the strategy with the code for the other modules to create a complete model. Then, the score of the strategy is computed as the expected utility accumulated until the end of the decision horizon, using the PRISM model checker to analyze the following PCTL property extended with rewards.

$$\mathsf{R}^{\mathsf{util}}_{=?}[\mathsf{F}^{\mathsf{c}}\mathsf{end}]$$

This computation is encapsulated in a function $evaluateStrategy(s, c, env)$, where s is the strategy to be evaluated; c is the current state of the system, which is used to generate the initial state of the variables in the model; and env is a model of the predicted evolution of the environment. The following section explains how this function is used in the overall adaptation decision with SB-PLA.

7.2.2 Adaptation Decision

The adaptation decision has to determine which strategy from the repertoire, if any, should be started to maximize the expected utility that the system will accrue over the decision horizon. To be proactive, this decision has to be done periodically, so that if a strategy involves a tactic

[5]In practice, another strategy may be invoked to fill out the horizon. However, accounting for that would require analyzing a model that would not only score a strategy but also make decisions about how to adapt after that strategy. As a trade-off favoring scalability, we chose to limit the analysis to the scoring of a single strategy.

with latency, it can be started with the necessary lead time. However, since the decision is done periodically without a concrete need for adaptation as a trigger, it is necessary to consider the possibility of not adapting when making the adaptation decision. To this end, we define the *NoOp* strategy, which has no actions and is always applicable.

Algorithm 7.3 shows how the adaptation decision is made. The function SBPLA takes the current system configuration, and a model of the predicted evolution of the environment as input, and returns the strategy selected by the adaptation decision. The algorithm starts by initializing s^*, the selected strategy, to $NoOp$, so that this strategy is the result in case no other strategy is applicable, or if no other applicable strategy has a higher score than $NoOp$. It then computes S, the set of all the strategies from the repertoire that are applicable in the current system state (line 3). Note that $NoOp$ is not in the repertoire. If S is empty, the result is $NoOp$, otherwise, the best strategy has to be selected using the scoring approach presented in the previous section. In the latter case, $v*$, representing the best score, is initialized to the score of the $NoOp$ strategy given by the $evaluateStrategy$ function (line 5). For each strategy in S, the score of the strategy is computed, and only if it is higher than the previous best score, the best strategy and best score, s^* and v^* respectively, are updated (lines 8-11). In the end, the best strategy is returned, which constitutes the outcome of the SB-PLA adaptation decision.

```
1:  function SBPLA(c, env)
2:      s* ← NoOp
3:      S ← {s ∈ Repertoire : isApplicable(s, c)}
4:      if S ≠ ∅ then
5:          v* ← evaluateStrategy(NoOp, c, env)
6:          for all s ∈ S do
7:              v ← evaluateStrategy(s, c, env)
8:              if v > v* then
9:                  v* ← v
10:                 s* ← s
11:             end if
12:         end for
13:     end if
14:     return s*
15: end function
```

Figure 7.3: SB-PLA adaptation decision algorithm.

7.3 Effectiveness

To show how SB-PLA improves over non-proactive non-latency-aware strategy-based adaptation (SB), we compare two runs of the RUBiS simulation (see Section 8.1.1), one with each approach. These runs were done with both approaches implemented in Rainbow. The SB approach was implemented with the scoring of the strategy implemented using probabilistic model checking

as well. The only difference between the approaches was that in SB, the clock module was modified to allow the strategy to execute completely before advancing the clock to compute the accumulated utility over the decision horizon, thereby, ignoring the latency of the tactics used by the strategy. Table 7.1 summarizes the strategies available to the system.

Table 7.1: Adaptation strategies for RUBiS.

Strategy	Description
AddServer	Adds a server
ReduceContentAndAddServer	Decreases the dimmer one level and adds a server
MinimizeContent	Sets the dimmer to the minimum level
RestoreFullContent	Sets the dimmer to the maximum level
RemoveServer	Removes a server

The results for SB and SB-PLA are summarized in Table 7.2. Although both approaches are close in terms of the average number of servers used and the optional content served, we can see that SB-PLA attains much higher utility (about 50% more), and has considerably fewer late responses. Plots of different metrics shown in Figure 7.4 and Figure 7.5 give some insight into the reasons why SB-PLA does better. As expected, SB-PLA starts the strategies that involve the addition of a server with more lead time, taking into account the latency of that tactic. SB never used the strategy *ReduceContentAndAddServer*, and always used the strategy *AddServer* when a new server was needed. This is expected, since SB assumes that the addition of a server takes no time, and consequently, it sees no benefit in reducing the content while the server is being added. Interestingly, SB-PLA used both of these strategies. In most cases it used *ReduceContentAndAddServer* to deal with the latency. However, at time 1440, it used *AddServer*, apparently because the upward trend of the arrival rate was low enough to give it time to add a server without reducing the content. This shows that SB-PLA is more nuanced with respect to the timing of the adaptation in the context of the evolution of the environment.

Table 7.2: Comparison of strategy-based approaches.

Approach	Utility	% Optional	% Late	Avg. Servers
SB	1072	72.6	17.2	1.9
SB-PLA	1568	66	2.8	1.8

7.4 Scalability Analysis

We now assess the scalability of SB-PLA compared to PLA-SDP.[6] Solving the adaptation decision problem with either a tactic-based or a strategy-based approach involves the computation of

[6]This section only compares SB-PLA and PLA-SDP. The validation of the claims of the thesis is presented in Chapter 8.

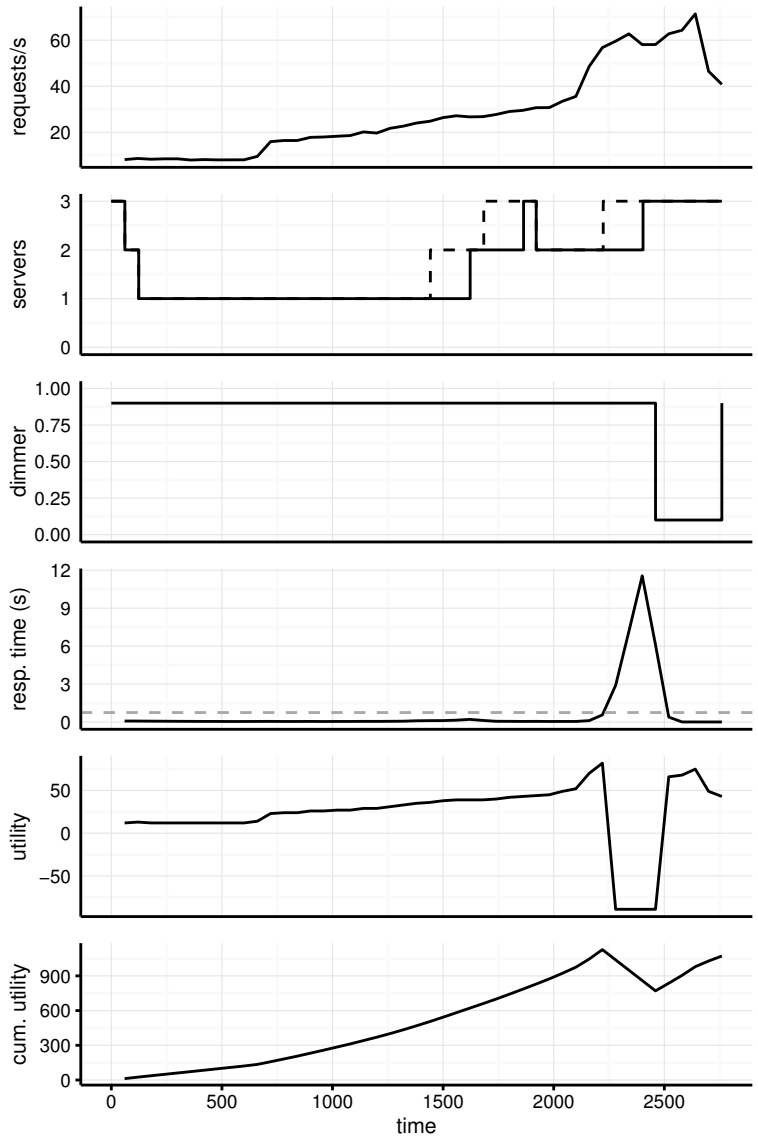


Figure 7.4: Sample run of SB adaptation.

the estimated utility the system would accrue for numerous system and environment state pairs. This computation outweighs other simpler operations the algorithms use, since it involves not only computing the utility, but also predicting the inputs to the utility function, such as the response time. For this reason, we compare the scalability of the two approaches in terms of the number of utility calculations that they have to perform, which we denote with K.

In PLA-SDP, a utility calculation must be performed for every possible system configuration, at each possible environment state in a given period. The number of possible system configurations is $|C|$. The number of environment states in a given period in the decision horizon depends on the structure of the environment DTMC, and it may be different for different periods, as is the case in a probability tree. Thus, we use the maximum number of environment states in a period and denote it by e_{max}. The evaluation of all system configurations in all possible environment

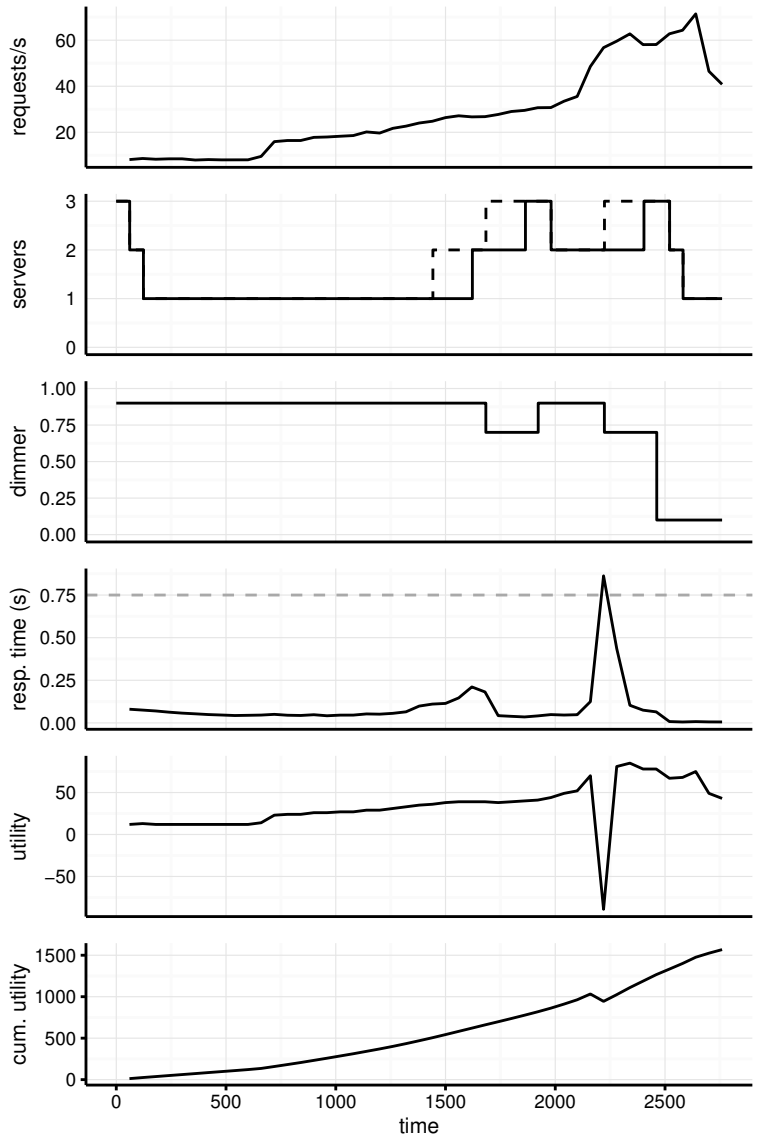


Figure 7.5: Sample run of SB-PLA adaptation.

states has to be done for each period in the decision horizon, and therefore repeated H times. The number of utility calculations for PLA-SDP is then

$$K_{PLA\text{-}SDP} = |C| e_{max} H \tag{7.1}$$

In SB-PLA, the number of utility calculations depends mainly on the number of nodes (leaves included) in the decision tree defined by the strategy. We characterize the tree by its branching factor, b, and its depth in terms of tactics, d. The worst case for the number of utility calculations is when all the tactics in the strategy have no latency, because all the leaves of the decision tree are possibly reachable by the first period of the decision horizon, requiring utility calculations

through the end of the horizon. Since the number of leaves is b^d, and assuming that the number of applicable strategies is g, the number of utility calculations for SB-PLA is

$$K_{SB\text{-}PLA} = g b^d e_{max} H \tag{7.2}$$

If the tactics in a strategy have a latency equal to one decision period, the number of utility calculations in each of the first d periods of the decision horizon increases as the number of nodes at the corresponding level of the tree increases. That is, for all of the first d periods, the number of utility calculations is the number of nodes in the decision tree times the number of environment states; and for the remaining $H - d$ periods, it is equal to the number of leaves in the decision tree times the number of environment states. The number of internal nodes in a tree is given by $(b^d - 1)/(b - 1) - 1$, thus the number of utility calculations for SB-PLA when tactics have one-period latency is

$$K^1_{SB\text{-}PLA} = g \left(\frac{b^d - 1}{b - 1} - 1 + b^d (H - (d - 1)) \right) e_{max} \tag{7.3}$$

Generalizing (7.3) for tactics with latency L, we have

$$K^L_{SB\text{-}PLA} = g \left(L \left(\frac{b^d - 1}{b - 1} - 1 \right) + b^d (H - L(d - 1)) \right) e_{max} \tag{7.4}$$

A direct comparison of how PLA-SDP and SB-PLA scale is not possible because the number of calculations for each depends on different parameters. For PLA-SDP, the number of system configurations will have the most impact, since the H, and e_{max} are likely to be much smaller than the size of C. On the other hand, for SB-PLA, the number of strategies and their structure have the most influence on the number of calculations it requires. Since the size of C is affected by the number of tactics and their latency—due to the need to keep track of tactic progress—we can make some assumptions and compute $|C|$ as a function of the number of tactics in the system and their latency. In addition, we can make an assumption about how the number of tactics affect the number of applicable strategies and compute g. In this way, we can compare the scalability of the two approaches varying the number of tactics and their latency.

Let us assume that each pair of tactics controls a property of the system with three possible values. If a is the number of tactics, then the number of possible system configurations, not accounting for tactic progress tracking, is $3^{a/2}$. If the tactics have latency, and assuming all tactics have latency L, then the number of system states needed to keep track of tactic progress is $(L + 1)^a$. Thus, we can compute the number of system configurations as

$$|C| = 3^{a/2} (L + 1)^a \tag{7.5}$$

We assume that the number of applicable strategies is half the number of tactics, since the number of strategies in the repertoire will be smaller than the number of tactics, and not all of them are applicable for a given adaptation decision. For the rest of the parameters, the following values are used: $H = 10$, $e_{max} = 9$, $b = 2$, and $d = 3$.

Figure 7.6 shows plots of the number of utility calculations needed for each approach with these assumptions as the number of tactics increases, with each plot using a different tactic

latency. We can observe that the number of calculations increases much faster for PLA-SDP as the number of tactics increases, especially considering that the plots use a logarithmic scale. In addition, it is important to note that in SB-PLA each strategy can be scored independently of the others. Therefore, it is possible to parallelize the adaptation decision, making SB-PLA even more scalable.

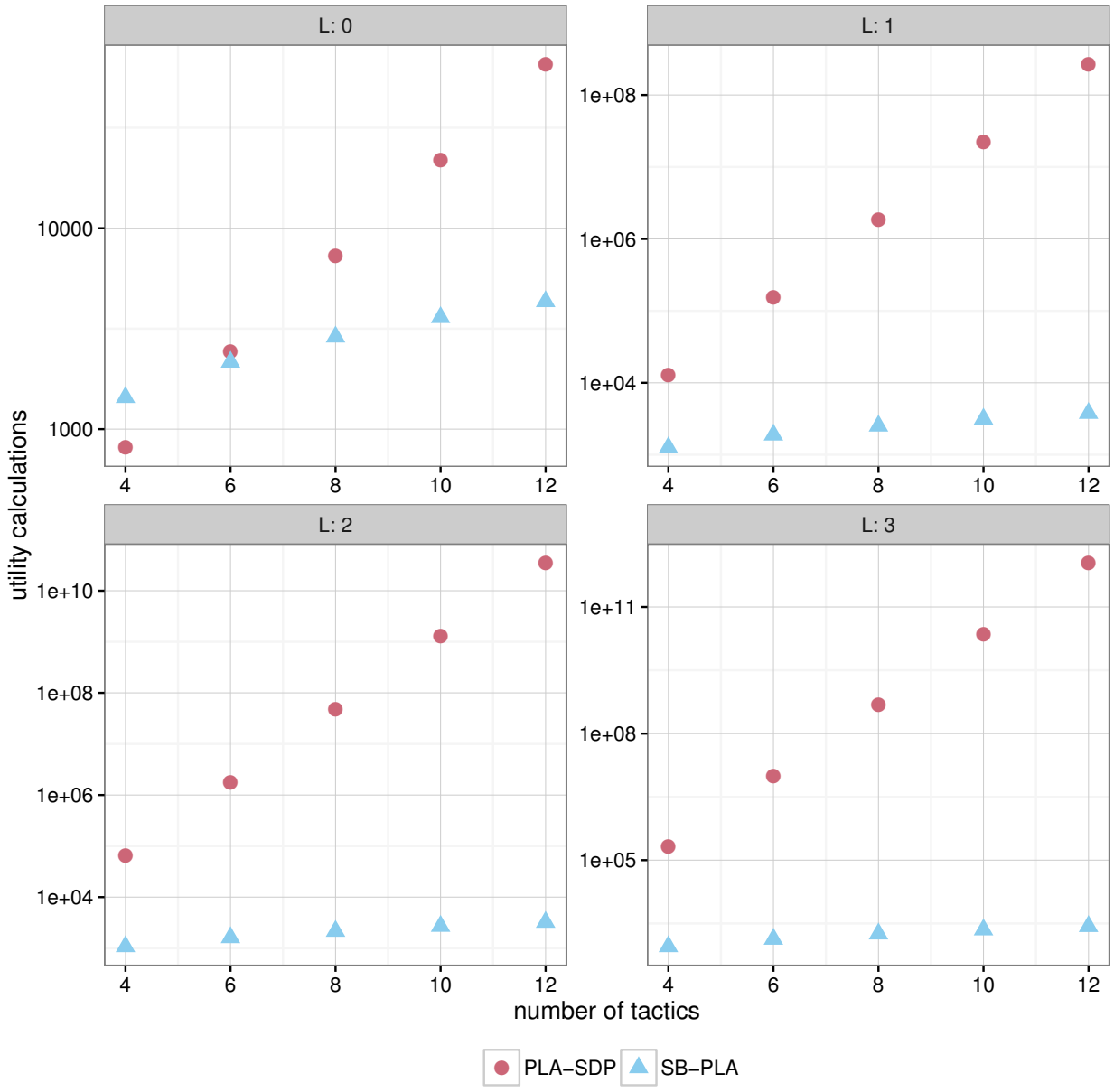


Figure 7.6: Scalability comparison of PLA-SDP and SB-PLA.

7.5 Summary

In this chapter we have presented SB-PLA, an alternative proactive latency-aware self-adaptation approach based on adaptation strategies. Instead of having the freedom to compose adaptation tactics in every feasible way as PLA-PMC and PLA-SDP do, this approach is limited to selecting the best strategy—a predefined composition of tactics—from a repertoire. Doing this may be desired for reasons of scalability and trust, limiting the adaptive behavior to strategies that are understandable by humans, and have been tested before.

We showed that SB-PLA scales much better than the other approaches presented in this thesis, while still providing an improvement over non-PLA strategy-based adaptation. Therefore, SB-PLA could be a suitable compromise for systems with large adaptation spaces in which the decision time of the other approaches would overrun the desired or needed adaptation period.

Chapter 8

Validation

This chapter presents the validation of the claims of the thesis, which was introduced in Chapter 1 and is restated here.

> **We can improve the effectiveness of self-adaptation over reactive time-agnostic adaptation by (a) explicitly considering the latency of adaptation tactics, (b) adapting proactively, and (c) potentially allowing concurrent execution of adaptation tactics.**

The principles of proactive latency-aware adaptation introduced in Chapter 3, and its main approaches presented in chapters 4 and 5 realize the key ideas stated in (a), (b), and (c) above. In this chapter, we focus on the validation of the claims described in Section 1.4, summarized here:

Claim 1. *The approach improves the effectiveness of self-adaptation.*
Claim 2. *The approach is applicable to different kinds of systems.*
Claim 3. *The approach scales to systems of realistic size.*

To this end, we use two systems in completely different domains: RUBiS, a web system; and DART, a self-adaptive formation of drones. The systems are described in Section 8.1 first, and the results and evidence that substantiate the claims of the thesis are presented in Section 8.2.

8.1 Validation Systems

We used two systems for the validation of the thesis, RUBiS and DART. These systems are different in several ways, not only in their application domain, but also in the kind of adaptation goal, the adaptation frameworks used to implement them, and the kinds of environment predictions they use. Table 8.1 summarizes the difference between these systems, which are described in detail in the following sections.

8.1.1 RUBiS

RUBiS was already introduced in Section 1.1. Here we provide more details about its implementation, and formalize its adaptation goal. In addition, we describe the RUBiS simulation, which was used for some of the experiments.

Table 8.1: Validation systems.

	RUBiS	**DART**
domain	website	formation of unmanned air vehicles in hostile environment
adaptation goal	maximize utility according to SLA at minimum cost	maximize number of targets detected while keeping the probability of surviving the mission above a threshold
environment prediction	forecast based on past environment states using time series predictor	forward-looking sensors that give false positives and false negatives for targets and threats
measure of performance prediction	queuing theory	formulas that model the sensor performance, and the effect of threats
implementation frameworks	two different implementations: • Rainbow [53] • custom PLA self-adaptation loop in OMNeT++ simulation	two different implementations: • DART architecture [65] • custom PLA self-adaptation loop in C++ simulation
adaptation tactics	add server, remove server, increase dimmer, decrease dimmer	increase altitude, decrease altitude, switch to tight formation, switch to loose formation, turn ECM on, turn ECM off

Although RUBiS has been used extensively for research in web application performance, and various areas of cloud computing [39, 50, 74, 122], it was not originally designed as a self-adaptive system, since it does not provide actuators suitable for adaptation. We started with the version of RUBiS extended with *brownout* support [83], and added a load balancer, as shown in Figure 1.1, so that the number of servers could be changed dynamically. Also, an adaptation layer was added with monitoring, adaptation, and execution components to implement the self-adaptation loop depicted in Figure 3.2.

We included two pairs of adaptation tactics that can be used to deal with the changing arrival rate and the load it induces. One pair of tactics can be used to add and remove servers, thus changing the capacity of the system. The tactic to add a server has a latency λ.[1] The inverse tactic removes a server. Although this requires waiting for the server to complete the processing of its requests, we assume that time to be negligible, and thus assume the tactic to be immediate.[2] The other pair of tactics leverages the brownout paradigm. All responses that RUBiS provides

[1]The latency is assumed to be constant, but if it were a random variable, λ would be its expected value.

[2]This is just a choice we made for this example. If that time were not negligible compared to the decision period, the tactic could be modeled as a tactic with latency.

must include mandatory content, such as the details of an item being browsed. The dimmer in brownout, in this case, controls the proportion of responses that include optional content, which in this case is a list of recommended related items. In that way, it is possible to use the dimmer to control the load on the system that is induced by the requests users make. The value of the dimmer can be thought of as the probability of a response including the optional content, thus taking values in $[0..1]$. To control the dimmer, the system has two immediate adaptation tactics that increase and decrease its value. We allow tactics to be executed concurrently only if they belong to different pairs. For example, if a server is being added, a server cannot be removed, but it is possible to increase or decrease the dimmer.

The goal of self-adaptation in our example, introduced in Chapter 1 and repeated here for convenience, is to maximize the utility provided by the system at the minimum cost. The utility is computed according to a service level agreement (SLA) with rewards for meeting the average response time requirement in a measurement interval, and penalties for not meeting it [77]. The cost is proportional to the number of servers used. The SLA specifies a threshold T for the average response time requirement. The utility obtained in an interval depends on whether the response time requirement is met or not, as given by

$$U = \begin{cases} \tau a(dR_O + (1-d)R_M) & \text{if } r \leq T \\ \tau \min(0, a - \kappa)R_O & \text{if } r > T \end{cases} \tag{8.1}$$

where τ is the length of the interval, a is the average request rate, r is the average response time, d is the dimmer value, κ is the maximum request rate the site is capable of handling with optional content, and R_M and R_O are the rewards for serving a request with mandatory and optional content, respectively, with $R_O > R_M$.

For our experimental setup, RUBiS was deployed on a quad-core server running Ubuntu Server 14.04 as the host operating system (OS), with three virtual machines (VM), also running Ubuntu, each pinned to a dedicated core. These cores were isolated, and thus not used by the host OS. The VMs were used to deploy up to three web servers with RUBiS hosted in an Apache HTTP Server. The load balancer HAProxy [63] was run in the host OS to distribute requests among the servers using the round-robin algorithm. In order to keep the latency of the tactic to add a server experimentally controlled, the server VMs were kept running at all times, and the addition and removal of a server was simulated by enabling and disabling the server in the load balancer, respectively. When the tactic to add a server was used, the execution manager enabled the server in the load balancer after a time of λ had elapsed, simulating the latency of the tactic. The tactic to remove a server disconnects the server from the load balancer, but lets it process all the requests already in, or queued for, that server. The adaptation layer (monitoring, adaptation decision, execution manager, and knowledge model) was also deployed in the host OS. A second computer was used to generate traffic to the website using requests traces previously recorded, which were replayed using a client able to make as many concurrent requests as needed to reproduce the requests according to their timestamps. All the requests target a single URL in the system, which in turn selects a random item from the auction to render its details page.

To implement the self-adaptation goal using PLA, we had to define the decision utility function $\hat{U}$, and implement the ability to generate an environment model capturing the predicted environment behavior. The latter is done as described in Section 3.5. For this case, we used a Holt's

time series predictor with autoregressive damping [70]. The monitoring component measures the average request arrival rate at the load balancer over a decision period. These observations are supplied to the predictor so that it can update its internal model, and the predictor is then used to obtain the predictions needed for building the environment DTMC (see Section 3.5 for details).

The SLA defined in (8.1) assigns utility per measurement interval, and the goal of the system is to maximize the utility it accumulates. Therefore, the decision utility function is a simple additive utility function implementing (8.1), with two additional considerations. The first one is to achieve the secondary goal of minimizing cost. To that end, $\hat{U}$ is defined so that if $U(c_1, e) = U(c_2, e)$, then $\hat{U}(c_1, e) > \hat{U}(c_2, e)$ if c_1 has lower cost. This is achieved by rounding the original utility values and scaling them so that this additional preference ordering due to cost can be represented by the values in between the original values. The second consideration is to avoid unstable solutions that would make it very difficult to regain control of the system. In particular, in a case in which all the configurations would exceed the response time threshold, the utility function would choose the one with the smallest number of servers to minimize cost. This is not the right decision because removing resources from an overloaded system would cause the backlog of requests to increase at a higher rate, making the recovery of the system in subsequent decisions even more unlikely. Therefore, an exception to this rule is included in $\hat{U}$ to favor the configuration with the most servers and lower dimmer setting in such a case.

The utility function U in (8.1) is a function of the response time the system provides. For the decision utility function $\hat{U}$, it is necessary to estimate the response time that the system would attain under a certain configuration and state of the environment. In this case, we use a queuing theory model to compute the average response time for each period based on the system configuration and the environment state. Considering that web servers can only processes a limited number of requests simultaneously while the rest are queued, we use a limited processor sharing (LPS) model [145], which considers a system in which the number of concurrent requests that can be processed by each server simultaneously is limited by a constant. We set this constant equal to the maximum number of processes configured for each Apache HTTP server in the system.

One important parameter that queuing theory needs in order to compute the estimated response time is the service rate, which is the number of requests a server can process per second. Although this parameter can be obtained through profiling, it varies throughout the execution of the system for several reasons. For example, when servers are added to the system, their cache is cold, making the service rate lower until their cache warms up. Also, there may be background processes that affect the service rate, and in cloud computing settings, the sharing of physical resources between virtual servers can also have an effect on the service rate. To make matters worse, service rate is not a parameter that is easily measurable because it is based on the time it takes to service requests without contention, something hardly achievable when the system is in normal use.

To deal with this issue, researchers have proposed using Kalman filters to estimate the value of the service rate as it changes at run time [50, 89, 146]. The high level idea is that other measurable parameters are related to this hidden parameter through a performance model. Consequently, the filter can update the estimation of the hidden parameter to reduce the error the model would have predicting the observed parameters. For example, suppose that in a simple performance model, the response time is a function of the request arrival rate, the service rate,

and the utilization of the server. Except for the service rate, all the other values can be measured. If the current estimation of the service rate is erroneous, the performance model would predict responses that differ from the measured ones. In that case, the Kalman filter updates the estimation of the service rate, to minimize this error. In addition, it also has smoothing characteristics that reduce jitter in the estimation.

For our implementation, we used OPERA, which supports performance modeling and uses a Kalman filter for parameter estimation [112]. In addition to dealing with the changing service rate due to the effects mentioned above, the use of the Kalman filter also helps to deal with the limitations of queuing theory regarding transient states. The queuing theory model predicts response time in steady state. However, when there is an increase in traffic that surpasses the current capacity of the system, a backlog of requests builds up. Until this backlog is worked off, returning to steady-state queue lengths, the observed response time is higher than normal, even if the service rate does not change. In this case, the Kalman filter will adjust the estimated service rate to match the observed reality, and this deflated service rate will, in turn, allow the queuing theory model to provide better response time estimates. As the backlog is worked off, the filter keeps updating the service rate, until it eventually matches the actual service rate when the system is in steady state.

RUBiS Simulation

In addition to using the RUBiS software, a simulation of it was also used for experimentation. Having a simulation is advantageous since it allows us to compress time, replicate experiments with exactly the same conditions, and simulate larger platforms. Time can be compressed in a simulation in two ways. There is never a real wait for something to happen, since the simulation clock can just be advanced to the next relevant event. In addition, operations that take time and whose actual result is not needed can be simulated by calculating when the operation would be completed and just inserting a completion event at that time into the future, without actually performing the time consuming operation.

To support experimentation through simulation, we developed SWIM, a simulator of web applications like RUBiS. SWIM was implemented using OMNeT++, an extensible discrete event simulation environment [139]. SWIM does not simulate the actual functionality of the web site, or what particular pages a user accesses. Instead, it only simulates processing of requests at a high level—simply as a computation that takes time to execute—which is sufficient to evaluate approaches in terms of qualities such as response times, type of responses served, and number of servers used. In addition, SWIM provides a TCP interface that allows the adaptation manager to access probes and effectors that it can use to monitor and execute adaptation actions on the system. The probes provide the following information: current dimmer value, number of servers and active servers, utilization of each server, average request arrival rate, and average throughput and response time for the two kinds of responses. The effectors allow the adaptation layer to change the dimmer setting, remove and add servers. All operations have negligible execution time, except for adding a server, which takes an amount of time configurable in the simulation. This time simulates the time it takes to boot a server, or instantiate a new VM in the cloud.

The users' requests are simulated by reading their time stamps from previously recorded traces from real websites, and replaying the traces with the requests happening with their recorded

interarrival time. The requests arrive at the load balancer, and are forwarded to one of the servers following a round robin algorithm. Each server simulates the processing of requests in the web server. The maximum number of concurrent requests in a server is configurable so that it can match the maximum number of processes configured in the real web server. When more than one request is being processed by a server, the sharing of its processor is simulated by inflating its processing time accordingly. Requests assigned to a server that is already processing the maximum number of concurrent request are queued and serviced in FIFO order.

The processing of a request is simulated only in terms of the time that it takes, not the results it produces. The service time (i.e., the amount of time processing the request would take if there is no contention) is drawn from a normal distribution truncated from below so that every request has positive service time. The server supports the brownout approach, and has a dimmer parameter that controls the probability of a response including the optional content. As it is done in the real RUBiS, when a request arrives at the server, a random number is drawn from a uniform distribution to determine whether its response should include the optional content. The service time is then drawn from a random distribution whose parameters depend on the type of response. Although the parameters of the two distributions are configurable in a flexible way, generally the service time for responses that include the optional content have higher mean and variance.

In SWIM, the effect of a cold cache is also simulated by increasing service times when a server is newly instantiated, emulating how cache misses add to the normal steady state service time. As the server processes more requests, this effect gradually disappears.

All the random number generators used in the simulation are seeded so that it is possible to replicate experiments with the same conditions, and thus make a direct comparison of different adaptation approaches. The system is available for others to use at `https://github.com/cps-sei/swim`

8.1.2 DART

This system represents a simulated team of unmanned aerial vehicles (UAVs) developed in the context of the DART (Distributed Adaptive Real-Time) Systems project at the Carnegie Mellon® Software Engineering Institute [65]. In particular, we use a scenario of DART that embodies a trade-off that a team of drones faces during a reconnaissance mission in hostile environment. Namely, there is a trade-off between detecting targets on the ground—the main purpose of the mission—and avoiding threats that could jeopardize the mission. Since the environment (i.e., the location of targets and threats) is discovered only during the execution of the mission, and even then, with some uncertainty, it is not possible to pre-plan the complete execution of the mission. Self-adaptation is required for the team to best deal with the uncertain environment.

In this system, the team of drones has a designated leader and they all fly in formation, with the leader at the center. High level decisions, such as what formation to adopt, where to fly, or whether to move the formation up or down, are made autonomously by the leader (i.e., these UAVs are not remotely piloted). The leader communicates these decisions to the rest of the team to be executed. Each drone controls its own flight, following the leader's instructions, and implements a portion of a collision avoidance protocol that allows them to fly in close proximity without colliding.

Problem Formulation

The team of drones has the following mission: to follow a planned route at constant forward speed, detecting as many targets on the ground as possible along the route. There are threats along the route that can destroy the team, so there is a trade-off between avoiding threats and detecting targets. Both targets and threats are static, but neither their number nor their location is known a priori. The team has to adapt by changing altitude and/or formation to maximize the number of targets detected, taking into account that if the formation is lost to a threat, the mission fails. The lower the team flies, the more likely it is to detect targets, but also, the more likely it is to be hit by a threat. Changing formation also involves a similar trade-off, since flying in tight formation reduces the probability of being hit by a threat, but at the same time reduces the chances of detecting targets.

We assume that the route is divided into D segments of equal length, and an adaptation decision is made periodically at the boundary between segments. Since the team flies at constant speed, there is a direct mapping between decision periods and segments (i.e., we can refer to decision period t and segment t interchangeably). Let us define the configuration $c \in C$ of the team as the pair (a, ϕ), where a is the altitude and ϕ is the formation of the team. The environment state for segment i, referred to as e_i, is a pair (ρ_i, z_i), where ρ_i is the probability that it contains a target, and z_i is the probability that it contains a threat. We discuss later how these probabilities are obtained.

Ignoring the requirement to survive for the moment, the team's goal is to adapt, changing its configuration in order to maximize the expected number of targets detected, given by

$$q = \sum_{t=1}^{D} \left(\prod_{i=1}^{t} s(c_i, e_i) \right) g(c_t, e_t) \tag{8.2}$$

where $s(c_i, e_i)$ is the probability of survival at time i when the configuration of the team is c_i and the environment is e_i; and $g(c_t, e_t)$ is the expected number of targets to be detected at time t when the team is in configuration c_t and in environment e_t.[3] The first factor in the summation in (8.2) represents the probability of the team being operational at time t, which requires having survived since the start of the mission. The probability of survival at each time is the complement of the probability of being destroyed. A threat can destroy the team only if both are in the same segment. However, a threat has range r_T, and its effectiveness is inversely proportional to the altitude of the team. In addition, the formation of the team affects the probability of it being destroyed. The team can be in two different formations: loose ($\phi = 0$), and tight ($\phi = 1$).[4] The latter reduces the probability of being destroyed [140] by a factor of ψ. Taking altitude and formation into account, the probability of the team with configuration c being destroyed is[5]

$$d(c) = \left((1 - \phi(c)) + \frac{\phi(c)}{\psi} \right) \frac{\max\big(0, r_T - a(c)\big)}{r_T} \tag{8.3}$$

[3] In this scenario there is at most one target per segment, so $g(c_t, e_t)$ is, effectively, the probability of detecting a target at time t.

[4] The time required to change formations is assumed to be negligible. Consequently, no intermediate formation states are considered.

[5] We use $a(c)$ and $\phi(c)$ to refer to the properties of configuration c.

If the probability of segment t having a threat is $z(e_t)$,[6] the probability of the team surviving in segment t is

$$s(c_t, e_t) = 1 - z(e_t)d(c_t)$$

The expected number of targets found in a segment depends on the probability of a target being there and the configuration of the team. The probability of detecting a target given that the target is in the current segment is inversely proportional to the altitude of the team [135]. In addition, flying in tight formation reduces the detection probability due to sensor occlusion or overlap. The expected number of detected targets in segment t with environment e_t and configuration c_t is

$$g(c_t, e_t) = \rho(e_t) \left((1 - \phi(c_t)) + \frac{\phi(c_t)}{\sigma} \right) \frac{\max\big(0, r_S - a(c_t)\big)}{r_S}$$

where r_S is the range of the sensor (i.e., at an altitude of r_S or higher, it is not possible to detect targets), and σ is the factor by which the detection probability is reduced due to flying in tight formation.

The expected number of targets detected during the whole mission according to (8.2) takes into account the fact that targets can by detected only as long as the team has not been hit by a threat. However, it does not express any survivability requirement. As it is, it may happen that, upon not seeing any more targets ahead for the remainder of the mission, the team commits suicide, given that surviving will not help it detect more targets. One possible way to solve this issue is to include a reward for surviving the mission. However, the reward has to be calibrated with respect to the number of targets that could be detected during the mission. A reward too large would cause the team to be very risk-avoiding; but if it is too small, it will make the team take riskier behavior for a better chance at detecting targets. Given that the number of targets is not known a priori, this reward is very difficult to calibrate. Furthermore, different missions may require different trade-offs between target detection and survivability, and having to express the trade-off by balancing rewards would be unwieldy.

Instead, we want to explicitly include the survivability requirement in the adaptation goal. Using one of the adaptation goal combinations introduced in Chapter 6, namely {RG3, CS2}, we can express the DART adaptation problem over the decision horizon as

$$\begin{aligned} \underset{c_1,\ldots,c_H}{\text{maximize}} \quad & \sum_{t=1}^{H} \left(\prod_{i=1}^{t} s(c_i, e_i) \right) g(c_t, e_t) \\ \text{subject to} \quad & \prod_{t=1}^{H} s(c_t, e_t) \geq P \end{aligned} \tag{8.4}$$

That is, targets can be detected as long as the survivability constraint is satisfied, and the probability of surviving must be at least P.

[6] We use $\rho(e)$ and $z(e)$ to refer to the properties of environment state e.

Environment Monitoring and Model

There could be various ways to obtain the probabilities that represent the environment state. For example, these probabilities could be assigned based on the expert opinion of some operator, or using previously gathered intelligence. For this scenario, we assume that the team has a way of sensing the environment ahead with some finite horizon equal to the decision horizon. More concretely, the team has a low quality sensor[7] that makes observations of the segments ahead, giving an output of 1 for the detection of a threat in a segment, and 0 otherwise. The same is done for targets, and we assume that targets and threats can be sensed independently, through the use of two different sensors, or with a single sensor that can distinguish between both types of objects. In each monitoring interval, n samples are taken for each segment in the horizon of length H. The observations are accumulated over the monitoring intervals, so that when a segment first enters the look-ahead horizon, it will obtain n observations. In general, segment i in the horizon (with $i = 0$) for the current segment, will have $n(H - i + 1)$ samples. In order to build and maintain an environment model, it is necessary to keep track of two numbers for each segment and each object type: the number of detections, and the number of non-detections (or equivalently, the number of samples taken, and the number of detections).

To build the environment model, we assume that the two random variables in the environment state (i.e., probability of segment containing a threat, and a target, respectively) are independent. Given that, we can construct two independent environment models, and then join them to produce the joint environment model. We first describe how the independent threat and target environment models are created, and then explain how they are joined.

Since the team visits one segment per decision interval, t time steps into the future, the team will be t segments further down the route. Using the information captured by the environment monitoring, we can describe the probability density for ρ_t and z_t using the Beta distribution [12]. For each segment, the number of detections and non-detections correspond to the parameters α and β of the Beta distribution, respectively. This continuous distribution can then be discretized using the Extended Pearson-Tukey (EP-T) three-point approximation [80], allowing us to consider three possible realizations of the environment for each segment. We assume no dependencies between the state of different segments. Instead, we assume that a given state of the environment at time $t + 1$ can be reached with equal probability from every possible state at time t. Consequently, creating a tree to represent the evolution of the environment would result in unnecessary replication of environment states. To avoid that, we represent the environment model as a DTMC with the topology exemplified in Figure 8.1 for a model of the threats with a horizon of length 3. The root corresponds to the state of the environment at the current time/segment. Since the environment model only covers the lookahead-horizon for the adaptation decision, we refer to the current time as $t = 0$. For each value of $t \in 1, \ldots, H$ there are three nodes corresponding to the three-point approximation of the distribution of the environment state at time t. Every node for $t < H$ has one edge going to each of the nodes that correspond to $t + 1$, with their probabilities set according to the EP-T discretization.

Once the independent environment models for the threats and the targets have been created as described above, the joint environment model is built by generating every possible combination of threat and target environment states, and creating the edges accordingly. Note that only com-

[7]The sensor has false positive and false negative rates (FPR, and FNR, respectively) greater than zero.

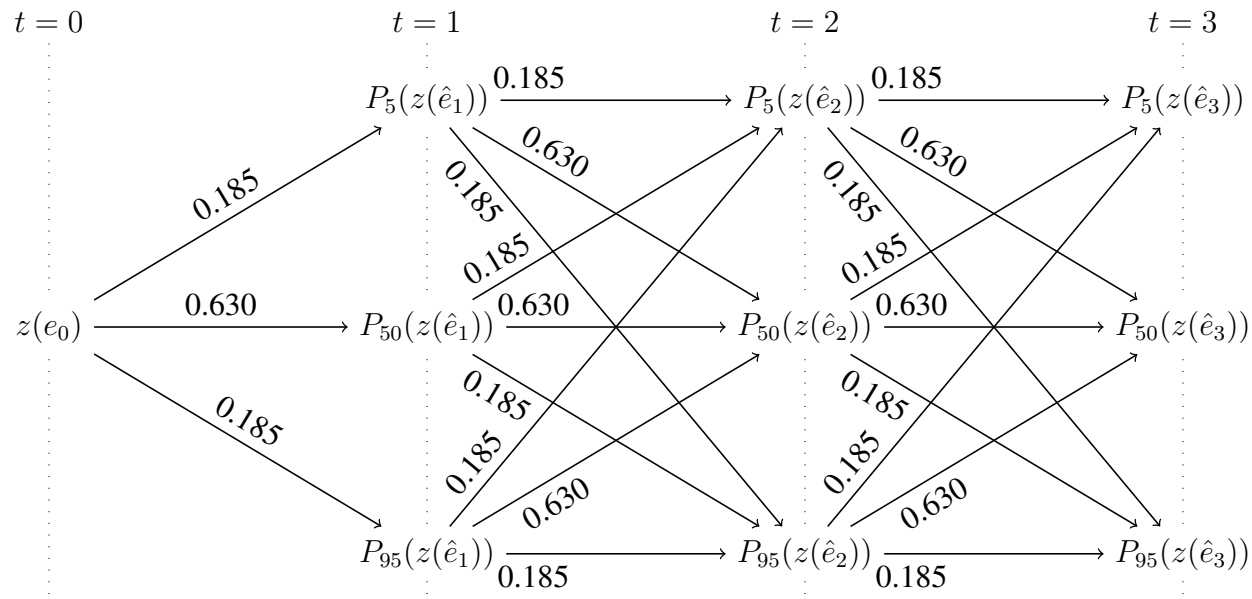


Figure 8.1: DART environment model for threats ($H = 3$).

binations of states that correspond to the same time are feasible, and this fact can be exploited to reduce the state space. Therefore, the joint model will have a single root node, and for each $t \in 1, \ldots, H$ it will have 9 nodes.

8.2 Claims Validation

In this section we present the results of using the proposed approaches as evidence to validate the claims of the thesis, organizing their presentation by claim.

8.2.1 Effectiveness Improvement

The main claim of this thesis is that the proposed approach improves the effectiveness of self-adaptation. The validation of this claim is done by comparing the effectiveness of PLA (using either of the two main solution approaches) with a self-adaptation approach that lacks the timing aspects of PLA (i.e., it is not proactive and is latency-agnostic). The comparison is done in terms of the adaptation goals of each system.

RUBiS

For validating the effectiveness improvement claim with RUBiS, we present two sets of results. The first is based on runs of the actual RUBiS implementation,[8] and the second is based on the RUBiS simulation. In both cases, we use as a baseline for comparison a feed-forward (FF) self-adaptation approach that is latency-agnostic. FF uses a single-point one-step-ahead prediction of

[8]These results were described in our original publication of PLA-SDP [109].

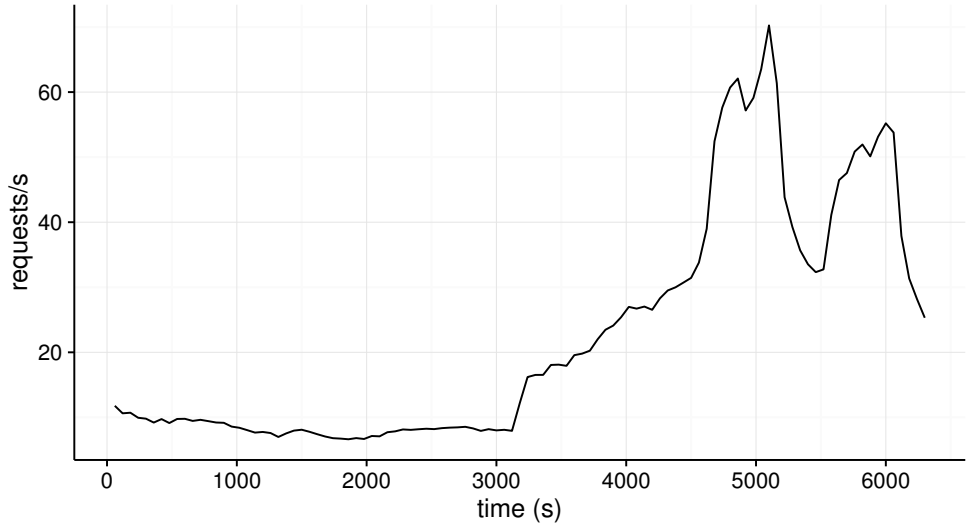
the request rate to select the adaptation tactic(s) that would result in the highest utility, assuming that tactics are instantaneous, and not looking beyond the current decision.

The period for the adaptation layer (i.e., the monitoring and adaptation interval) was $\tau = 60$ seconds. The length of the look-ahead horizon used for the adaptation decision was computed as $H = \max\left(5, \left\lceil \frac{\lambda}{\tau} \right\rceil (S_{max} - 1) + 1\right)$, where $S_{max} = 3$ is the maximum number of servers. In this way, the horizon is long enough for the system to go from one server to S_{max}, with an additional time interval to observe the benefit. A minimum of 5 intervals enforces look-ahead even if the tactic latency is small. The parameters of the utility function were set as follows: response time threshold $T = 0.75$ seconds; rewards for responses with mandatory and optional $R_M = 1$, $R_O = 1.5$ respectively; and maximum system capacity $\kappa = 67.4$ requests per second (this value was obtained through profiling). The adaptation tactics could change the number of servers between 1 and S_{max}, and the dimmer among the values 0.10, 0.30, 0.50, 0.70, and 0.90.

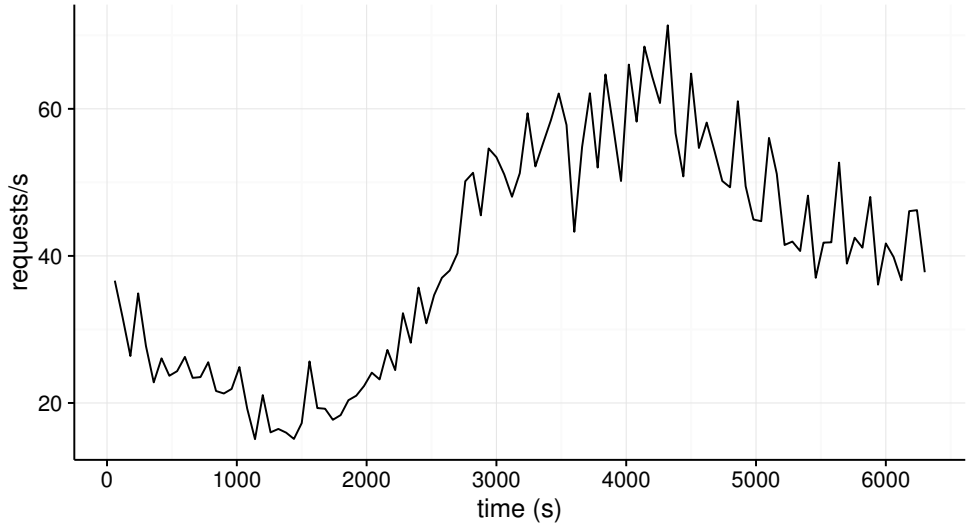
The stream of requests to the system was generated from publicly available traces captured from real websites. Specifically, we used half of a day from the WorldCup '98 trace [4], and one day from the ClarkNet trace [5]. As it can be observed in the plots of the traces in Figure 8.2, these traces have different characteristics. The WorldCup '98 trace presents large spikes in traffic, whereas the ClarkNet trace is bursty. Even though these traces do not correspond to an auctions website, the point of using them is to exercise the system with realistic traffic patterns, and not to replicate the behavior of users of an specific kind website. Also, even though these traces are several years old, they still represent a significant traffic for our validation platform. In fact, both traces were scaled down to last for 105 minutes, and to reach the maximum capacity of the validation setup at their peak.

For each approach we ran the system four times, each with a different latency for the tactic to add a server ($\lambda = 60$, 120, 180, and 240 seconds). For each run, the first 15 minutes were used to let the system warm up with no adaptation. This allows the time series predictor to be primed, and the estimation of the service time tracked by the Kalman filter to adjust. Self-adaptation was used during the remaining 1.5 hours of the run, during which the metrics were collected.

The results of the comparison of the PLA approaches with FF are shown in Figures 8.3 and 8.4 for the two traces respectively, with the arrows in the side labels pointing in the direction of better outcomes. We can observe that with the FF approach, the utility provided by the system drops as the tactic latency gets larger, whereas the PLA approaches are able to maintain the level of utility despite the increased latency. The effectiveness improvements that PLA brings for the case with the largest latency are 65% and 60%, for each trace respectively. Additionally, we show other metrics that, even though are not the main criteria for adaptation, are interesting to observe. The FF approach provides more responses with optional content. This is understandable because a latency-agnostic approach ignores the fact that the tactic to change the dimmer is much faster than the tactic to add a server, thus favoring the latter to deal with an increase in request rate, expecting to get a higher reward. However, the percentage of responses that do not meet the response time requirement increases with latency when latency is ignored, resulting in penalties instead. The PLA approaches, on the other hand, are able to keep the percentage of late responses very low in spite of the increase in tactic latency. The charts plotting the average utility per server show that despite using more servers, the PLA approaches obtain more utility per server for larger tactic latencies.



(a) WorldCup '98



(b) ClarkNet

Figure 8.2: Traces used for workload generation.

In these experiments, both PLA-PMC and PLA-SDP produced similar results. The slight difference between the two is due to disturbances in the runs, such as network delays, and background processes that are not possible to control when running the real system. However, when run in a simulation with exact replication of environmental conditions, PLA-PMC and PLA-SDP produce exactly the same results. Figure 8.5 shows the results for the same experiment run in the RUBiS simulation instead. This shows that PLA-SDP produces precisely the same results as PLA-PMC in spite of the considerable speedup it achieves, as shown in Figure 5.5.

The rest of the results presented for RUBiS are based on its simulation, which allows us to easily introduce variances to the adaptation approaches and test scalability beyond what our real experimental platform supports. Comparing Figures 8.4 and 8.5 shows that the results obtained with the simulation are similar to those obtained by running the experiments in the real system.

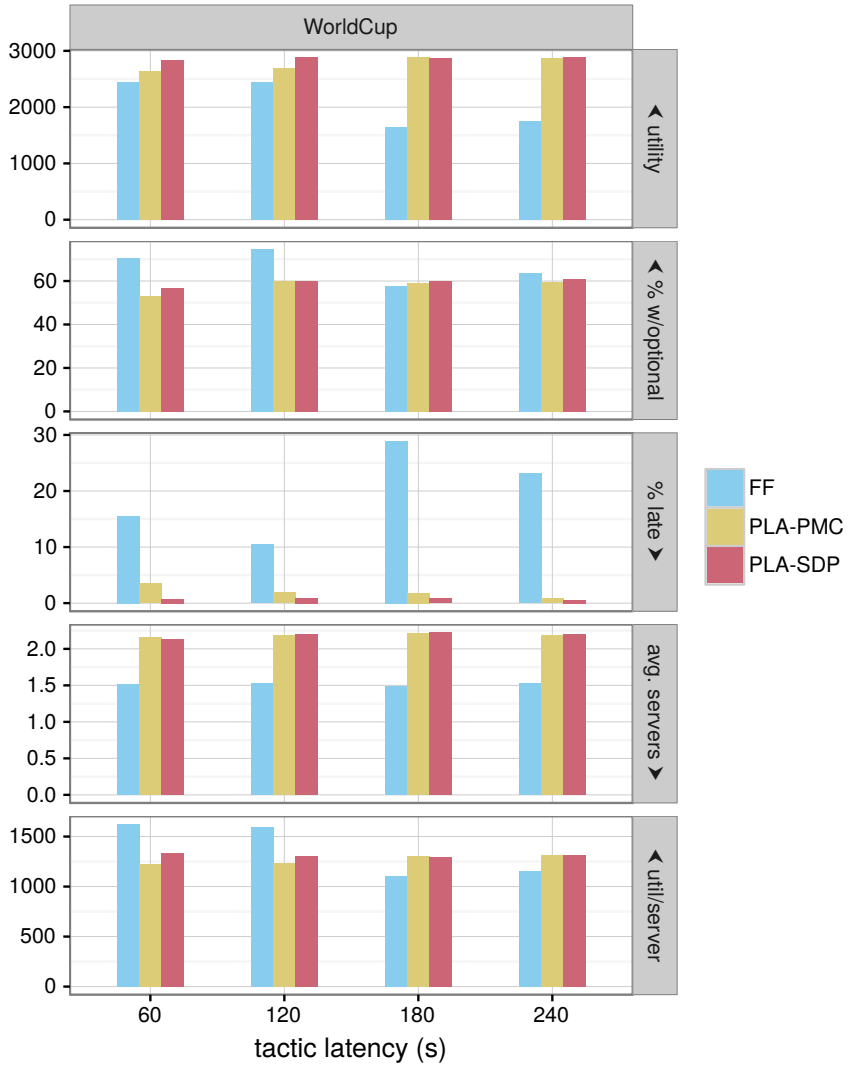


Figure 8.3: Comparison of approaches in RUBiS with WorldCup '98 trace.

Thus, we feel confident that the results for RUBiS presented in the rest of this chapter, which are based on simulation, are similar to what experimentation on the real system would produce.

DART

The following results are based on 5,000 runs of the DART simulation with each approach.[9] For each run, there were 20 targets and 7 threats randomly placed along a route of 100 segments. In addition, there are several other random behaviors in the simulation of the mission. The forward-looking sensors and the target sensor are subject to random effects, as is the impact of threats on the drones. For example, to determine if the team is destroyed while traversing a segment with a threat, a random number d' is drawn from a uniform distribution with range $[0, 1]$, and if $d' < d(c)$ (see (8.3)), the team is destroyed in the simulation. Despite the large number of runs,

[9]Since PLA-PMC and PLA-SDP produce the same results, only the latter was used for these experiments.

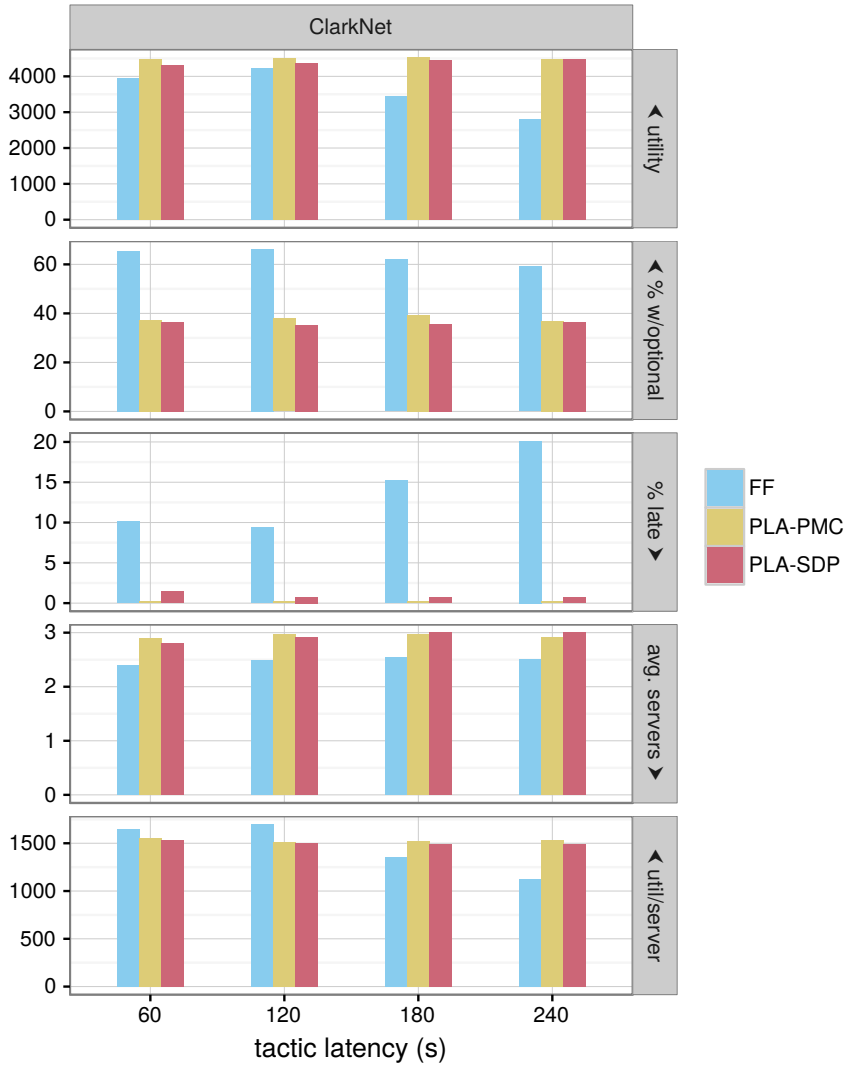


Figure 8.4: Comparison of approaches in RUBiS with ClarkNet trace.

for better comparison, the random number generators that control all of these random effects were seeded with matching seeds for the same run number of the two solution approaches. In that way, both approaches faced the same behavior of the environment.

The tactics used in this simulation control the altitude and the formation of the team. There are tactics to increase and decrease the altitude level at which the drones fly. The airspace is divided vertically into 10 altitude levels. Transitioning from one level to the next, up or down, takes the same time the team takes to traverse a segment in the route. Therefore, the latency of these tactics is τ; that is, the same as the decision interval. There are two other tactics that allow the team to change formation. One takes the team to a close formation, and the other to a tight formation. These two tactics are assumed to be immediate, or have negligible latency compared to the decision interval.

As a baseline for comparison and measurement of the effectiveness improvement, we also use a feed-forward approach (FF), which is latency-agnostic, and not proactive, in that it does

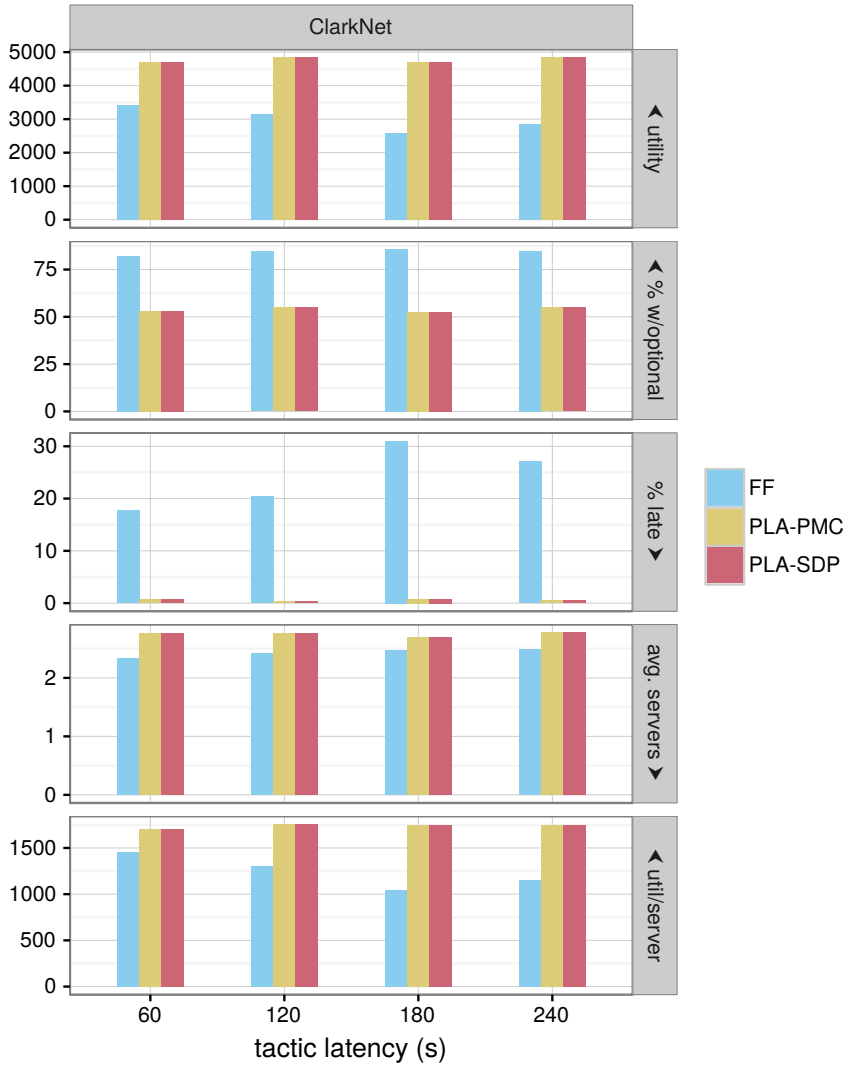


Figure 8.5: Comparison of approaches in RUBiS with ClarkNet trace (simulation).

not look ahead further than the segment it is about to enter. More precisely, it uses a single-point estimation of the environment state in the segment that it is about to enter when making the adaptation decision. Given the adaptation goal for this system, the PLA approach used in these experiments is PLA-SDP as extended in Chapter 6—namely, using the {RG3, CS2} combination. PLA-SDP was run with a horizon $H = 5$. For these experiments, the probability bound for the survivability constraint was set to $P = 0.90$. It is worth noting that the FF approach was implemented to also consider the survivability requirement in addition to the maximization of detected targets. This consideration, however, is limited to the period starting at the time the decision is being made, given the lack of look-ahead in FF.

Figure 8.6 shows the statistics for the number of targets detected with each approach. PLA-SDP detected the highest number of targets on average, and in addition it has less variance. The average improvement of PLA-SDP over FF in the 5,000 missions simulated was 74%, with 3.5 more targets detected per mission on average.

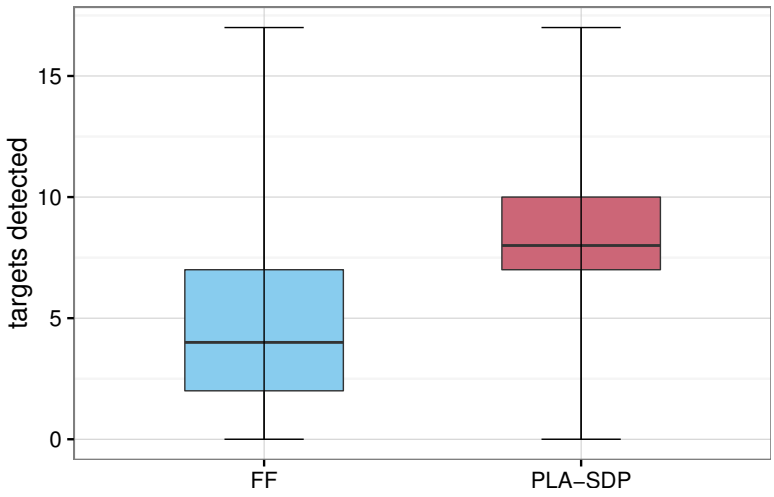


Figure 8.6: Targets detected in DART.

Figure 8.7 shows the proportion of missions in which the team survived with each approach. Here, the difference is impressive, with FF surviving less than 10% of the missions, while PLA-SDP satisfied the requirement of surviving with at least 90% probability.

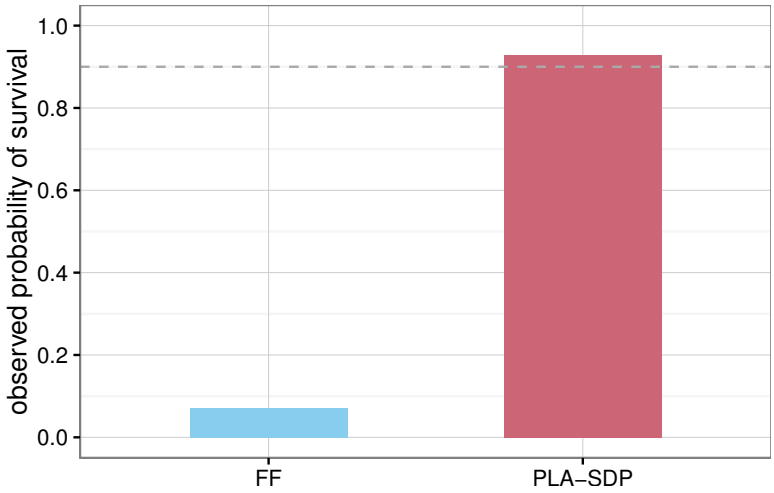


Figure 8.7: Probability of mission survival in DART.

The results shown in Figure 8.6 include targets detected even in missions in which the team failed to survive. However, depending on the drones and/or the mission, the team may not be able to transmit detected targets back to the base, requiring the team to complete the mission before the data can be downloaded. Adjusting the target count so that targets detected are counted only if the drones survive the mission, Figure 8.8 shows that the difference in effectiveness is even more pronounced. In this case, even the 3rd quartile for FF is 0 because it only survived and detected targets in 7% of the missions.

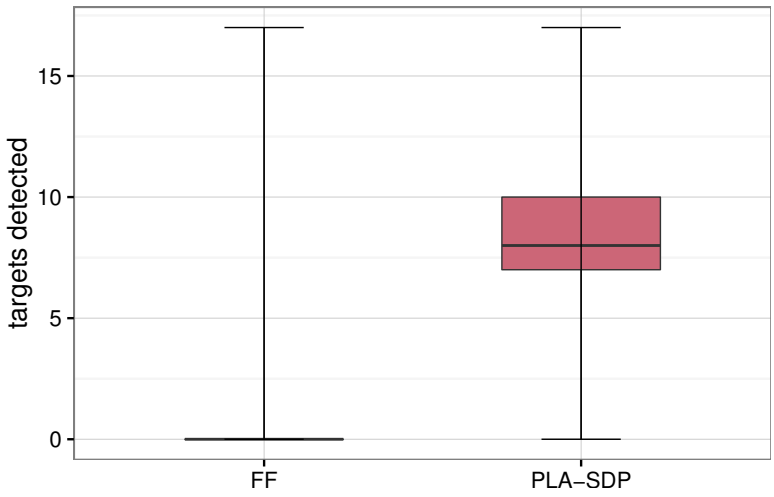


Figure 8.8: Targets detected in DART adjusted for mission survival.

8.2.2 Applicable to Different Kinds of Systems

To validate this claim, we provide evidence supporting applicability to systems that are different in several key ways. The two systems used for the validation were chosen with two main criteria. First, they had to conform to the basic assumptions of PLA enumerated in Chapter 3 and repeated here: (i) that the adaptation goal can be expressed with one of the utility forms described in Chapter 6; (ii) that it is possible to estimate the system measures of performance that are used to compute utility; (iii) that tactics have deterministic effect on the structure and properties of the system; and (iv) that the actions of the system do not affect the evolution of the environment.[10] Second, the systems had to be different in several important ways. Table 8.1 summarizes the differences between the two systems used for the validation, and we elaborate on them here.

These systems belong to two very different domains. RUBiS is an information system, in which the adaptation is driven mainly by quality of service, revenue, and cost. The tactics in this system deal with the allocation of resources, and the control of the optional content served, which drive revenue and quality of service in opposite directions. The environment for this system is determined by the users of the system, reflected as the rate with which user requests are sent to the system. In this case, environment predictions are made using forecasting based on past observations of the request arrival rate collected at run time through the monitoring component. These observations are then processed by a time series predictor to compute the predictions. DART, on the other hand, is a cyber-physical system, with tactics that affect its physical properties (i.e., altitude and formation) and its configuration (i.e., turning electronic countermeasures (ECM) on and off). In this system, adaptation is driven by mission objectives, which include detecting targets in a hostile environment, and the requirement to survive with a specified probability in order to complete the mission. The environment in DART is based on physical elements, targets and threats with unknown locations. Contrary to what happens in RUBiS, observations of the past environment are useless in DART. Therefore, the environment predictions come from sensors that sense the world ahead of the team as it is flying. Since these

[10] These assumptions and their potential limitations are discussed in Section 9.3.

sensors have lower quality when sensing far ahead, there is uncertainty in the predictions of the environment that self-adaptation must deal with.

These systems have different adaptation goals. In RUBiS, the goal is to maximize the utility according to a SLA, while minimizing cost. In DART, the goal is to maximize the number of targets detected in a mission, while keeping the probability of surviving the mission above a given requirement. In addition, in Chapter 6, we provided examples of systems with different adaptation goals that are also supported by the approach.

The approach is not tied to a particular self-adaptation framework as shown by the different implementations of the systems used for validation. The core of the approach—the decision-making—was implemented in C++ and packaged in a library, which was used in all the implementations. The library allows the user to provide the utility function and the configuration space that is used for the adaptation decisions. Self-adaptation for RUBiS was implemented in two different ways. In one implementation, the PLA self-adaptation loop was implemented as modules in the OMNeT++ framework, with the adaptation decision module invoking the library. Even though OMNeT++ is geared towards creating discrete-event simulations, its scheduler can be replaced so that events are processed in real-clock time instead of simulated time. In fact, this implementation of the adaptation manager was used to control both the real RUBiS and its simulation by having two sets of monitoring probes and effectors with the same interface, allowing it to interact with both the real system and the simulation.

The second implementation of PLA self-adaptation for RUBiS was done using the Rainbow framework for self-adaptation. Since Rainbow is implemented in Java, a wrapper for the library was generated using SWIG, a tool that automatically generates wrappers for C++ code [134].[11] The different customization points that Rainbow provides (see Figure 7.1) were used to implement PLA adaptation for RUBiS.[12] An adaptation manager class was created to make adaptation decisions using the library, and custom probes and effectors were used to monitor the system, and to execute the operations needed by the adaptation tactics.

Self-adaptation for DART was implemented also in two different ways. One was using the DART architecture and its tool chain [65]. In this case, the self-adaptation loop was implemented in DMPL [24], invoking the library to make adaptation decisions. In the second implementation, a custom PLA self-adaptation loop was implemented directly in C++.

The application of PLA self-adaptation to two systems that are different along several dimensions provides a strong argument that our approaches are applicable to different kinds of systems.

8.2.3 Scales to Systems of Realistic Size

To validate this claim, we conducted two experiments: one to show that it can handle adaptation decisions in a system with a large number of servers, capable of handling realistic traffic; and another one that shows that the DART adaptation decisions can be done fast enough even in a computer of modest performance used in small real drones. The results presented in this section

[11] Since SWIG can generates wrappers in several languages including Python, C#, and Perl, it would be possible to use the library in the same way with frameworks written in other languages.

[12] Special thanks to Bradley Schmerl who modified Rainbow to support concurrent tactic execution.

are based on PLA-SDP, the fastest of the two main solution approaches. In Section 7.4 we showed that SB-PLA is more scalable than PLA-SDP.

RUBiS in Large Server Cluster

To show PLA can handle a large system, we simulated RUBiS with the trace for day 51 (June 15, 1998) of the WorldCup '98 dataset, using 18 hours of the traffic directed to one regional cluster. This trace was not scaled down, and given the capacity of the simulated servers, we set the maximum numbers of servers to 60. There were three games played on that day, so the trace shows three main peaks in traffic (see top chart in Figure 8.9).

For this experiment the latency of the tactic to add a server was 180 seconds, and the look-ahead horizon was set to 10. The average decision time was 8.2 seconds, much shorter than the decision interval of 60 seconds. We can see in Figure 8.9 that in spite of the decision time not being negligible, PLA-SDP had only four violations of the response time requirement.

To provide contrast in terms of the effectiveness of the approach at this scale, we also ran this experiment using the FF approach, obtaining the results shown in Figure 8.10. We can see that FF had many more violations of the response time requirement, incurring in many penalties as defined by the SLA. Since FF is not able to assess the future impact of making several successive server additions, it relies much more on the dimmer, and uses fewer servers in general.

In case this limitation of FF was unfair for comparison, we also ran this experiment with a purely reactive approach that works as follows. If the measured response time is above the requirement, it tries to add a server, and if it cannot do that either because it has maxed out the cluster or because another server is being added, it reduces the dimmer if possible. If the response time is below the requirement and it has at least two servers of spare capacity, it increases the dimmer if it can, and if the dimmer is already at its maximum, it removes a server. In addition, it keeps more than one server of extra capacity up at all times.[13] With this adaptation approach, the decisions are not biased by the limitation of not being able to project the future benefit of an action (although it does not consider the future either). As shown in Figure 8.11, the results are not better than with FF, with a large percentage of late responses and using many more servers on average than FF. The results of this experiment with a large number of servers are summarized in Table 8.2.

Table 8.2: Comparison of approaches in large cluster simulation of RUBiS.

Approach	Utility	% Optional	% Late	Avg. Servers
PLA-SDP	576385	42	0.1	22.8
FF	241366	25.8	3	7.6
Reactive	62227	76	25.6	17.5

[13]This adaptation approach was designed to avoid reliance on tactic impact estimation. That is why predefined recipes for adaptation were encoded to deal with each condition; for example, *add a server if possible; if not, reduce the dimmer*. These are akin to strategies in Rainbow.

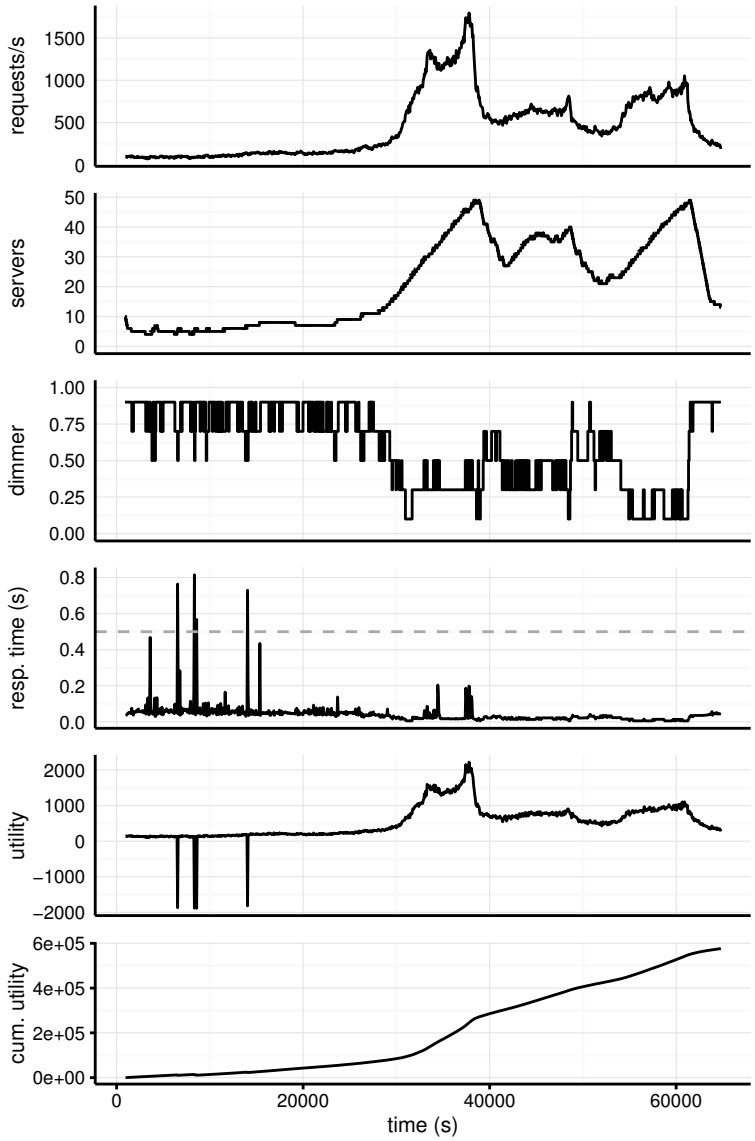


Figure 8.9: RUBiS simulation of 18 hours of traffic for a whole regional cluster of the WorldCup '98 website using PLA-SDP.

DART in Embedded Computer

To show that adaptation decisions can be computed sufficiently fast in an embedded computer with modest processing power, we ran the PLA-SDP adaptation manager for DART in a Raspberry Pi 3, the onboard computer in the research drones used in the DART project. In order to determine what the decision time requirement should be, we searched the published literature of similar missions to obtain concrete measures. Kim et al. divide the mission space into cells with a side of 2 km for UAVs with a speed of 300 km/h, which results in 24 seconds to traverse a cell [82]. Flint et al. use a planning step of 30 seconds for their approach for controlling UAVs searching for targets [45]. The most stringent requirement was that found in Baker et al., where

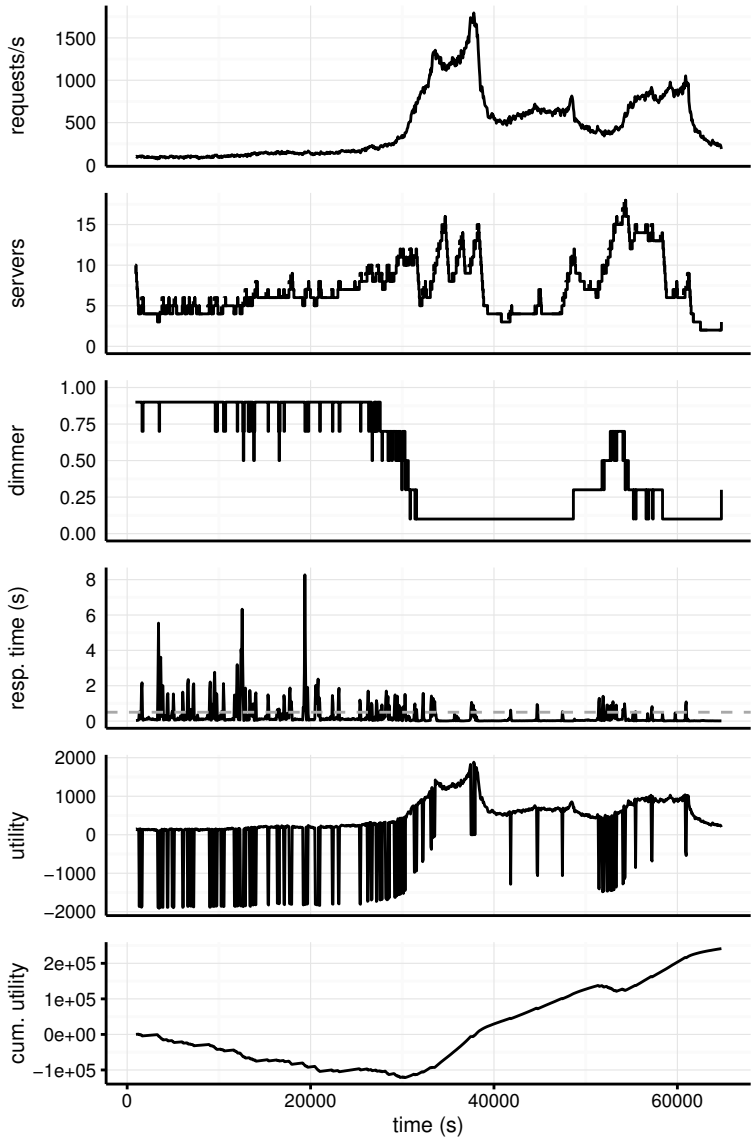


Figure 8.10: RUBiS simulation of 18 hours of traffic for a whole regional cluster of the WorldCup '98 website using FF.

the search map is divided into cells 10 m wide, with UAVs flying at a speed of 10 m/s [8]. In this case, a new cell is entered every second.

With a new cell (or segment) entered every second, the system has to be able to make an adaptation decision in less than a second. In our experiments, the average decision time was 113 ms, and even if we add two more tactics to turn ECM on and off, the decision time is 494 ms. Note that these results were obtained running in a very modest onboard computer, considering that other drones, even small research drones, have much more processing power. For example, the AscTec Firefly[14] has an Intel Core i7 processor with speeds up to 3.1 GHz compared to the

[14] `http://www.asctec.de/en/uav-uas-drones-rpas-roav/asctec-firefly/`

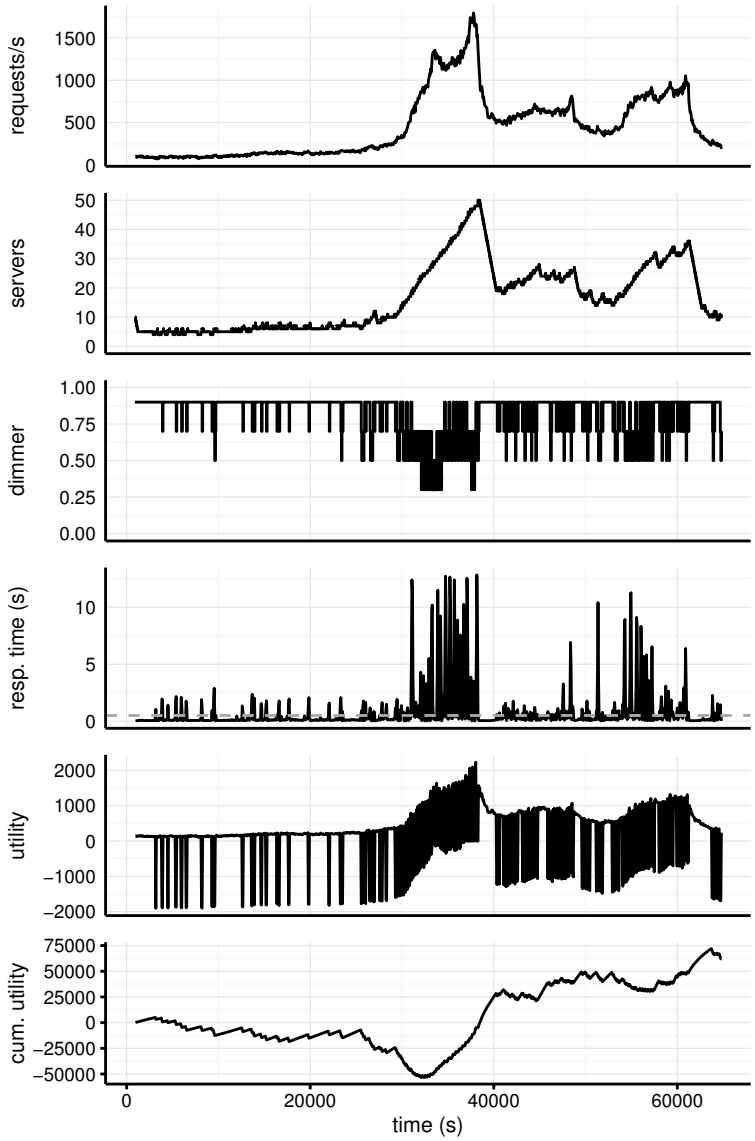


Figure 8.11: RUBiS simulation of 18 hours of traffic for a whole regional cluster of the WorldCup'98 website using Reactive adaptation.

1.2 GHz processor of the Raspberry Pi 3. Therefore, we can conclude that PLA-SDP can make adaptation decisions fast enough for using it in a real drone.

8.3 Summary

In this chapter we have presented results that support the claims of the thesis. We have shown that PLA consistently improves the effectiveness of self-adaptation when compared to an approach that lacks the timing aspects of PLA (i.e., it is not proactive and is latency-agnostic). PLA was applied to two systems that are different in significant ways. In addition, we showed that it

scales to large systems, and it is sufficiently fast to make adaptation decisions using embedded computers.

Chapter 9

Discussion and Future Work

In this thesis we have presented a conceptual framework for proactive latency-aware adaptation, and three different solution approaches that implement those concepts. The key pillars of PLA are (i) leveraging predictions of the near future state of the environment to adapt proactively; (ii) considering the latency of adaptation tactics when deciding how to adapt; and (iii) executing tactics concurrently in PLA-SDP and PLA-PMC. In previous chapters we have demonstrated that PLA improves the effectiveness of self-adaptation in different domains, using different adaptation frameworks. In this chapter we discuss how the combination of the three pillars is better than any one of them individually, and why the two main solution approaches presented are both relevant and needed. Also, we analyze the limitations of the approach, and discuss future work.

9.1 Analysis of the Contributions of the Elements of the Approach

The key elements of PLA are proactivity, latency-awareness, and concurrent tactic execution. In Chapter 8, we have shown how PLA improves the effectiveness of self-adaptation. However, it is worth analyzing if it would be possible to attain the same improvement without having all these three elements in the approach. To answer this question, we developed a suite of adaptation managers that had different combinations of these elements, as shown in Table 9.1.[1]

Using these adaptation managers, we ran simulations of RUBiS for 18 hours of the WorldCup trace for one whole regional cluster with 20 servers. The resulting utility obtained with the different approaches for different latencies of the tactic to add a server are shown in Figure 9.1. In this chart we can clearly observe two things. One is that proactivity by itself provides a substantial improvement, and the second is that the full approach, PLA-SDP, is the only one that consistently gives the highest utility, except for being slightly worse for a latency of 60 seconds—we will come back to that. The other behaviors may seem odd at first, such as why adding concurrency to a proactive non-latency-aware approach makes it worse. By analyzing

[1] Latency-awareness requires proactivity, thus, there are no latency-aware adaptation managers without proactivity.

Table 9.1: Adaptation managers with different combinations of the PLA elements.

	proactive	**latency-aware**	**concurrency**
FF			
P-NLA-NC	✓		
P-NLA	✓		✓
PLA-NC	✓	✓	
PLA-SDP	✓	✓	✓

the traces of the behavior of the approaches in these experiments, we arrived at the following conclusions:

- P-NLA does worse than P-NLA-NC because when we add concurrency, it uses the dimmer more because it has more opportunities to do so, especially while a server is being added and the only tactic available is the dimmer. This causes a loss of utility because lowering the dimmer results in less reward. Since this approach ignores latency, it still incurs penalties for assuming that servers will become available faster than they do.
- Which of P-NLA and PLA-NC does better depends on the tactic latency. When we add latency-awareness to proactivity, but without concurrency support, it will avoid tactics with long latency because it is aware that if it starts a server addition, it will not be able to deal with environment changes while the tactic is running. Therefore, it seems that unless there is a large penalty for using the dimmer, PLA-NC prefers to use it instead of committing to starting a tactic with long latency in order to avoid being in a situation in which it cannot adapt. All of the approaches make optimistic estimations of the response time they will attain after adding a server, since the effects of a cold cache are not fully modeled. This possibly results in penalties due to high response time. However, since PLA-NC tends to avoid server additions and removals, it may obtain an advantage by avoiding these estimation errors. That is why it performs very well, and even slightly better than PLA-SDP for low latency. However, for larger latencies, that advantage seems to be canceled out by the penalties incurred by avoiding to add a server, which when absolutely necessary, will take a considerable amount of time.

Using the full PLA-SDP with the three elements is the only choice that consistently gives the highest utility. If for some reason it were not possible to use the full approach, the second best choice would be to use proactivity alone. However, it depends on the available tactics and how their use affects the utility attained by the system. For example, in the case of RUBiS, the less difference in reward there is between serving and not serving the optional content, the less consequential it is to use the dimmer in terms of how it affects the reward. Nevertheless, the dimmer still provides an option to avoid penalties. In that case, for example, the PLA-NC approach would not suffer the penalties of relying too much on the dimmer and it would perform better than it did in these experiments. Still, using the full approach removes all those unknowns, because it is able to adjust to differences in the utility function, relying more or less on concurrency as it deems appropriate.

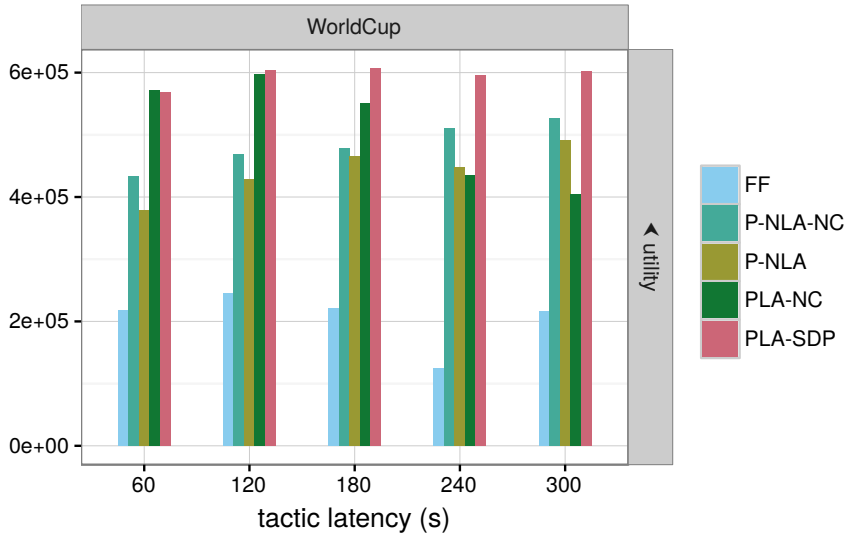


Figure 9.1: Comparison of partial approaches in RUBiS with large WorldCup '98 trace (simulation).

We also ran a similar experiment with DART, using the different adaptation managers. In this case, all the adaptation managers consider the survivability requirement in addition to the maximization of detected targets. Figure 9.2 shows that proactivity brings some improvement over FF in terms of targets detected. However, when latency-awareness is added, the adaptation performs the best. In this case, having concurrency does not make a difference, probably because the tactics with latency in this example have a latency equal to one decision period, so concurrency does not provide the ability to execute another tactic while a longer one is executing, because by the time a new decision is made, the tactic with latency has always completed. Similar results were obtained with regards to the probability of surviving the mission, as shown in Figure 9.3. Both PLA-NC and PLA-SDP satisfy the survivability requirement. Even though PLA-NC has a higher probability of mission survival, Figure 9.4 shows that it had more missions in which it detected fewer targets than PLA-SDP (as indicated by the lower 1^{st} quartile). Therefore, PLA-NC was more risk-avoiding than was necessary to satisfy the survivability requirement. Again, in the case of DART, using the full approach provides the largest improvement in the effectiveness of self-adaptation.

9.2 The Rationale for Two Main Solutions Approaches

In this thesis we have presented two main solution approaches to PLA self-adaptation, PLA-SDP and PLA-PMC. In Section 1.3, we argued that having both was desirable because PLA-PMC was modifiable, while PLA-SDP was much faster. Now, we revisit that discussion in light of the contents of this dissertation, and present the argument considering how the two approaches are similar or different, but complementary, along different dimensions, as summarized in Table 9.2.

Both approaches take into account the uncertainty of the environment when deciding how to adapt. Although this seems irrelevant now, given that both are equally good at it, it is worth

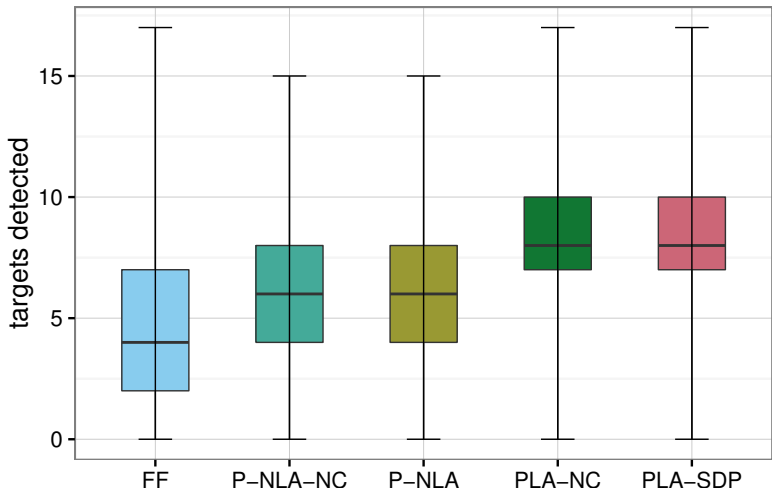


Figure 9.2: Comparison of target detection in DART with partial approaches.

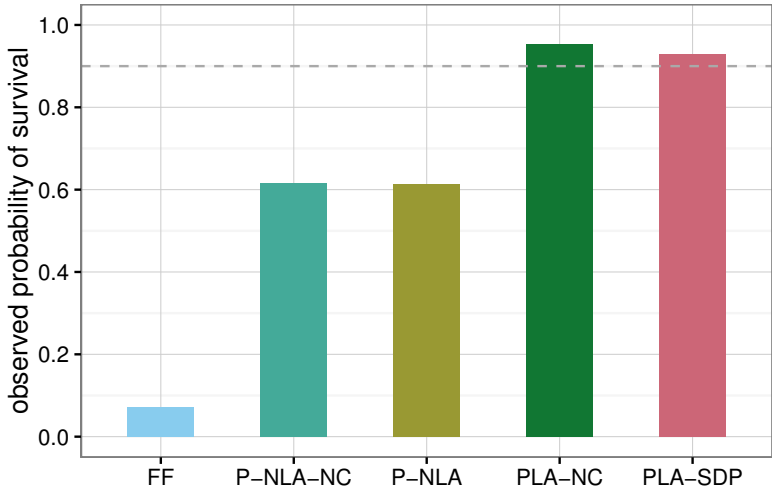


Figure 9.3: Comparison of probability of mission survival in DART with partial approaches.

Table 9.2: Comparison of PLA-SDP and PLA-PMC solutions approaches.

Feature	PLA-SDP	PLA-PMC
handles environment uncertainty	yes	yes
flexibility to change adaptation goal	limited	yes
optimal	yes	yes
fast	yes	no
can invoke external code	yes	no
easy to modify	no	yes

pointing out for two reasons. First, probabilistic model checking naturally handles probabilistic behavior, and that was one of the main reasons it was considered first as a solution approach. PLA-SDP, on the other hand, had to be designed to have that capability, not just by using prin-

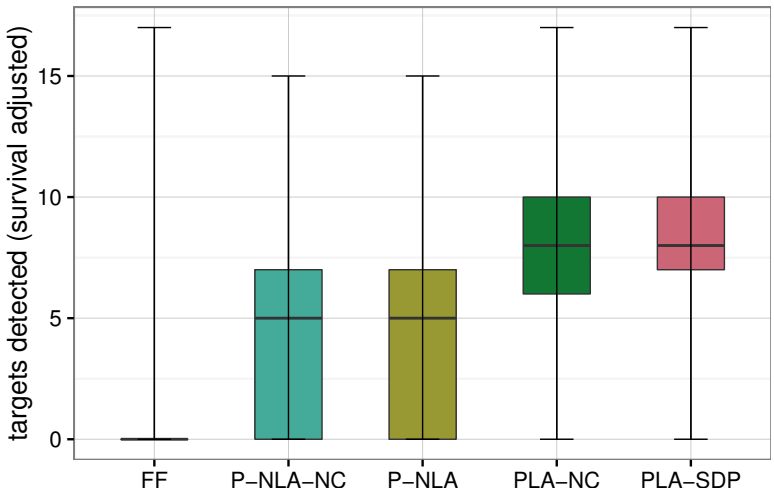


Figure 9.4: Comparison of target detection adjusted for mission survival in DART with partial approaches.

ciples of stochastic dynamic programming, but by carefully considering how environment and system transitions interacted, and how environment uncertainty affected those transitions, in order to deal with them in a way that avoids the state explosion that would result from constructing and solving the full joint MDP of the system and the environment (see Section 5.1.1). This is one of the design choices that, in part, gives PLA-SDP its speed advantage over PLA-PMC, but it does so at the expense of modifiability. As we discuss later, modifying PLA-PMC to handle other kinds of uncertainty would be relatively straightforward, whereas for PLA-SDP that would require considerable changes.

Another form of modifiability refers to changing the adaptation goal that the adaptation decision aims to achieve, which in its simplest form is the maximization of aggregate utility. PLA-SDP was designed to support different kinds of adaptation goals, as we described in Chapter 6. Even though they cover a wide range of cases, the kinds of adaptation goals it supports are limited to the different combinations of the three reward gain forms and the three[2] constraint satisfaction forms presented. In PLA-PMC, the adaptation goal is expressed as an extended PCTL property, which gives it much more flexibility (see Section 4.6). For example, the following multi-objective property for PRISM asks for the policy that maximizes the number of targets detected, such that the probability of not being destroyed is greater that 90% and the total amount of energy used is no more than 100.

multi (**R**{"targets"}**max**=?[**C**], **P**>=0.9 [**G** !destroyed], **R**{"energy"}<=100 [**C**])

An adaptation goal like this cannot be achieved with PLA-SDP unless it is further modified to support the third objective.

One aspect in which PLA-PMC is less flexible than PLA-SDP is that it cannot invoke external code. This means that everything needed to compute the utility function has to be programmed in the PRISM language. Even though utility functions are relatively simple, and thus easy to

[2]Not having a constraint satisfaction requirement is the third option.

program in PRISM, the inputs to the function may not be as simple to compute. For example, the estimation of the response time needed for RUBiS was done using queuing theory equations programmed in PRISM. However, it would not be possible to use third-party performance estimation tools such as OPERA [112] or LQNS [100] to compute the response time. PLA-SDP, on the other hand, can invoke third-party code as needed. Nevertheless, it would possible to work around this limitation of PLA-PMC by computing the utility for all the system and environment state pairs that the model could evaluate—which are countable—before invoking PRISM, and including the results as a table in the input to the model checker.

PLA-PMC is easier to modify than PLA-SDP. We have already shown how a new adaptation goal could be handled by changing the property that PLA-PMC aims to satisfy. Similar changes are doable with PLA-SDP, although they would require changes to the algorithm similar to the ones introduced in Section 6.2. Handling additional forms of uncertainty, such as stochastic tactic latencies would also be relatively easy to do with PLA-PMC by having the tactic module choose probabilistically a latency for the tactic when it starts executing. Such a change in PLA-SDP would require major changes, not only in the off-line computation of the reachability predicates, but also in the algorithm, which now considers uncertainty as belonging exclusively to the environment.

Considering the different features for the two approaches shown in Table 9.2, we can see that no approach dominates the other. Given the modifiability of PLA-PMC, and that it is the gold standard in terms of optimality, we envision that further advances in self-adaptation decision approaches could be led by the relaxation of assumptions and extensions of PLA-PMC, or by similar approaches inspired by it that leverage the generality and optimality of probabilistic model checking. The results from experimentation with these versions could be used to assess the benefit a particular modification. For example, it is not obvious that modeling stochastic tactic latencies would provide a substantial improvement over considering only its expected value. Extensions to PLA-PMC could be used to test the hypothesis that the extension improves its effectiveness, without expending the larger effort that a modification to PLA-SDP would require. If the extensions prove to be useful, they would then be followed by special purpose algorithms that exploit the structure of the problem to achieve speedups as we did with PLA-SDP. Having the probabilistic model checking version of the approach as the standard to compare against, the optimized approaches could strive to achieve the same effectiveness with faster decision times. In fact, the reason why we can confidently say that PLA-SDP is optimal is because in all our experiments it has produced exactly the same results as PLA-PMC.

9.3 Limitations

In this section, we discuss the limitations of our approach, and suggest ways in which they could be addressed in future work.

The approach only deals with exogenous uncertainty. In decision problems, uncertainty can be classified into two classes: *exogenous*, in which the stochastic process is not affected by the decisions; and *endogenous*, in which decisions affect the evolution of the stochastic process [60].

In the approach presented in this thesis, the stochastic process is the environment of the self-adaptive system, and we assume that, within the decision horizon, the adaptation actions do not affect the environment. That is, the approach deals only with exogenous uncertainty. This means that it is not suitable for systems that have tactics that affect the environment. For example, considering the DART scenario, if the drones were weaponized and one of the tactics were to fire at a threat, then that system would be subject to endogenous uncertainty, since the presence of a threat could be affected by the actions of the drones.

Despite this limitation, there is a broad class of self-adaptive systems that are not subject to endogenous uncertainty. For example, in an IT system such as RUBiS, the actions of the system do not directly affect the request arrival rate. One could argue that the actions of the system can end up affecting the environment in this case. For example, sustained poor performance would drive users away from the website, or showing the related products panel next to a product could induce more clicks and requests. Nevertheless, we argue that the approach is still suitable for such system for two reasons. First, the exogenous uncertainty assumption is only for the duration of the decision horizon. That means, that if the environment takes longer than the decision horizon to react to the actions of the system, then it is the same as if there were no endogenous uncertainty. Second, even if the environment reacts within the decision horizon, the approach decides periodically, with an interval between decisions that is much shorter than the horizon. Therefore, the system is able to make new adaptation decisions that take into account the change in the environment possibly induced by a previous adaptation. Obviously, if a previous decision started a tactic with latency that cannot be preempted and that affected the environment, then the system would suffer from not having considered the endogenous uncertainty. However, since the approach supports parallel tactics, it could compensate by using different tactics if available, such as changing the dimmer in RUBiS.

This limitation could be overcome in future work. The main challenge to accomplishing this is how to model the effect of tactics on the environment. For instance, in the case of RUBiS, one would have to be able to create a model of the environment capturing how the request rate is affected by the presence of the optional content in the responses; or in the case of DART, how the firing of a weapon at a threat modifies the probability of the presence of a threat.

Having such a model of the environment, PLA-PMC could be extended to deal with endogenous uncertainty by making the environment model synchronize with the tactics. In that way, when a tactic is executed in the model, the environment model would be able to reflect the impact of the tactic on its evolution. In the case of PLA-SDP, the changes are more fundamental, since the solution algorithm for stochastic dynamic programs with endogenous uncertainty cannot use the backwards induction approach, unless the history of the actions is included in the state. That notwithstanding, solution techniques for such decision problems exist [60], and their performance continues to be improved [67]. In addition, as long as transition feasibility remains independent of the state of the environment, as it is in PLA-SDP, it should be possible to use the same approach to compute the reachability predicates off-line, thus attaining a performance edge over a solution approach using probabilistic model checking.

The approach does not consider latency uncertainty. The latency of adaptation tactics is assumed to be deterministic (i.e., it is represented by a single value, and not a probability dis-

tribution). In addition, for PLA-SDP, the latency is assumed to be constant (i.e., that it does not change over time).

If the latency of a tactic is actually deterministic, then the approach computes the optimal solution over the decision horizon. If the latency is not deterministic, we still need to represent it with a single value, such as the expected value. What could happen is that if the realization of the latency is larger than the expected value, the protracted execution of the tactic may prevent other subsequent adaptations the decision could have planned from starting on time. If the realization could be shorter than the expected value, then a tactic may be disfavored over others, when in some realizations it would be better to have used it. Similar arguments can be made for other single-value representation of the latency, such as its worst-case behavior.

The assumption of constant latency in PLA-SDP is, however, not a fundamental one, and it would be relatively easy to overcome. The latency of the tactics could be monitored when they are executed, so that the latency value that is used for adaptation decisions can be updated at run time. In the models, the latency of the tactics is rounded up to the nearest multiple of the decision period. If the latency changes at run time such that this rounded value changes, all that needs to be done is to redo the generation of the delayed reachability predicate. However, it is not necessary to wait until a decision has to be made to generate the predicate for a different latency. For example, different predicates could be generated off-line for different latencies, or they could be generated at run time, before they are needed, using the lookahead technique proposed by Gerasimou et al. [56], by which spare resources are used to precompute solutions in the neighborhood of the current state (or tactic latency, in our case). Furthermore, a recently published technique and tool, *Titanium*, can speed up the analysis of evolving Alloy specifications by tightening the bounds of relations that are not changed [7]. Using that tool may result in a speedup of the time required to recompute the reachability predicate when only the tactic latency changes.

Relaxing the assumption of deterministic latency in PLA-SDP would require a fundamental change, since the introduction of probabilistic transitions in the system model of the MDP, would make impossible the use of Alloy to build the system model. One possibility to retain the advantage of using Alloy to do part of the MDP construction off-line, would be to remove the tracking of the tactic progress from the Alloy specifications, but still use Alloy to compute reachability predicates between states that do not include representation of tactic progress. The computation of the immediate reachability predicate would not be affected. The delayed reachability predicate, on the other hand, would be limited to representing whether or not, after some *unknown* delay, one configuration can be reached from another. The solution algorithm would then have to be changed to incorporate the probabilistic distribution of the latency of the tactics involved in a delayed transition, which will likely require abandoning the backward induction solution approach.

In PLA-PMC, on the other hand, it would be easier to deal with probabilistic tactic latencies. The selection of the tactic latency could be probabilistic, for example, by having different probabilistic transitions in a tactic module that assign different values to the latency of the tactic. Once the latency has been selected in this way, the tactic progress can be modeled in the say way as it is done in PLA-PMC.

This limitation would also be relatively easy to overcome in SB-PLA using the same approach to select the tactic latency probabilistically. The main difference is that instead of having separate

modules for each tactic, SB-PLA has a single module that models a complete strategy with its tactics. Nevertheless, the same idea could be used. Referring to the example shown in Listing 7.3, a probability distribution for the latency of the tactic TAddServer would be encoded replacing the update portion of the command in line 14 with one including multiple probabilistic updates, each with a different probability and a different tactic latency. An example of a command encoding a probability distribution for the tactic latency is

```
[] sys_go & node=2 & !tacticRunning ->
        0.25 : (tacticRunning'=true) & (exec'=TAddServer_LATENCY_LOW)+
        0.50 : (tacticRunning'=true) & (exec'=TAddServer_LATENCY_MEDIUM)+
        0.25 : (tacticRunning'=true) & (exec'=TAddServer_LATENCY_HIGH);
```

The approach requires being able to estimate the system measures of performance that are used to compute utility. To be proactive, the approach needs to be able to estimate the utility that the system will accrue over the decision horizon under many different system configurations that could be reached through adaptation, and under many different realizations of the environment. This means that it is necessary to be able to estimate the utility that a given system configuration would yield in a given environment. In turn, utility depends on one or more measures of performance of the system, such as the probability of detecting a target, or of being hit by a threat in DART; or the response time in RUBiS. In many cases, there are theories or analyses that can be used to estimate these measures. For instance, response time can be estimated using queuing theory, or layered queuing network analysis. In other situations, models specific to the system can be used, as it was done in the DART example to compute the probability of detecting a target.

Using theories, analyses, or models to make adaptation decisions or solve planning problems is not uncommon, since it is necessary to have a way to estimate the consequence of future actions [6, 45, 51, 73, 77, 103, 117, 146, 147]. However, there may be measures of performance for which accurate theories or models do not exist. For those cases, it could be possible to use machine learning to learn to estimate the impact of adaptation tactics. Using machine learning in self-adaptive systems is not a novel idea (see [41] for example), however, we think it is possible to avoid the initial error-prone learning phase by having the self-adaptive system use an initial rough approximation to do the estimation, and improve over time through experience using machine learning. If the initial approximation is good enough, even if largely suboptimal, the system can start operating, collecting observations of the measure that can be used to train an estimator using machine learning. Different approaches to combine the baseline estimation with the machine-learned one can be used to achieve better estimations than would be possible with either by itself [34].

The approach requires that the adaptation goal be expressible in one of the twelve forms of utility supported. This limitation comes from the *Principle of Optimality* in dynamic programming, which states that "an optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision" [10]. This implies that the optimal policy starting from any state in any decision interval in the decision horizon is independent of how that state is

reached, which is the key to solving optimization problems through backwards induction as we do in PLA-SDP. An example of a utility function that does not satisfy the Principle of Optimality in our approach is one in which the utility of particular system configuration in a given state of the environment depends on the configuration the system had in a previous decision interval.

The satisfaction of the Principle of Optimality is not a property of the actual problem being solved, but a property of how the problem is modeled. For example, it would be possible to encode the history of previous system states in the system state, making the decision in a given state independent of how that state was reached. That is because there would be only one way in which a state could be reached, which would be encoded in the state itself. Doing this in PLA-SDP would require not only changing the definition of a system state, but also making changes so that states can be reached only if that is consistent with the history they encode. That could be done in two ways. Either the computation of the reachability predicates is modified so that reachability is consistent with history; or the algorithm is changed to add history consistency to the selection of feasible next states. Regardless of what is changed, the main drawback is that the size of the state space would increase by a factor of $|C|^{H-1}$. For PLA-PMC, it would also be necessary to encode the history in the system state, thus causing the same increase of the size of the state space as in PLA-SDP. However, PLA-PMC would not require other changes to compute the optimal policy.

The approach does not directly use feedback of actual performance for adaptation decisions. An important reason to use feedback in control theory is to reduce the impact of disturbances and modeling errors on the ability to control a system [15]. In a feedback control loop, the process output (i.e., the metric the system is trying to control) is measured so that its deviation with respect to the desired set point can be computed. Based on this deviation or error, the controller decides what actuation, if any, is needed to bring the output close to the set point.

The approach presented in this dissertation uses direct feedback of the actual system state. For example, if a tactic fails to execute, as long as the model is updated by monitoring the running system, the adaptation manager will know that the system is not in the state it was supposed to reach after the execution of the tactic, and if it still deems the tactic is needed, it will decide to execute it again. This is accomplished by the periodic nature of the adaptation decision, which uses the current *observed* state of the system to make a new adaptation decision.

However, unlike classical feedback loops, the approach does not compare the utility of the system—what we are trying to control—with a set point. To illustrate the consequence of this limitation, let us suppose that in the RUBiS adaptation manager the underlying queuing theory model estimates that by adding a server the response time will drop below the threshold. The adaptation manager then decides to add a server, but after it does so, the response time is still above the threshold. The approach does not directly use that feedback to take a corrective action because it does not use the observed response time as an input to the next decision. Furthermore, unless the request rate changes, it will not take further actions because the theory will estimate—as it did before—that the response time will be below the threshold with the current configuration, even if it is actually above.

This limitation is overcome by using feedback within the underlying predictor so that its predictions are corrected by the observations gathered from the running system. For example,

this was already done in the RUBiS adaptation manager with the use of the Kalman filter to estimate the service rate of the servers in the system. This estimated service rate was, in turn, used as an input to the queuing theory model to predict the response time. In that way, the approach does not directly use the feedback, but benefits from it indirectly, through the use of feedback to improve the predictions it relies on to decide.

Other ways in which feedback could be used is by exploiting the previous history of the execution of the system. For example, Rainbow keeps track of the failure rate of each strategy, and that information can be used to avoid strategies that have had a high failure rate [28]. Another way of using feedback would be with the approach that improves the utility function through machine learning as was previously described.

The approach requires formal models of the adaptive system. Formal models of the system and the adaptation tactics have to be specified in Alloy for PLA-SDP, and PRISM for PLA-PMC. Since the vast majority of software engineering practitioners are not trained in the use of formal methods, this could be a barrier for the adoption of the approach. However, it would be possible to automate the generation of these models, and even though this generation is not in the scope of this thesis, special effort was put in devising models that follow very regular patterns (see sections 4.5, 5.2, and 7.2.1), facilitating the future development of this automatic model generation.

We envision that these formal models can be generated from a suitable domain-specific language (DSL), such as an extended version of Stitch. The main components of the formal models are the system model and the tactic models. The system model consists of one or more properties representing a highly abstracted representation of the system, containing only the information needed to make adaptation decisions. For example, the underlying architectural model of RUBiS may have the topology of the 3-tier architecture, but only the number of active servers is needed. Given a description of the tactics and the utility function, it would be possible to automatically determine which properties of the system would be needed in the model. These properties would include those used in tactic applicability conditions; and those that are both affected by tactics, and needed to compute the utility function. For tactics, we need the DSL to capture the applicability condition as a predicate over the properties of the system, the tactic latency, and the effect of tactic completion on the properties of the system. Given this information, it would be possible to generate the models following the patterns presented in chapters 4, 5, and 7.

The approach may not scale to large system state spaces. As noted in the scalability analysis presented Section 7.4, the complexity of PLA-SDP is linear in the number of system states, which in turn is exponential in the number of adaptation tactics. Even though the number of tactics is unlikely to be more than 20 (see [147] for example), when compounded with the properties that define the system configuration, it can lead to large state spaces. Depending on the decision period, the decision time of PLA-SDP may not be sufficiently fast to avoid overrunning the period. In such cases, it would be necessary to forgo optimality. We have already presented SB-PLA in Chapter 7 to address this limitation within this thesis. Since SB-PLA relies on a predefined repertoire of adaptation strategies, it has less flexibility when deciding how to adapt, which can result in less effectiveness than the main PLA approaches. On the other hand, having

fewer choices to select from when making adaptation decisions, SB-PLA can scale to systems with large state spaces. In addition, there are other possible alternatives that could be explored in future work to address scalability in large systems.

One approach would be to have hierarchical adaptation decisions that operate at different levels of discretization. For example, suppose that the number of servers in RUBiS could be any in the range $[1, 100]$. A first decision could be made in a state space discretized to tens of servers, and find an approximate solution at that level of granularity. Subsequently, another adaptation decision would be made with the range of servers reduced to the block of ten servers selected by the previous decision.

Another approach consists of computing approximate solutions. There are different strategies that can be used to deal with the scalability limits of dynamic programming, such as solving forward in time—instead of backwards—using value function approximations, thus avoiding having to evaluate all the states in the state space [119]. A different approximation can be obtained by considering only adaptation decisions at the beginning of the decision horizon, and assuming that no further adaptations are made when computing the value of the different solutions [3].

Another promising approximate solution technique is the cross-entropy (CE) method, a generic approach to solve combinatorial optimization problems [32]. The CE method consists in (i) randomly generating a set of candidate solutions (adaptation paths in our case); (ii) selecting an "elite set" with a fraction of the solutions that score highest according to the optimization criteria; (iii) using the sample distributions of the elite set to update the parameters of the distributions from which candidate solutions are sampled; and (iv) repeating the whole procedure until some stopping criterion is met (e.g., convergence, or time limit).

In cases in which the maximum rate of change of the environment can be known, an approach that can speed up the decision is to limit the possible adaptation actions to those that would be necessary to deal with the maximum possible change of the environment within the decision horizon. For example, Naskos et al. bound the number of virtual machines that could be added or removed in an adaptation decision based on the maximum change the system load could experience [111]. This technique is also suitable for cases in which the system state change is limited within the decision horizon. For example, in the DART example, given the maximum rate of altitude change, it would be possible to limit the state space considered in a decision to altitude levels that would be reachable within the decision horizon, subject to the physical characteristics of the system.

It is important to note that even if some of these techniques are used to speed up the adaptation decision, most of the elements of the approach proposed in this thesis remain the same. For example, if CE were used, only the algorithm to compute the solution to the MDP would be changed. All the off-line computation would remain the same, and the scoring of the solutions would be largely based on the formulation in (5.4)-(5.6), except that instead of finding the configurations that maximize the value, the value of the configuration determined by the solution would be computed.

Although in this discussion about scalability we have focused primarily on the run-time adaptation decision time, the state spaces size also has an impact on the time it takes to compute the reachability predicates using the Alloy analyzer. Even though long analysis times are tolerable, since they affect only the off-line computation, it is possible for the state space to be too large for Alloy to analyze (i.e., it would run out of memory). In our experiments, however, this has

not been an issue, even when we increased the size of the state space by adding more tactics and increasing the number of possible values state variables can take. The off-line computation for the experiment with a large RUBiS cluster described in Section 8.2.3 took 3 minutes. In a configuration of DART with eight tactics and the altitude discretized into 100 levels, the off-line analysis took 15 minutes. If for some system the off-line computation became an issue, it would be possible to use the same approaches that were previously discussed to reduce the state space size.

The approach relies on parameters that may need to be tuned. There are two main parameters that the approach requires to be specified: the look-ahead horizon H and the decision interval τ. The duration of the horizon (i.e., $H\tau$) should be long enough to cover the largest latency of all the tactics, otherwise those long latency tactics would never be selected. On the other hand, given that the further into the future, the more uncertainty environment predictions have, it does not make sense to have a horizon that is too long, forcing the decision to consider very uncertain predictions. In addition, the larger H, the slower the adaptation decision will be.

The decision interval τ also affects the effectiveness of the approach. The smaller τ is, the more effective the approach is because it has more frequent opportunities to deal with changes in the environment. However, if τ is too small, scalability suffers because H would have to be large in order to cover the latency of the adaptation tactics. If τ is too large, then the benefit of being able to use very low latency tactics to complement large latency tactics may be diluted. For example, if the decision interval were set to the latency of the tactic to add a server in RUBiS, then, it would not be possible to change the dimmer half-way through the addition of a server to adjust to unpredicted changes in the environment.

In some domains, there may be system and environment characteristics to help guide the selection of these parameters. In DART, for example, τ is the length of a route segment divided by the speed of the drones. In that way, an adaptation decision can be made at each route segment boundary. The horizon H can be determined based on the range of the forward looking sensors. In other domains, it could be possible to determine the period τ considering the feasible rate of change of the environment in addition to the latency of the tactics. For example, in a wireless sensor network for forest fire detection, like the one described by Paez Anaya et al. [113], the relevant property of the environment is the forest temperature. In that case, the decision interval τ could be computed based on the maximum rate of change that the forest temperature can experience. In addition, how much error is tolerated also plays an important part in the determination of τ. For example, the system designer should consider whether it is acceptable for the temperature to increase by 5 degrees before an adaptation decision is made, or if the system should make more frequent decisions to adapt to smaller changes in temperature. Even in information systems, the rate of change of the environment could be taken into account to determine τ. Naskos et al. use the maximum change the system load could experience in a decision interval to limit the adaptive behavior to be considered (i.e., number of virtual machines to be added or removed) [111]. Similarly, the maximum rate of change for the load could be considered to determine the decision interval τ given the maximum load change that available tactics can deal with.

In this thesis we have not made a special effort to tune these parameters—we just used heuristics to set them. However, it is likely that to obtain the most benefit of the approach, these parameters will have to be tuned for particular systems. This is not unlike other well known control approaches like model predictive control (MPC) [19]. Implementing MPC requires tuning its parameters, which are similar to those of this approach. Although many guidelines and heuristics have been proposed for tuning the parameters of MPC [55], there are no equations that can optimally determine them. Thus, some approaches have been proposed to do MPC parameter tuning using sequential parameter optimization [31], and particle swarm optimization [133]. Since these approaches treat the controller as a black box, we believe it would be possible to adapt them to tune the parameters of the approaches presented in this thesis.

9.4 Future Work

In the previous section we have already suggested ways in which the limitations of the approach could be addressed, and all those are candidates for future work. We now highlight some of those, and describe other areas that, although related, would require substantial research. These areas of future work are presented roughly in increasing order of perceived difficulty.

PLA model generation. Both PLA-SDP and PLA-PMC require formal models of the adaptive system and the tactics specified in Alloy and PRISM respectively. The models for the two approaches follow patterns, as shown in chapters 4 and 5, that make their automatic generation possible. In the previous section, we already provided a summary of what elements would be needed to generate the models. How challenging the generation would be depends on the desired level of automation. The most challenging aspect would be determining what system properties must be included in the model given the utility function and the specification of the tactics. As was explained in previous chapters, there is a method for a human to do this, but for a generator to do it would require that it be possible to express the utility function in the DSL used as input. From the utility function, the generator has to determine what inputs are necessary. If the inputs are properties of the system, the mapping is direct. However, some inputs may be emergent properties of the system, such as the response time. In that case, a specification of the function that estimates this emergent properties from the properties of the system and the environment would be needed (or its interface at minimum). In the end, the generator has to be able to define all the properties of the system state that are needed to compute the utility function $U(c, e)$.

Combining reactive with PLA self-adaptation. In this thesis we have shown how proactive adaptation can improve over reactive adaptation. However, there are situations in which a combination of both would be desirable, especially to address one of the limitations of proactive adaptations. Proactive adaptation relies on being able to estimate future measures of performance such as response time. However, suppose that in some situation it fails to do so correctly. For example, the estimation model used by the proactive adaptation decision may have erroneously estimated a response time below the required threshold, but then, the response time does not meet the requirement. As we pointed out in the previous section, because PLA does not use feedback directly, it will not do anything to address this problem if its estimation model

continues to predict wrongly a lower response time. Reactive adaptation, which does not rely on estimates of future performance, could directly use the feedback of not meeting the response time requirement to take action.

A combination of proactive and reactive adaptation was already proposed by Gmach et al., applying it to resource pool management [58]. In their approach, undesired interactions between the two paradigms are avoided by having the proactive part make longer term adaptation decisions, and reactive adaptation making corrections in between. The proactive part relies on historical workload patterns that tend to repeat weekly or seasonally, but given that it runs with a much longer decision interval than that of the reactive part (4 hours vs. 5 minutes), it would not be very useful to deal with faster changing workloads. We believe it would be more effective to have the proactive adaptation operate with a shorter decision interval, as we do in this thesis to gain its full benefit. However, naively combining the reactive with proactive adaptation would not work well. For example, assuming there are no errors in its estimates, proactive adaptation could have decided not to add a server in spite of an upcoming spike in traffic because it may have known it was going to be short lived, or because it knew there was a tactic already executing that would bring the response time down, or because it knew that starting a tactic would have blocked a more important adaptation tactic needed in the near future. If reactive adaptation then came and added that server anyway, it would undo part of the benefit of proactivity, which is making current decisions taking into account how they will affect future decisions. For this combination to work correctly, there would have to be some way for the proactive side to inform the reactive part what its intention was. For example, it may inform the reactive part in some way that it expected the response time to go above the threshold at time t, so that the reactive part does not attempt to deal with that. In summary, the idea would be to use reactive adaptation to address the limitations of proactive adaptation, but making sure that it does not override proactive decisions that were right.

Improving adaptation decisions by learning from experience. Being proactive requires being able to estimate the utility that a given system configuration would attain in a given environment, so that the best adaptation action can be taken to achieve the adaptation goal. The utility function depends on one or more measures of performance of the system, such as the probability of detecting a target, or the response time. In general, these emergent properties can only be approximately estimated using models such as a queuing theory model, or a model for a UAV on-board camera [6]. Using these approximations would of course result in an approximation to the actual utility to be attained, and consequently, the adaptation decision may be affected by errors in this approximation.

Approaches that use machine learning to estimate the impact of adaptation tactics, on the other hand, do not require these models to estimate future utility, and can eventually learn the utility function. However, as pointed out by Esfahani et al., they require an initial learning phase, during which the system may not perform well, as they need to do exploration to learn [41].

Using predictors such as those used in PLA (referred to as baseline predictors) in combination with machine-learned predictors would allow the system to operate initially relatively well (as much as the approximation of the baseline predictors allows), and improve over time through learning from the experience it gets while it is in operation. The approaches proposed by Didona

et al. allow combining baseline predictors with machine-learned predictors obtaining better prediction accuracy than them individually [34]. For example, the K-Nearest-Neighbors approach computes the average prediction errors for both the baseline predictor and the machine-learned predictor in the neighborhood of the state for which a new prediction is needed. Then, the one with the least error in the vicinity of that state is used. This will allow determining automatically which predictor to use for different areas of the state space. In areas in which the system has had less experience, the machine-learned predictor is likely to have higher prediction error than the baseline, and thus, the latter will be used. On the other hand, the decision can exploit what it has learned with machine learning in areas that have been well explored at run time.

Dealing with other aspects of timing. In this thesis, we have dealt with the timing of adaptation considering two aspects of the problem: when the adaptation is carried out, and how long it takes. There are other aspects of timing in self-adaptation that could be considered. Pandey et al. propose combining different planning approaches for self-adaptation, taking into account the trade-off between how quickly each can make an adaptation decision and the quality of the solution [114]. For example, one planner may provide quick approximate decisions, whereas other approaches may take more time to decide, but provide better decisions. In real-time systems, the adaptation planning itself may need to be scheduled as a task that must be completed within a deadline. Musliner proposes an approach to do controller synthesis for self-adaptive real-time systems, in which the reconfiguration of the system has to be done within a deadline [110]. Adaptation decision approaches based on any-time algorithms, such as the cross-entropy method previously discussed, could be useful for systems with such constraints. For real-time control systems, the Simplex architecture provides a way to automatically switch to a baseline controller if the more advanced, but possibly less tested, controller takes the system to an undesirable state [130]. In this case, the adaptation decision is about selecting the controller whose output will be used to control the system. When controllers for cyber-physical systems are developed, models are used to do off-line verification and provide guarantees about the system behavior. However, those guarantees are only valid as long as the behavior observed at run time matches these models. ModelPlex is a method that can monitor the system to detect deviations from the models, and if the behavior does not fit the model, it can initiate fail-safe actions [107].

Additionally, when self-adaptation is used in real-time systems, it must be ensured that the system will continue to meet its timing requirements after an adaptation is carried out. Steiner et al., for example, propose an approach that performs a schedulability analysis of the system configuration that would result from an adaptation [131].

Integrating these timing aspects with those considered by PLA would be desirable, or even required, for some kinds of systems.

Combining PLA with control theory. For years, control theory has been a source of inspiration for the self-adaptive systems research community, not only because it provides examples of different forms of control loops, but also because of its mathematical underpinnings that allow computing properties such as controllability, and stability [15, 26]. Two of the biggest challenges for the use of control theory for software self-adaptation has been the need to have mathematical models of the software system's dynamics, and the lack of software engineering methods that

treat controllability as a first class concern [44]. However, several contributions have been made in the last few years addressing them [3, 43, 83]. Control theory is naturally better at handling continuous actuators, but PLA requires discretizing them, as we did with the dimmer in RUBiS. Conversely, PLA inherently deals with a discrete state space (and corresponding actuators), whereas control theory has to resort to some conversion from the continuous control signal it computes to discrete actuation. In addition, PLA can reason about conflicts between tactics, tactic applicability given the system state, and how the use of a tactic can constrain the reachable states in the near future, whereas control theory cannot easily handle these.

Given these complementary characteristics, finding a way to combine both approaches may increase the effectiveness and/or the applicability of each approach. For example, handling dimmer settings continuously would allow the approach to be more effective than its discretized counterpart without incurring the state explosion that even approximating it would imply with PLA. At the same time, state-dependent, discrete tactics with latency would be better handled by PLA, rather than computing a continuous signal that represents the change in the number of servers, which first must be discretized, and second, may not be aware of the infeasibility of adding servers at a given time.

There are several criteria that could be used to allocate actuators to the two kinds of controls. For example, it could be by discrete vs. continuous, or by immediate vs. with latency. Or perhaps the controllers would be hierarchical with the inner control loop based on control theory and the outer loop using PLA. It is not immediately obvious how this combination would be best realized, and that poses an interesting research challenge.

Handling endogenous uncertainty. As we noted in the previous section, the PLA approach only deals with exogenous uncertainty; that is, it assumes that adaptation actions do not affect the environment. Dealing with endogenous uncertainty may be necessary for some domains. As mentioned before, weaponized drones in DART with a tactic to fire at a threat would require that in order to account for the fact that such a tactic could alter the presence of threats in the environment ahead. In general, given their actual interaction with the environment, cyber-physical systems are more likely to require dealing with endogenous uncertainty. For example, a robot could push an object out of the way, thus changing its environment; and a smart home could turn off outdoor lights to save power, but at the same time make itself more attractive to burglars. In addition, any self-adaptive system for which its environment is adversarial would benefit from, or even require, understanding how its actions affect the behavior of the environment. Cybersecurity is one clear example, in which self-protecting actions taken by the system can influence the behavior of the attacker. For instance, the disconnection of a compromised computer will be detected by attackers prompting them to change their behavior; however, if the network connections or services the attacker is accessing are replaced with emulated ones, that change is not detected by the attackers, allowing a defender to gather more information about them [61].

Handling endogenous uncertainty would not be too difficult with probabilistic model checking, as noted earlier. The challenge in this case would be modeling the impact of tactics on the environment, and how to model the environment in a way that includes not only the probabilistic transitions the represent the uncertainty about the state of the environment—what we currently model—but also the probabilistic transitions that result from the tactics the system can

use. For PLA-SDP, there is an additional challenge, since dealing with endogenous uncertainty means either foregoing the backwards induction approach that supported a fast solution of the decision problem, or storing the history of actions in the state. In any case, state space explosion will probably be an issue, something that would require looking at solution approaches that are parallelizable.

Reasoning about tactics that reduce uncertainty. This challenge is related to the previous one, however in this case the system's action do not affect the environment behavior, but change what the system knows about the environment. For example, in a self-protecting system, a tactic may consist in observing an attacker to identify its tactics, techniques and procedures. Such a tactic would have a large latency, but will give the system more information about its environment, rather than affect the environment. Another example is to have tactics that can change the number of samples taken by the forward-looking sensors in DART. Again, these tactics would only affect what the system knows about the environment (e.g., the more samples it takes, the less variance the prediction of the environment has). Having better information about the environment would result in better predictions, possibly at the expense of waiting for those information gathering tactics to execute. The fundamental difference between these kinds of tactics and the tactics used in this thesis is that for the former it is not possible to determine how using the tactic will concretely affect the model of the environment the system has. This means that it is not possible for the decision procedure to factor in precisely how using a tactic of this kind affects the future.

Dealing with this kind of tactic that updates the belief that the system has of the environment will likely require resorting to partially observable Markov decision processes (POMDP) [79]. However, the solution approaches for MDPs do not easily extend to POMDPs. Furthermore, the use of probabilistic model checking is not an option, because at the time of this writing PRISM does not support POMDPs. Perhaps solution approaches developed for planning could be used, but the challenge will be making them fast enough to make adaptation decisions in a reasonable time.

9.5 Summary

In this chapter we have shown that the combination of the three pillars of PLA (i.e., proactivity, latency-awareness, and concurrent tactics) performs consistently better than any of them used individually. We have also discussed why both PLA-PMC and PLA-SDP are relevant. PLA-PMC is more modifiable and better suited for trying new developments in self-adaptation decision-making, whereas PLA-SDP performs much faster and does not limit the encoding of the predictors needed for the approach to a single language.

In addition, we discussed the limitations of the approach and how they could be addressed, either with workarounds or with future research. We also discussed several areas of interesting future work in a spectrum of near term improvements to facilitate adoption, to more challenging topics that would require substantial research.

Chapter 10

Conclusion

In previous chapters we analyzed how the different elements of PLA contribute to the improvement in effectiveness that it attains; reviewed the importance of having both PLA-PMC and PLA-SDP; discussed limitations of the approach; and proposed areas of future work. This chapter concludes the thesis by listing its contributions and providing a brief summary.

10.1 Contributions

This thesis advances software engineering through improvements in the effectiveness of self-adaptive systems by considering timing in adaptation—when to adapt relative to the predicted needs, and how long that takes. The main contributions of this thesis are:

- a conceptual framework for proactive latency-aware adaptation that describes how the elements of PLA are combined, and defines the PLA adaptation decision problem independent of the solution approach. (Chapter 3)
- a solution approach based on probabilistic model checking, which, through exhaustive analysis, can find the optimal solution to the PLA adaptation decision problem. Given its optimality, it serves as a gold standard with which other approaches can be compared. In addition, it is easily modifiable, and thus suitable to explore future extensions to PLA. (Chapter 4)
- a solution approach based on the principles of stochastic dynamic programming, which exploits the problem structure to reduce the adaptation decision time by an order of magnitude while computing the same optimal solution as PLA-PMC. (Chapter 5)
- a strategy-based solution approach that uses the principles of PLA to improve strategy-based adaptation. This approach provides a PLA solution that can be used in systems in which the adaptive behavior needs to be limited to predefined and tested adaptation strategies. Additionally, this approach is more scalable than the other two. (Chapter 7)
- support for a variety of adaptation goals formed by combining how reward is gained and an optional requirement on the satisfaction of a probabilistic constraint. This makes the PLA-SDP solution approach applicable to systems with different kinds of adaptation goals. (Chapter 6)

Additionally, there are other secondary contributions:

- demonstration of the approach applied to two different systems in two very different domains, and implemented with different self-adaptation frameworks
- an implementation of these approaches as a library suitable for different kinds of systems
- implementation of PLA adaptation managers, including SB-PLA, for Rainbow
- SWIM, a simulation of a web system suitable for experimentation in self-adaptive systems that can be used for comparing self-adaptation approaches, and as an easy-to-deploy target system for Rainbow

10.2 Summary

In Chapter 3 we explained why the timing aspects in self-adaptation are important, and introduced the concept of proactive latency-aware adaptation (PLA) as a way to explicitly consider them. PLA has three main pillars: (i) *latency awareness* to take into account how long adaptation tactics take to execute, to considered not only their delayed effect, but also how their execution affects the feasibility of subsequent adaptations; (ii) *proactivity* to adapt proactively considering the anticipated needs of the system based on predictions of the near-future state of the environment; and (ii) *concurrent tactic execution*, which leverages non-conflicting tactics to complement tactics that have long latency with faster tactics, and to reduce the amount of time required to complete an adaptation by executing tactics concurrently. The PLA adaptation decision problem was formulated independently of specific solution approaches, as the problem of deciding what adaptation tactic(s) to start at the time the decision is being made in order to maximize the utility accrued over the decision horizon. Additionally, we proposed using Markov decision processes as the formalism to model the adaptation decision problem, which we then solve using two novel approaches.

The first of these approaches, PLA-PMC was presented in Chapter 4. PLA-PMC uses probabilistic model checking to make adaptation decisions. The key idea is to create a model of the system, its environment, and the adaptation tactics, but leave the choice to start adaptation tactics or not underspecified through nondeterminism. This model is specified in the PRISM language and is a high-level representation of the underlying MDP. The model is then analyzed by the PRISM model checker to synthesize a policy resolving the nondeterminism so that the adaptation goal, specified as a PRCTL property, is satisfied.

In PLA-PMC, the model checker must process the model to build the MDP every time an adaptation decision has to be made. In Chapter 5 we introduced PLA-SDP, an approach that virtually eliminates that overhead by building most of the MDP off-line. PLA-SDP uses Alloy off-line to analyze a model of the system and the adaptation tactics to compute the reachability predicates that encode the deterministic transitions in the MDP. At run time, a novel custom algorithm based on the principles of stochastic dynamic programming is used to solve the adaptation decision problem, weaving in the probabilistic environment transitions as it computes the solution. PLA-SDP achieves a drastic speed-up in this way, while computing the same optimal solution as PLA-PMC.

In Chapter 6 we presented a way to support different notions of utility in addition to maximization of aggregate reward. With this approach, adaptation goals can be defined by composing a form of reward gain, with a form of constraint satisfaction. The former specifies how reward is gained relative to the constraint satisfaction (e.g., reward is gained as long as the constraint has always been satisfied). The latter optionally imposes a requirement on the probability of satisfying the constraint. These more complex notions of utility are necessary to deal with adaptation goals in some self-adaptive systems, as demonstrated in the DART system, in which the goal is to maximize the number of targets detected while keeping the probability of surviving the mission above a bound—something that cannot be encoded in a simple additive utility function since the two measures are not comparable.

Both PLA-PMC and PLA-SDP are tactic-based solution approaches, in that they have the flexibility to combine tactics in arbitrary ways. For some systems, the designer may prefer to limit the adaptations to tried and tested adaptation strategies, which combine tactics in predefined ways. In Chapter 7 we presented SB-PLA, a solution approach that uses the principles of PLA to improve the effectiveness of strategy-based adaptation. SB-PLA is also much more scalable than the other approaches, since the solution space is reduced when the adaptation decision is done in terms of strategies. Though not as effective as the other two approaches, SB-PLA still provides an improvement over non-PLA strategy-based adaptation, thus providing a reasonable compromise for systems in which the full approaches would take too long to make adaptation decisions.

The approaches were demonstrated with two systems that were used to validate thesis' claims. RUBiS is a web system, and DART is a cyber-physical system. These systems are different in several important aspects including their adaptation goal, the kinds of tactics they have, the kinds of environment predictions they use, and how they are implemented. Given that the approach presented in this thesis was used for these two different systems, we believe that PLA will improve the self-adaptation effectiveness of other systems, and that it will inspire other adaptation approaches by highlighting the importance of considering adaptation timing in self-adaptation.

Appendix A

PLA-PMC PRISM Model for RUBiS

The following listing shows the template for the PLA-PMC PRISM model for RUBiS. This template is completed at run time before each adaptation decision by injecting the initialization block in the tag //# init , and the environment model in the tag //#environment.

```
mdp
//  init  block must include values for  the  following  constants
//  const  int  HORIZON
//  const double PERIOD
//  const  int  DIMMER_LEVELS
//  const double DIMMER_MARGIN
//  const  int  MAX_SERVERS
//  const double RT_THRESHOLD
//  const  int   ini_servers
//  const  int  ini_dimmer
//  const double AddServer_LATENCY
//  const  int  ini_AddServer_state
//  const double serviceTimeMean
//  const double serviceTimeVariance
//  const double lowServiceTimeMean
//  const double lowServiceTimeVariance
//  const  int  threads
// /
//# init
label " initState " = servers = ini_servers & AddServer_state = ini_AddServer_state
  & dimmer = ini_dimmer;

label " final " = time = HORIZON & readyToTick;
formula sys_go = readyToTick;

module clk
  time :  [0.. HORIZON] init 0;
  readyToTick : bool init  true;
  [ tick ]  readyToTick & time < HORIZON -> 1 : (time' = time + 1) & (readyToTick'=false);
  [tack]  !readyToTick -> 1 : (readyToTick'=true);
endmodule

module env
//#environment

//  tactic  concurrency rules
formula AddServer_compatible = !RemoveServer_used;
formula RemoveServer_compatible = !AddServer_used;
formula IncDimmer_compatible = !DecDimmer_used;
formula DecDimmer_compatible = !IncDimmer_used;

//  tactic
```

```
formula AddServer_used = AddServer_state != 0;
const int AddServer_LATENCY_PERIODS = ceil(AddServer_LATENCY / PERIOD);

// applicability conditions
formula AddServer_applicable = servers < MAX_SERVERS & AddServer_compatible;

module AddServer
  AddServer_state : [0.. AddServer_LATENCY_PERIODS] init ini_AddServer_state;
  AddServer_go : bool init true;

  // tactic applicable, start it
  [AddServer_start] sys_go & AddServer_go // can go
    & AddServer_state = 0 // tactic has not been started
    & AddServer_applicable
    -> (AddServer_state' = 1) & (AddServer_go' = false);

  // tactic applicable, but don't use it
  [] sys_go & AddServer_go // can go
    & AddServer_state = 0 // tactic has not been started
    & AddServer_applicable
    -> (AddServer_go' = false);

  // pass if the tactic is not applicable
  [] sys_go & AddServer_go
    & AddServer_state = 0 // tactic has not been started
    & !AddServer_applicable
    -> 1 : (AddServer_go' = false);

  // progress of the tactic
  [] sys_go & AddServer_go
    & AddServer_state > 0 & AddServer_state < AddServer_LATENCY_PERIODS
    -> 1 : (AddServer_state' = AddServer_state + 1) & (AddServer_go' = false);

  // completion of the tactic
  [AddServer_complete] sys_go & AddServer_go
    & AddServer_state = AddServer_LATENCY_PERIODS // completed
    -> 1 : (AddServer_state' = 0) & (AddServer_go' = true); // so that it can start again at this time if needed

  [tick] !AddServer_go -> 1 : (AddServer_go' = true);
endmodule


// tactic
module RemoveServer
  RemoveServer_go : bool init true;
  RemoveServer_used : bool init false;

  [RemoveServer_start] sys_go & RemoveServer_go
    & servers > 1 & RemoveServer_compatible // applicability conditions
    -> (RemoveServer_go' = false) & (RemoveServer_used' = true);

  // tactic applicable but not used
  [] sys_go & RemoveServer_go // can go
    & servers > 1 & RemoveServer_compatible // applicability conditions
    -> (RemoveServer_go' = false);

  // pass if the tactic is not applicable
  [] sys_go & RemoveServer_go
    & !(servers > 1 & RemoveServer_compatible) // applicability conditions negated
    -> 1 : (RemoveServer_go' = false);

  [tick] !RemoveServer_go -> 1 : (RemoveServer_go' = true) & (RemoveServer_used' = false);
endmodule


// tactic
```

```
module IncDimmer
  IncDimmer_go : bool init true;
  IncDimmer_used : bool init false;

  [IncDimmer_start] sys_go & IncDimmer_go
    & dimmer < DIMMER_LEVELS & IncDimmer_compatible // applicability conditions
    -> (IncDimmer_go' = false) & (IncDimmer_used' = true);

  // tactic applicable but not used
  [] sys_go & IncDimmer_go // can go
    & dimmer < DIMMER_LEVELS & IncDimmer_compatible // applicability conditions
    -> (IncDimmer_go' = false);

  // pass if the tactic is not applicable
  [] sys_go & IncDimmer_go
    & !(dimmer < DIMMER_LEVELS & IncDimmer_compatible) // applicability conditions negated
    -> 1 : (IncDimmer_go' = false);

  [tick] !IncDimmer_go -> 1 : (IncDimmer_go' = true) & (IncDimmer_used' = false);
endmodule

// tactic
module DecDimmer
  DecDimmer_go : bool init true;
  DecDimmer_used : bool init false;

  [DecDimmer_start] sys_go & DecDimmer_go
    & dimmer > 1 & DecDimmer_compatible // applicability conditions
    -> (DecDimmer_go' = false) & (DecDimmer_used' = true);

  // tactic applicable but not used
  [] sys_go & DecDimmer_go // can go
    & dimmer > 1 & DecDimmer_compatible // applicability conditions
    -> (DecDimmer_go' = false);

  // pass if the tactic is not applicable
  [] sys_go & DecDimmer_go
    & !(dimmer > 1 & DecDimmer_compatible) // applicability conditions negated
    -> 1 : (DecDimmer_go' = false);

  [tick] !DecDimmer_go -> 1 : (DecDimmer_go' = true) & (DecDimmer_used' = false);
endmodule

// system
module sys
  servers : [1..MAX_SERVERS] init ini_servers;
  dimmer : [1..DIMMER_LEVELS] init ini_dimmer;

  [AddServer_complete] servers < MAX_SERVERS -> 1 : (servers' = servers + 1);
  [RemoveServer_start] servers > 1 -> 1 : (servers' = servers - 1);
  [IncDimmer_start] dimmer < DIMMER_LEVELS -> 1 : (dimmer' = dimmer + 1);
  [DecDimmer_start] dimmer > 1 -> 1 : (dimmer' = dimmer - 1);
endmodule


// continuous equivalent for the dimmer level
formula dimmerFactor = DIMMER_MARGIN + (1 - 2 * DIMMER_MARGIN) * (dimmer - 1) / (DIMMER_LEVELS - 1);

//*************************************************************
// Queuing model G/G/c LPS with round-robin allocation to servers
//*************************************************************
formula interarrivalMean = stateValue;

// assume arrivals have exponential distribution
formula interArrivalVariance = pow(interarrivalMean, 2);

```

```
formula lambda = 1 / (interarrivalMean * interArrivalScaleFactorForDecision * servers);
formula beta = dimmerFactor * serviceTimeMean + (1-dimmerFactor) * lowServiceTimeMean;
formula rho = lambda * beta;
formula overloaded = (rho >= 1);
formula ca2 = interArrivalVariance * servers / pow(interarrivalMean * servers, 2);
formula cs2 = (dimmerFactor * serviceTimeVariance + (1-dimmerFactor) * lowServiceTimeVariance) / pow(beta, 2);
formula dp = pow(rho, threads * (1+cs2)/(ca2 + cs2));
formula rb = ((ca2 + cs2) / 2) * dp * beta / (1 - rho);
formula rz = ((ca2 + cs2) / (1 + cs2)) * (1 - dp) * beta / (1 - rho);
formula totalTime = rb + rz;
formula rt = (interarrivalMean=0 ? 0 : totalTime);


// Response time to clients utility function
const double SERVER_COST_SEC = 1;
const double MAX_ARRIVAL_CAPACITY = 1/0.04452713;
const double MAX_ARRIVAL_CAPACITY_LOW = 1/0.002430258;

const double NORMAL_REVENUE = 1.5;
const double LOW_REVENUE = 1;
formula poweredServers = (AddServer_state > 0 ? servers + 1 : servers);
formula cost = poweredServers;
formula spacing = MAX_SERVERS + 1;
formula maxThroughput = MAX_SERVERS * MAX_ARRIVAL_CAPACITY;
formula latePenalty = maxThroughput * NORMAL_REVENUE * spacing;

formula throughput = (interarrivalMean > 0) ? 1/interarrivalMean : 0;

formula positiveUtilityTemp = throughput * (dimmerFactor * NORMAL_REVENUE + (1-dimmerFactor) * LOW_REVENUE);
formula positiveUtility = (( positiveUtilityTemp - floor( positiveUtilityTemp ) >= 0.5) ? ceil( positiveUtilityTemp ) : floor(
    positiveUtilityTemp )) * spacing;

formula uTotal = (overloaded) ? (-latePenalty - 2 * spacing + poweredServers + (1-dimmerFactor))
        : ((( rt>RT_THRESHOLD) ? min(0, throughput * NORMAL_REVENUE * spacing - latePenalty) : positiveUtility) -
            cost);

formula periodUtility = (PERIOD)*(uTotal);

formula UTILITY_SHIFT = PERIOD * (latePenalty + 2 * spacing + MAX_SERVERS);

rewards "util"
  [tack] true : UTILITY_SHIFT + periodUtility;
endrewards
```

Appendix B

PLA-PMC PRISM Model for DART

The following listing shows the template for the PLA-PMC PRISM model for DART. This template is completed at run time before each adaptation decision by injecting the initialization block in the tag //# init , and the environment model in the tag //#environment.

```
mdp
//  init block must include values for the following constants
//const int HORIZON
//const double PERIOD
//const int MAX_ALT_LEVEL
//const int  init_a
//const int  init_f
//const double IncAlt_LATENCY
//const double DecAlt_LATENCY
//const int  ini_IncAlt_state
//const int  ini_DecAlt_state
//const double destructionFormationFactor
//const double threatRange
//const double detectionFormationFactor
//const double sensorRange

//# init

//*************************************************************************
//  CLOCK
//*************************************************************************
const int TO_TICK = 0;
const int TO_TICK2 = 1; // intermediate tick for constraint satisf. update
const int TO_TACK = 2;

label "final" = time = HORIZON & clockstep=TO_TICK;
formula sys_go = clockstep=TO_TICK;

module clk
    time : [0.. HORIZON] init 0;
    clockstep : [0..2] init TO_TICK;

    [tick] clockstep=TO_TICK & time < HORIZON -> 1: (time'=time+1) & (clockstep'=TO_TICK2);
    [tick2] clockstep=TO_TICK2 -> 1 : (clockstep'=TO_TACK);
    [tack] clockstep=TO_TACK -> 1: (clockstep'=TO_TICK);
endmodule

module env
//#environment

//*************************************************************************
//  SYSTEM
```

```
43 //***********************************************************************
44
45 // Variable range and initialization
46 const a_MIN=0; const a_MAX=MAX_ALT_LEVEL; const a_INIT=init_a;
47 const f_MIN=0; const f_MAX=1; const f_INIT=init_f;
48
49 module sys
50   a : [a_MIN..a_MAX] init a_INIT;
51   f : [f_MIN..f_MAX] init f_INIT;
52
53   [GoTight_start] f=0 -> 1: (a'=a_GoTight_impact)
54       & (f'=f_GoTight_impact);
55   [GoLoose_start] f=1 -> 1: (a'=a_GoLoose_impact)
56       & (f'=f_GoLoose_impact);
57   [IncAlt_complete] a < MAX_ALT_LEVEL -> 1: (a'=a_IncAlt_impact)
58     & (f'=f_IncAlt_impact);
59   [DecAlt_complete] a > 0 -> 1: (a'=a_DecAlt_impact)
60       & (f'=f_DecAlt_impact);
61 endmodule
62
63
64 formula a_GoTight_impact = a + (0) >= a_MIN ? ( a+(0)<=a_MAX? a+(0) : a_MAX) : a_MIN;
65 formula f_GoTight_impact = f + (1) >= f_MIN ? ( f+(1)<=f_MAX? f+(1) : f_MAX) : f_MIN;
66 formula a_GoLoose_impact = a + (0) >= a_MIN ? ( a+(0)<=a_MAX? a+(0) : a_MAX) : a_MIN;
67 formula f_GoLoose_impact = f + (-1) >= f_MIN ? ( f+(-1)<=f_MAX? f+(-1) : f_MAX) : f_MIN;
68 formula a_IncAlt_impact = a + (1) >= a_MIN ? ( a+(1)<=a_MAX? a+(1) : a_MAX) : a_MIN;
69 formula f_IncAlt_impact = f + (0) >= f_MIN ? ( f+(0)<=f_MAX? f+(0) : f_MAX) : f_MIN;
70 formula a_DecAlt_impact = a + (-1) >= a_MIN ? ( a+(-1)<=a_MAX? a+(-1) : a_MAX) : a_MIN;
71 formula f_DecAlt_impact = f + (0) >= f_MIN ? ( f+(0)<=f_MAX? f+(0) : f_MAX) : f_MIN;
72
73
74 // tactic concurrency rules
75 formula IncAlt_used = IncAlt_state != 0;
76 formula DecAlt_used = DecAlt_state != 0;
77
78 formula GoTight_compatible = !GoLoose_used;
79 formula GoLoose_compatible = !GoTight_used;
80 formula IncAlt_compatible = !DecAlt_used;
81 formula DecAlt_compatible = !IncAlt_used;
82
83 //***********************************************************************
84 // TACTIC: GoTight
85 //***********************************************************************
86
87 // Applicability conditions
88 formula GoTight_applicable = GoTight_compatible & f=0;
89
90 module GoTight
91     GoTight_used : bool init false;
92     GoTight_go : bool init true;
93
94     // Tactic applicable, start it
95     [GoTight_start] sys_go & GoTight_go & GoTight_applicable -> (GoTight_used'=true) & (GoTight_go'=false);
96
97     // Tactic applicable, but do not start it
98     [] sys_go & GoTight_go & GoTight_applicable -> (GoTight_go'=false);
99
100    // Pass if the tactic is not applicable
101    [] sys_go & GoTight_go & !GoTight_applicable -> 1 : (GoTight_go'=false);
102
103    [tick] !GoTight_go -> 1: (GoTight_go'=true) & (GoTight_used'=false);
104 endmodule
105
106
107 //***********************************************************************
108 // TACTIC: GoLoose
```

```
109 //*********************************************************************
110
111 //  Applicability  conditions
112 formula GoLoose_applicable = GoLoose_compatible & f=1;
113
114 module GoLoose
115     GoLoose_used : bool init false;
116     GoLoose_go : bool init true;
117
118     // Tactic applicable, start it
119     [GoLoose_start] sys_go & GoLoose_go & GoLoose_applicable -> (GoLoose_used'=true) & (GoLoose_go'=false);
120
121     // Tactic applicable, but do not start it
122     [] sys_go & GoLoose_go & GoLoose_applicable -> (GoLoose_go'=false);
123
124     // Pass if the tactic is not applicable
125     [] sys_go & GoLoose_go & !GoLoose_applicable -> 1 : (GoLoose_go'=false);
126
127     [tick] !GoLoose_go -> 1: (GoLoose_go'=true) & (GoLoose_used'=false);
128 endmodule
129
130
131 //*********************************************************************
132 //  TACTIC: IncAlt
133 //*********************************************************************
134
135 const int IncAlt_LATENCY_PERIODS = ceil(IncAlt_LATENCY/PERIOD);
136
137 //  Applicability  conditions
138 formula IncAlt_applicable = IncAlt_compatible & a < MAX_ALT_LEVEL;
139
140 module IncAlt
141     IncAlt_state : [0..IncAlt_LATENCY_PERIODS] init ini_IncAlt_state;
142     IncAlt_go : bool init true;
143
144     // Tactic applicable, start it
145     [IncAlt_start] sys_go & IncAlt_go & IncAlt_state=0 & IncAlt_applicable -> (IncAlt_state'=IncAlt_LATENCY_PERIODS) & (
            IncAlt_go'=false);
146
147     // Tactic applicable, but do not start it
148     [] sys_go & IncAlt_go & IncAlt_state=0 & IncAlt_applicable -> (IncAlt_go'=false);
149
150     // Pass if the tactic is not applicable
151     [] sys_go & IncAlt_go & IncAlt_state=0 & !IncAlt_applicable -> 1 : (IncAlt_go'=false);
152
153     // Progress of the tactic
154     [] sys_go & IncAlt_go & IncAlt_state > 1 -> 1: (IncAlt_state'= IncAlt_state -1) & (IncAlt_go'=false);
155
156     // Completion of the tactic
157     [IncAlt_complete] sys_go & IncAlt_go & IncAlt_state=1 -> 1: (IncAlt_state'=0) & (IncAlt_go'=true);
158
159     [tick] !IncAlt_go -> 1: (IncAlt_go'=true);
160 endmodule
161
162
163 //*********************************************************************
164 //  TACTIC: DecAlt
165 //*********************************************************************
166
167 const int DecAlt_LATENCY_PERIODS = ceil(DecAlt_LATENCY/PERIOD);
168
169 //  Applicability  conditions
170 formula DecAlt_applicable = DecAlt_compatible & a > 0;
171
172 module DecAlt
173     DecAlt_state : [0..DecAlt_LATENCY_PERIODS] init ini_DecAlt_state;
```

```
174     DecAlt_go : bool init true;
175
176     // Tactic applicable, start it
177     [DecAlt_start] sys_go & DecAlt_go & DecAlt_state=0 & DecAlt_applicable -> (DecAlt_state'=DecAlt_LATENCY_PERIODS) & (
            DecAlt_go'=false);
178
179     // Tactic applicable, but do not start it
180     [] sys_go & DecAlt_go & DecAlt_state=0 & DecAlt_applicable -> (DecAlt_go'=false);
181
182     // Pass if the tactic is not applicable
183     [] sys_go & DecAlt_go & DecAlt_state=0 & !DecAlt_applicable -> 1 : (DecAlt_go'=false);
184
185     // Progress of the tactic
186     [] sys_go & DecAlt_go & DecAlt_state > 1 -> 1: (DecAlt_state'=DecAlt_state-1) & (DecAlt_go'=false);
187
188     // Completion of the tactic
189     [DecAlt_complete] sys_go & DecAlt_go & DecAlt_state=1 -> 1: (DecAlt_state'=0) & (DecAlt_go'=true);
190
191     [tick] !DecAlt_go -> 1: (DecAlt_go'=true);
192 endmodule
193
194 //*******************************
195 //  Utility Function
196 //*******************************
197 const int LOOSE = 0;
198 const int TIGHT = 1;
199
200 formula probOfThreat = stateValue;
201
202 formula probabilityOfDestruction = probOfThreat
203   * ((f = LOOSE) ? 1.0 : (1.0 / destructionFormationFactor))
204     * max(0.0, threatRange - (a + 1)) / threatRange; // +1 because level 0 is one level above ground
205
206 module constraint // in this case the constraint is surviving
207   satisfied : bool init true;
208   [tick2] satisfied -> (1.0 - probabilityOfDestruction): (satisfied'=true)
209     + probabilityOfDestruction: (satisfied'=false);
210   [tick2] !satisfied -> true;
211 endmodule
212
213
214 formula probOfTarget= stateValue1;
215
216 formula probOfDetection = probOfTarget
217   * ((f = LOOSE) ? 1.0 : (1.0 / detectionFormationFactor))
218     * max(0.0, sensorRange - (a + 1)) / sensorRange; // +1 because level 0 is one level above ground
219
220 module sensor
221   targetDetected: bool init false;
222   [tick2] true -> probOfDetection: (targetDetected'=true) + (1.0 - probOfDetection): (targetDetected'=false);
223 endmodule
224
225 rewards "util"
226   [tack] satisfied & targetDetected : 1;
227
228   // give slight preference to not adapting
229   [tick] time = 0 & IncAlt_state = ini_IncAlt_state & DecAlt_state=ini_DecAlt_state & a=init_a & f= init_f : 0.000000001;
230 endrewards
```

Appendix C

PLA-SDP Alloy Models for RUBiS

C.1 Immediate Reachability Model

The following listing contains the PLA-SDP Alloy model for computing immediate reachability for RUBiS.

```
1  open util/ordering[S] as servers
2  open util/ordering[TAP] as progress // tactic add progress
3  open util/ordering[TraceElement] as trace
4  open util/ordering[D] as dimmer
5  open util/ordering[T] as TO
6
7  abstract sig TP {} // tactic progress
8  sig TAP extends TP {} // one sig for each tactic with latency
9
10 abstract sig T {} // tactics
11 abstract sig LT extends T {} // tactics with latency
12 one sig IncDimmer, DecDimmer, RemoveServer extends T {} // tactics with no latency
13 one sig AddServer extends LT {} // tactics with latency
14
15 // define configuration properties
16 sig S {} // the different number of active servers
17 sig D {} // the different dimmer levels
18
19 /* each element of C represents a configuration */
20 abstract sig C {
21     s : S, // the number of active servers
22     d : D // dimmer level
23 }
24
25 pred equals[c, c2 : C] {
26     all f : C$.fields | c.(f.value) = c2.(f.value)
27 }
28
29 pred equalsExcept[c, c2 : C, ef : univ] {
30     all f : C$.fields | f=ef or c.(f.value) = c2.(f.value)
31 }
32
33 /*
34  * this sig is a config extended with the progress of each tactic with latency
35  */
36 sig CP extends C {
37     p: LT -> TP
38 } {
39     ~p.p in iden // functional (i.e., p maps each tactic to at most one progress)
40     //#p = #LT // every tactic in LT has a mapping in p
```

```
41      p.univ = LT // every tactic in LT has a mapping in p (p.univ is domain(p) )
42      p[AddServer] in TAP // restrict each tactic to its own progress class
43  }
44
45  fact tacticOrdering {
46      TO/first = AddServer
47      AddServer.next = RemoveServer
48      RemoveServer.next = IncDimmer
49      IncDimmer.next = DecDimmer
50  }
51
52  sig TraceElement {
53      cp : CP,
54      starts : set T // tactics started
55  }
56
57  // do not generate atoms that do not belong to the trace
58  fact {
59      CP in TraceElement.cp
60  }
61
62  pred equals[e, e2 : TraceElement] {
63      all f : TraceElement$.subfields | e.(f.value) = e2.(f.value)
64  }
65
66  fact traces {
67      let fst = trace/first | fst.starts = none
68      all e : TraceElement - last | let e' = next[e] | {
69          equals[e, e']
70          equals[e', trace/last]
71      } or  ((addServerTacticStart[e, e'] or removeServerTactic[e, e'] or decDimmerTactic[e, e'] or incDimmerTactic[e, e']) and
72          (let s = e'.starts - e.starts | all t : s | validOrder[t, e]))
73  }
74
75  pred validOrder[t : T, e : TraceElement] {
76      all s : T | s in e.starts => !(s in t.nexts)
77  }
78
79  pred addServerCompatible[e : TraceElement] {
80      e.cp.p[AddServer] = progress/last
81      !(RemoveServer in e.starts)
82  }
83
84  pred addServerTacticStart[e, e' : TraceElement] {
85      addServerCompatible[e] and e.cp.s != servers/last
86      e'.starts = e.starts + AddServer
87      let c = e.cp, c'=e'.cp | {
88          c'.p[AddServer] = progress/first
89
90          // nothing else changes
91          equals[c, c']
92          (LT - AddServer) <: c.p in c'.p
93      }
94  }
95
96  pred removeServerCompatible[e : TraceElement] {
97      !(RemoveServer in e.starts)
98      e.cp.p[AddServer] = progress/last // add server tactic not running
99  }
100
101 pred removeServerTactic[e, e' : TraceElement] {
102     removeServerCompatible[e] and e.cp.s != servers/first
103     e'.starts = e.starts + RemoveServer
104     let c = e.cp, c'=e'.cp | {
105         c'.s = servers/prev[c.s]
106
```

```
107             // nothing else changes
108             equalsExcept[c, c', C$s]
109             c'.p = c.p
110         }
111 }
112
113 pred incDimmerCompatible[e : TraceElement] {
114     !(IncDimmer in e.starts) and !(DecDimmer in e.starts)
115 }
116
117 pred incDimmerTactic[e, e' : TraceElement] {
118     incDimmerCompatible[e] and e.cp.d != dimmer/last
119     e'.starts = e.starts + IncDimmer
120
121     let c = e.cp, c'=e'.cp | {
122         c'.d = c.d.next
123
124         // nothing else changes
125         equalsExcept[c, c', C$d]
126         c'.p = c.p
127     }
128 }
129
130 pred decDimmerCompatible[e : TraceElement] {
131     !(DecDimmer in e.starts) and !(IncDimmer in e.starts)
132 }
133
134 pred decDimmerTactic[e, e' : TraceElement] {
135     decDimmerCompatible[e] and e.cp.d != dimmer/first
136     e'.starts = e.starts + DecDimmer
137
138     let c = e.cp, c'=e'.cp | {
139         c'.d = c.d.prev
140
141         // nothing else changes
142         equalsExcept[c, c', C$d]
143         c'.p = c.p
144     }
145 }
146
147
148 pred show {
149 }
150
151 // the scope for TraceElement, C and CP has to be one more than the maximum
152 // number of tactics that could be started concurrently
153 run show for exactly 3 S, exactly 3 TAP, 2 D, 3 C, 3 CP, 3 TraceElement
```

C.2 Delayed Reachability Model

The following listing contains the PLA-SDP Alloy model for computing delayed reachability for RUBiS.

```
1 open util/ordering[S] as servers
2 open util/ordering[TAP] as progress // tactic add progress
3 open util/ordering[D] as dimmer
4
5 abstract sig TP {} // tactic progress
6 sig TAP extends TP {} // one sig for each tactic with latency
7
8 abstract sig T {} // tactics
9 abstract sig LT extends T {} // tactics with latency
```

```
10 one sig IncDimmer, DecDimmer, RemoveServer extends T {} // tactics with no latency
11 one sig AddServer extends LT {} // tactics with latency
12
13 // define configuration properties
14 sig S {} // the different number of active servers
15 sig D {} // the different dimmer levels
16
17 /* each element of C represents a configuration */
18 abstract sig C {
19     s : S, // the number of active servers
20     d : D // dimmer level
21 }
22
23 pred equals[c, c2 : C] {
24     all f : C$.fields | c.(f.value) = c2.(f.value)
25 }
26
27 pred equalsExcept[c, c2 : C, ef : univ] {
28     all f : C$.fields | f=ef or c.(f.value) = c2.(f.value)
29 }
30
31
32 fact uniqueInstances { all disj c, c2 : CP | !equals[c, c2] or c.p != c2.p}
33
34
35 /*
36  * this sig is a config extended with the progress of each tactic with latency
37  */
38 sig CP extends C {
39     p: LT -> TP
40 } {
41     ~p.p in iden // functional (i.e., p maps each tactic to at most one progress)
42     //#p = #LT // every tactic in LT has a mapping in p
43     p.univ = LT // every tactic in LT has a mapping in p (p.univ is domain(p) )
44     p[AddServer] in TAP // restrict each tactic to its own progress class
45 }
46
47
48 pred addServerTacticProgress[c, c' : CP] {
49     c.p[AddServer] != progress/last implies { // tactic is running
50         c'.p[AddServer] = progress/next[c.p[AddServer]]
51         c'.p[AddServer] = progress/last implies c'.s = servers/next[c.s] else c'.s = c.s
52     } else {
53         c'.p[AddServer] = progress/last // stay in not running state
54         c'.s = c.s
55     }
56
57     // nothing else changes other than s and the progress
58     equalsExcept[c, c', C$s]
59     (LT - AddServer) <: c.p in c'.p
60 }
61
62 pred oneStepProgress[c, c' : CP] { // is c' reachable from config c in one evaluation period?
63     addServerTacticProgress[c, c']
64 }
65
66 sig Result {
67     c, c' : CP
68 } {
69     oneStepProgress[c, c']
70 }
71
72 // this reduces the number of unused configurations
73 fact reduceUsedConfigs {
74     all cp : CP | {some r : Result | r.c = cp or r.c' = cp
75         }
```

```
76 }
77
78 pred show {
79 }
80
81 /*
82  * (numOfTacticsWithLatency + 1) for CP and C to allow the progress for all the tactics with latency + the initial state
83  */
84 run show for exactly 3 S, exactly 3 TAP, exactly 2 D, 2 C, 2 CP, exactly 1 Result
```

Appendix D

PLA-SDP Alloy Models for DART

D.1 Immediate Reachability Model

The following listing contains the PLA-SDP Alloy model for computing immediate reachability for DART.

```
1  open util/ordering[F] as FO
2  open util/ordering[TraceElement] as trace
3  open util/ordering[A] as AO
4  open util/ordering[TPIA] as TPIAO
5  open util/ordering[TPDA] as TPDAO
6  open util/ordering[T] as TO
7
8  abstract sig TP {} // tactic progress
9  sig TPIA extends TP {} // one sig for each tactic with latency
10 sig TPDA extends TP {} // one sig for each tactic with latency
11
12 abstract sig T {} // tactics
13 abstract sig LT extends T {} // tactics with latency
14 one sig GoLoose, GoTight extends T {} // tactics with no latency
15 one sig IncAlt, DecAlt extends LT {} // tactics with latency
16
17 // define configuration properties
18 sig F {} // the different formations
19 sig A {} // the different altitude levels
20
21 /* each element of C represents a configuration */
22 abstract sig C {
23     f : F, // formation
24     a : A // altitude level
25 }
26
27 pred equals[c, c2 : C] {
28     all f : C$.fields | c.(f.value) = c2.(f.value)
29 }
30
31 pred equalsExcept[c, c2 : C, ef : univ] {
32     all f : C$.fields | f=ef or c.(f.value) = c2.(f.value)
33 }
34
35 /*
36  * this sig is a config extended with the progress of each tactic with latency
37  */
38 sig CP extends C {
39     p: LT -> TP
40 } {
```

```
41      ~p.p in iden // functional (i.e., p maps each tactic to at most one progress)
42      p.univ = LT // every tactic in LT has a mapping in p
43      p[IncAlt] in TPIA // restrict each tactic to its own progress class
44      p[DecAlt] in TPDA
45  }
46
47  fact tacticOrdering {
48      TO/first = GoLoose
49      GoLoose.next = GoTight
50      GoTight.next = IncAlt
51      IncAlt.next = DecAlt
52  }
53
54  sig TraceElement {
55      cp : CP,
56      starts : set T // tactics started
57  }
58
59  // do not generate atoms that do not belong to the trace
60  fact {
61      CP in TraceElement.cp
62  }
63
64  pred equals[e, e2 : TraceElement] {
65      all f : TraceElement$.subfields | e.(f.value) = e2.(f.value)
66  }
67
68  fact traces {
69      let fst = trace/first | fst.starts = none
70      all e : TraceElement − last | let e' = next[e] | {
71          equals[e, e']
72          equals[e', trace/last]
73      } or  ((incAltTacticStart[e, e'] or decAltTacticStart[e, e'] or goLooseTactic[e, e'] or goTightTactic[e, e']) and (
74          let s = e'.starts − e.starts | all t : s | validOrder[t, e]))
75  }
76
77  pred validOrder[t : T, e : TraceElement] {
78      all s : T | s in e.starts => !(s in t.nexts)
79  }
80
81  pred incAltCompatible[e : TraceElement] {
82      e.cp.p[IncAlt] = TPIAO/last // IncAlt tactic not running
83      e.cp.p[DecAlt] = TPDAO/last // DecAlt tactic not running
84  }
85
86  pred incAltTacticStart[e, e' : TraceElement] {
87      incAltCompatible[e] and e.cp.a != AO/last
88      e'.starts = e.starts + IncAlt
89      let c = e.cp, c'=e'.cp | {
90          c'.p[IncAlt] = TPIAO/first
91
92          // nothing else changes
93          equals[c, c']
94          (LT − IncAlt) <: c.p in c'.p
95      }
96  }
97
98  pred decAltCompatible[e : TraceElement] {
99      e.cp.p[DecAlt] = TPDAO/last // DecAlt tactic not running
100     e.cp.p[IncAlt] = TPIAO/last // IncAlt tactic not running
101 }
102
103 pred decAltTacticStart[e, e' : TraceElement] {
104     decAltCompatible[e] and e.cp.a != AO/first
105     e'.starts = e.starts + DecAlt
106     let c = e.cp, c'=e'.cp | {
```

```
        c'.p[DecAlt] = TPDAO/first

        // nothing else changes
        equals[c, c']
        (LT − DecAlt) <: c.p in c'.p
    }
}

pred goLooseCompatible[e : TraceElement] {
    !(GoLoose in e.starts) and !(GoTight in e.starts)
}

pred goLooseTactic[e, e' : TraceElement] {
    goLooseCompatible[e] and e.cp.f != FO/first
    e'.starts = e.starts + GoLoose

    let c = e.cp, c'=e'.cp | {
        c'.f = FO/first

        // nothing else changes
        equalsExcept[c, c', C$f]
        c'.p = c.p
    }
}

pred goTightCompatible[e : TraceElement] {
    !(GoTight in e.starts) and !(GoLoose in e.starts)
}

pred goTightTactic[e, e' : TraceElement] {
    goTightCompatible[e] and e.cp.f != FO/last
    e'.starts = e.starts + GoTight

    let c = e.cp, c'=e'.cp | {
        c'.f = FO/last

        // nothing else changes
        equalsExcept[c, c', C$f]
        c'.p = c.p
    }
}

pred show {
}

// the scope for TraceElement, C and CP has to be one more than the maximum
// number of tactics that could be started concurrently
run show for exactly 3 A, exactly 2 TPIA, exactly 2 TPDA, exactly 2 F, 3 C, 3 CP, 3 TraceElement
```

D.2 Delayed Reachability Model

The following listing contains the PLA-SDP Alloy model for computing delayed reachability for DART.

```
open util/ordering[F] as FO
open util/ordering[A] as AO
open util/ordering[TPIA] as TPIAO
open util/ordering[TPDA] as TPDAO

abstract sig TP {} // tactic progress
sig TPIA extends TP {} // one sig for each tactic with latency
sig TPDA extends TP {} // one sig for each tactic with latency
```

```
9
10 abstract sig T {} // tactics
11 abstract sig LT extends T {} // tactics with latency
12 one sig GoLoose, GoTight extends T {} // tactics with no latency
13 one sig IncAlt, DecAlt extends LT {} // tactics with latency
14
15 // define configuration properties
16 sig F {} // the different formations
17 sig A {} // the different altitude levels
18
19 /* each element of C represents a configuration */
20 abstract sig C {
21     f : F, // formation
22     a : A // altitude level
23 }
24
25 pred equals[c, c2 : C] {
26     all f : C$.fields | c.(f.value) = c2.(f.value)
27 }
28
29 pred equalsExcept[c, c2 : C, ef : univ] {
30     all f : C$.fields | f=ef or c.(f.value) = c2.(f.value)
31 }
32
33 fact uniqueInstances { all disj c, c2 : CP | !equals[c, c2] or c.p != c2.p}
34
35 /*
36  * this sig is a config extended with the progress of each tactic with latency
37  */
38 sig CP extends C {
39     p: LT -> TP
40 } {
41     ~p.p in iden // functional (i.e., p maps each tactic to at most one progress)
42     p.univ = LT // every tactic in LT has a mapping in p
43     p[IncAlt] in TPIA // restrict each tactic to its own progress class
44     p[DecAlt] in TPDA
45 }
46
47 pred incAltTacticProgress[c, c' : CP] {
48     c.p[IncAlt] != TPIAO/last implies { // tactic is running
49         c'.p[IncAlt] = TPIAO/next[c.p[IncAlt]]
50         c'.p[IncAlt] = TPIAO/last implies c'.a = AO/next[c.a] else c'.a = c.a
51     } else {
52         c'.p[IncAlt] = TPIAO/last // stay in not running state
53         c'.a = c.a
54     }
55
56     // nothing else changes other than s and the progress
57     equalsExcept[c, c', C$a]
58     (LT - IncAlt) <: c.p in c'.p
59 }
60
61 pred decAltTacticProgress[c, c' : CP] {
62     c.p[DecAlt] != TPDAO/last implies { // tactic is running
63         c'.p[DecAlt] = TPDAO/next[c.p[DecAlt]]
64         c'.p[DecAlt] = TPDAO/last implies c'.a = AO/prev[c.a] else c'.a = c.a
65     } else {
66         c'.p[DecAlt] = TPDAO/last // stay in not running state
67         c'.a = c.a
68     }
69
70     // nothing else changes other than s and the progress
71     equalsExcept[c, c', C$a]
72     (LT - DecAlt) <: c.p in c'.p
73 }
74
```

```
75 pred oneStepProgress[c, c' : CP] { // is c' reachable from config c in one evaluation period?
76     some tc : CP | incAltTacticProgress[c, tc] and decAltTacticProgress[tc, c']
77 }
78
79 sig Result {
80     c, c' : CP
81 } {
82     oneStepProgress[c, c']
83 }
84
85 // this reduces the number of unused configurations
86 // each cp in CP is either in a pair in a result, or an intermediate one needed for that pair
87 fact reduceUsedConfigs {
88     all cp : CP | {some r : Result | r.c = cp or r.c' = cp
89                 or (incAltTacticProgress[r.c, cp] and decAltTacticProgress[cp, r.c'])
90         }
91 }
92
93 pred show {
94 }
95
96 /*
97  * (numOfTacticsWithLatency + 1) for CP and C to allow the progress for all the tactics with latency + the initial state
98  */
99 run show for exactly 3 A, exactly 2 TPIA, exactly 2 TPDA, exactly 2 F, 3 C, 3 CP, exactly 1 Result
```

Bibliography

[1] Mehdi Amoui, Mazeiar Salehie, Siavash Mirarab, and Ladan Tahvildari. Adaptive action selection in autonomic software using reinforcement learning. In *Fourth International Conference on Autonomic and Autonomous Systems (ICAS'08)*, pages 175–181. IEEE, March 2008. ISBN 978-0-7695-3093-2. doi: 10.1109/ICAS.2008.35. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4488342. 2.5

[2] Konstantinos Angelopoulos, Vítor E. Silva Souza, and John Mylopoulos. Dealing with multiple failures in Zanshin: a control-theoretic approach. In *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems - SEAMS 2014*, pages 165–174, New York, New York, USA, 2014. ACM Press. ISBN 9781450328647. doi: 10.1145/2593929.2593936. URL http://dl.acm.org/citation.cfm?doid=2593929.2593936. 1

[3] Konstantinos Angelopoulos, Alessandro V. Papadopoulos, Vítor E. Silva Souza, and John Mylopoulos. Model predictive control for software systems with CobRA. In *Proceedings of the 11th International Workshop on Software Engineering for Adaptive and Self-Managing Systems - SEAMS '16*, pages 35–46, Austin, Texas, 2016. ACM Press. ISBN 9781450341875. doi: 10.1145/2897053.2897054. URL http://dl.acm.org/citation.cfm?doid=2897053.2897054. 2.4, 9.3, 9.4

[4] Martin F. Arlitt and T. Jin. A workload characterization study of the 1998 World Cup web site. *IEEE Network*, 14(3):30–37, 2000. ISSN 08908044. doi: 10.1109/65.844498. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=844498. 8.2.1

[5] Martin F. Arlitt and Carey L. Williamson. Web server workload characterization. *Proceedings of the 1996 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems - SIGMETRICS '96*, 24:126–137, May 1996. ISSN 0163-5999. doi: 10.1145/233013.233034. URL http://dl.acm.org/citation.cfm?id=233013.233034. 8.2.1

[6] Stanley S. Baek, Hyukseong Kwon, Josiah A. Yoder, and Daniel Pack. Optimal path planning of a target-following fixed-wing UAV using sequential decision processes. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2955–2962. IEEE, November 2013. ISBN 978-1-4673-6358-7. doi: 10.1109/IROS.2013.6696775. URL http://ieeexplore.ieee.org/document/6696775/. 9.3, 9.4

[7] Hamid Bagheri and Sam Malek. Titanium: Efficient analysis of evolving Alloy specifi-

cations. In *Proceedings of the 2015 24th International Symposium on the Foundations of Software Engineering (FSE 2016)*. ACM Press, 2016. 9.3

[8] Chris Baker, Gopal Ramchurn, Luke Teacy, and Nicholas Jennings. Planning search and rescue missions for UAV teams. In *PAIS 2016: Conference on Prestigious Applications of Intelligent Systems at ECAI 2016*, The Hague, NL, 2016. IOS Press. 8.2.3

[9] Enda Barrett, Enda Howley, and Jim Duggan. Applying reinforcement learning towards automating resource allocation and application scalability in the cloud. *Concurrency and Computation: Practice and Experience*, 25(12):1656–1674, August 2013. ISSN 15320626. doi: 10.1002/cpe.2864. URL `http://doi.wiley.com/10.1002/cpe.2864`. 2.5

[10] Richard Bellman. Some applications of the theory of dynamic programming—a review. *Journal of the Operations Research Society of America*, 2(3):275–288, 1954. ISSN 00963984. URL `http://www.jstor.org/stable/166640`. 9.3

[11] Carlo Bertolli, Gabriele Mencagli, and Marco Vanneschi. A cost model for autonomic reconfigurations in high-performance pervasive applications. In *Proceedings of the 4th ACM International Workshop on Context-Awareness for Self-Managing Systems - CASEMANS '10*, pages 20–29, New York, New York, USA, September 2010. ACM Press. ISBN 9781450302135. doi: 10.1145/1858367.1858370. URL `http://dl.acm.org/citation.cfm?id=1858367.1858370`. 2.3

[12] L.F. Bertuccelli and J.P. How. Robust UAV search for environments with imprecise probability maps. In *Proceedings of the 44th IEEE Conference on Decision and Control*, pages 5680–5685. IEEE, 2005. ISBN 0-7803-9567-0. doi: 10.1109/CDC.2005.1583068. URL `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1583068`. 8.1.2

[13] Andrea Bianco and Luca Alfaro. Model checking of probabilistic and nondeterministic systems. In *Foundations of Software Technology and Theoretical Computer Science*, pages 499–513, Bangalore, India, 1995. Springer Berlin Heidelberg. doi: 10.1007/3-540-60692-0_70. URL `http://link.springer.com/10.1007/3-540-60692-0_70`. 4.1

[14] David P. Biros, Mark Daly, and Gregg Gunsch. The influence of task load and automation trust on deception detection. *Group Decision and Negotiation*, 13(2):173–189, March 2004. ISSN 0926-2644. doi: 10.1023/B:GRUP.0000021840.85686.57. URL `http://link.springer.com/10.1023/B:GRUP.0000021840.85686.57`. 7

[15] Yuriy Brun, Giovanna Di Marzo Serugendo, Cristina Gacek, Holger Giese, Holger M. Kienle, Marin Litoiu, Hausi A. Müller, Mauro Pezzè, and Mary Shaw. Engineering self-adaptive systems through feedback loops. In *Software Engineering for Self-Adaptive Systems*, volume 5525, pages 48–70. Springer Berlin Heidelberg, 2009. URL `http://link.springer.com/chapter/10.1007/978-3-642-02161-9_3`. 1, 2.1, 9.3, 9.4

[16] Tomas Bures, Ilias Gerostathopoulos, Petr Hnetynka, Jaroslav Keznikl, Michal Kit, and Frantisek Plasil. DEECo—an ensemble-based component system. In *Proceedings of*

the 16th International ACM SIGSOFT Symposium on Component-Based Software Engineering - CBSE '13, page 81, New York, New York, USA, June 2013. ACM Press. ISBN 9781450321228. doi: 10.1145/2465449.2465462. URL http://dl.acm.org/citation.cfm?id=2465449.2465462. 1

[17] Radu Calinescu, Lars Grunske, Marta Kwiatkowska, Raffaela Mirandola, and Giordano Tamburrelli. Dynamic QoS management and optimization in service-based systems. *IEEE Transactions on Software Engineering*, 37(3):387–409, May 2011. ISSN 0098-5589. doi: 10.1109/TSE.2010.92. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5611553. 2.2

[18] Radu Calinescu, Carlo Ghezzi, Marta Kwiatkowska, and Raffaela Mirandola. Self-adaptive software needs quantitative verification at runtime. *Communications of the ACM*, 55(9):69, September 2012. ISSN 00010782. doi: 10.1145/2330667.2330686. URL http://dl.acm.org/ft_gateway.cfm?id=2330686&type=html. 1.2, 2.6

[19] Eduardo F. Camacho and Carlos Bordons Alba. *Model Predictive Control*. Springer, 2013. ISBN 0857293982. 2.4, 9.3

[20] Javier Cámara, Pedro Correia, Rogério de Lemos, David Garlan, Pedro Gomes, Bradley Schmerl, and Rafael Ventura. Evolving an adaptive industrial software system to use architecture-based self-adaptation. In *2013 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 13–22. IEEE, May 2013. ISBN 978-1-4673-4401-2. doi: 10.1109/SEAMS.2013.6595488. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6595488. 1

[21] Javier Cámara, Antonia Lopes, David Garlan, and Bradley Schmerl. Impact models for architecture-based self-adaptive systems. In *Proceedings of the 11th International Symposium on Formal Aspects of Component Software (FACS2014)*, Bertinoro, Italy, 2014. 7.1

[22] 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 - SEAMS 2014*, pages 155–164, New York, New York, USA, June 2014. ACM. ISBN 9781450328647. doi: 10.1145/2593929.2593933. URL http://dl.acm.org/citation.cfm?id=2593929.2593933. 1.5

[23] Javier Cámara, Gabriel A. Moreno, David Garlan, and Bradley Schmerl. Analyzing latency-aware self-adaptation using stochastic games and simulations. *ACM Transactions on Autonomous and Adaptive Systems*, 10(4):1–28, January 2016. ISSN 15564665. doi: 10.1145/2774222. URL http://dl.acm.org/citation.cfm?id=2872308.2774222. 1.5, 7

[24] Sagar Chaki and David Kyle. DMPL: Programming and verifying distributed mixed-synchrony and mixed-critical software. Technical Report CMU/SEI-2016-TR-005, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2016. URL http://resources.sei.cmu.edu/library/asset-view.

cfm?AssetID=464254. 8.2.2

[25] Huoping Chen and Salim Hariri. An evaluation scheme of adaptive configuration techniques. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 493–496, Atlanta, Georgia, USA, 2007. ACM. URL http://doi.acm.org/10.1145/1321631.1321717. 2.3

[26] Betty H. C. Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, Jeff Magee, Jesper Andersson, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Giovanna Marzo Serugendo, Schahram Dustdar, Anthony Finkelstein, Cristina Gacek, Kurt Geihs, Vincenzo Grassi, Gabor Karsai, Holger M. Kienle, Jeff Kramer, Marin Litoiu, Sam Malek, Raffaela Mirandola, Hausi A. Müller, Sooyong Park, Mary Shaw, Matthias Tichy, Massimo Tivoli, Danny Weyns, and Jon Whittle. Software engineering for self-adaptive systems: A research roadmap. In Betty H. C. Cheng, Rogério Lemos, Holger Giese, Paola Inverardi, and Jeff Magee, editors, *Software Engineering for Self-Adaptive Systems*, volume 5525 of *Lecture Notes in Computer Science*, pages 1–26. Springer Berlin Heidelberg, Berlin, Heidelberg, June 2009. ISBN 978-3-642-02160-2. doi: 10.1007/978-3-642-02161-9. URL http://dl.acm.org/citation.cfm?id=1573856.1573858. 1, 9.4

[27] Shang-Wen Cheng. *Rainbow: Cost-Effective Software Architecture-Based Self-Adaptation*. PhD thesis, Carnegie Mellon University, May 2008. URL http://reports-archive.adm.cs.cmu.edu/anon/isr2008/abstracts/08-113.html. 7, 7, 7

[28] Shang-Wen Cheng and David Garlan. Stitch: A language for architecture-based self-adaptation. *Journal of Systems and Software*, 85(12):2860–2875, December 2012. ISSN 01641212. doi: 10.1016/j.jss.2012.02.060. URL http://dl.acm.org/citation.cfm?id=2381464.2381594. (document), 1, 1.3, 3.4, 7, 7.1, 7.1, 7.1, 9.3

[29] Shang-Wen Cheng, David Garlan, Bradley Schmerl, João Pedro Sousa, Bridget Spitznagel, and Peter Steenkiste. Using architectural style as a basis for system self-repair. In *Software Architecture*, pages 45–59. Springer US, Boston, MA, 2002. doi: 10.1007/978-0-387-35607-5_3. URL http://link.springer.com/10.1007/978-0-387-35607-5_3. 7.1

[30] 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. IEEE, May 2009. ISBN 978-1-4244-3724-5. doi: 10.1109/SEAMS.2009.5069082. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5069082. 3, 1.2

[31] A. Davtyan, S. Hoffmann, and R. Scheuring. Optimization of model predictive control by means of sequential parameter optimization. In *Computational Intelligence in Control and Automation (CICA)*, pages 11–16. IEEE, April 2011. ISBN 978-1-4244-9902-1. doi: 10.1109/CICA.2011.5945754. URL http://ieeexplore.ieee.org/document/5945754/. 9.3

[32] Pieter-Tjerk de Boer, Dirk P. Kroese, Shie Mannor, and Reuven Y. Rubinstein. A tuto-

rial on the cross-entropy method. *Annals of Operations Research*, 134(1):19–67, 2005. ISSN 1572-9338. doi: 10.1007/s10479-005-5724-z. URL http://dx.doi.org/10.1007/s10479-005-5724-z. 9.3

[33] Rogério de Lemos, Holger Giese, Hausi A. Müller, Mary Shaw, Jesper Andersson, Marin Litoiu, Bradley Schmerl, Gabriel Tamura, Norha M. Villegas, Thomas Vogel, Danny Weyns, Luciano Baresi, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Ron Demarais, Schahram Dustdar, Gregor Engels, Kurt Geihs, Karl M. Göschka, Alessandra Gorla, Vincenzo Grassi, Paola Inverardi, Gabor Karsai, Jeff Kramer, Antonia Lopes, Jeff Magee, Sam Malek, Serge Mankovskii, Raffaela Mirandola, John Mylopoulos, Oscar Nierstrasz, Mauro Pezzè, Christian Prehofer, Wilhelm Schäfer, Rick Schlichting, Dennis B. Smith, João Pedro Sousa, Ladan Tahvildari, Kenny Wong, and Jochen Wuttke. Software engineering for self-adaptive systems: A second research roadmap. *Software Engineering for . . .*, pages 1–32, 2013. URL http://link.springer.com/chapter/10.1007/978-3-642-35813-5_1. 1

[34] Diego Didona, Francesco Quaglia, Paolo Romano, and Ennio Torre. Enhancing performance prediction robustness by combining analytical modeling and machine learning. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering - ICPE '15*, pages 145–156, New York, New York, USA, 2015. ACM Press. ISBN 9781450332484. doi: 10.1145/2668930.2688047. URL http://dl.acm.org/citation.cfm?doid=2668930.2688047. 9.3, 9.4

[35] Peter A. Dinda. Design, implementation, and performance of an extensible toolkit for resource prediction in distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 17(2):160–173, February 2006. ISSN 1045-9219. doi: 10.1109/TPDS.2006.24. URL http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=1566594. 3.5

[36] Simon Dobson, Franco Zambonelli, Spyros Denazis, Antonio Fernández, Dominique Gaïti, Erol Gelenbe, Fabio Massacci, Paddy Nixon, Fabrice Saffre, and Nikita Schmidt. A survey of autonomic communications. *ACM Transactions on Autonomous and Adaptive Systems*, 1(2):223–259, December 2006. ISSN 15564665. doi: 10.1145/1186778.1186782. URL http://portal.acm.org/citation.cfm?doid=1186778.1186782. 1

[37] Simon Dobson, Roy Sterritt, Paddy Nixon, and Mike Hinchey. Fulfilling the vision of autonomic computing. *Computer*, 43(1):35–41, January 2010. ISSN 0018-9162. doi: 10.1109/MC.2010.14. URL http://ieeexplore.ieee.org/document/5398781/. 1

[38] Anca D. Dragan, Kenton C.T. Lee, and Siddhartha S. Srinivasa. Legibility and predictability of robot motion. In *2013 8th ACM/IEEE International Conference on Human-Robot Interaction (HRI)*, pages 301–308. IEEE, March 2013. ISBN 978-1-4673-3101-2. doi: 10.1109/HRI.2013.6483603. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6483603. 7

[39] Subhasri Duttagupta, Rupinder Virk, and Manoj Nambiar. Predicting performance in the presence of software and hardware resource bottlenecks. In *International Symposium on*

Performance Evaluation of Computer and Telecommunication Systems (SPECTS 2014), pages 542–549. IEEE, July 2014. ISBN 978-1-4799-5745-3. doi: 10.1109/SPECTS.2014.6879991. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6879991. 1.1, 8.1.1

[40] Naeem Esfahani and Sam Malek. Uncertainty in self-adaptive software systems. In Rogério de Lemos, Holger Giese, Hausi A. Müller, and Mary Shaw, editors, *Software Engineering for Self-Adaptive Systems II*, volume 7475 of *Lecture Notes in Computer Science*, pages 214–238. Springer Berlin Heidelberg, 2013. URL http://link.springer.com/chapter/10.1007/978-3-642-35813-5_9. 3.5

[41] Naeem Esfahani, Ahmed Elkhodary, and Sam Malek. A learning-based framework for engineering feature-oriented self-adaptive software systems. *IEEE Transactions on Software Engineering*, 39(11):1467–1493, 2013. 9.3, 9.4

[42] Peter Feiler, Kevin Sullivan, Kurt Wallnau, Richard Gabriel, John Goodenough, Richard Linger, Thomas Longstaff, Rick Kazman, Mark Klein, Linda Northrop, and Douglas Schmidt. *Ultra-Large-Scale Systems: The Software Challenge of the Future*. Software Engineering Institute, Carnegie Mellon University, 2006. 1

[43] Antonio Filieri, Henry Hoffmann, and Martina Maggio. Automated design of self-adaptive software with control-theoretical formal guarantees. In *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014*, pages 299–310, Hyderabad, India, 2014. ACM Press. ISBN 9781450327565. doi: 10.1145/2568225.2568272. URL http://dl.acm.org/citation.cfm?doid=2568225.2568272. 9.4

[44] Antonio Filieri, Martina Maggio, Konstantinos Angelopoulos, Nicolás D'Ippolito, Ilias Gerostathopoulos, Andreas Berndt Hempel, Henry Hoffmann, Pooyan Jamshidi, Evangelia Kalyvianaki, Cristian Klein, Filip Krikava, Sasa Misailovic, Alessandro V. Papadopoulos, Suprio Ray, Amir M. Sharifloo, Stepan Shevtsov, Mateusz Ujma, and Thomas Vogel. Software engineering meets control theory. In *Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 71–82, Florence, Italy, 2015. IEEE Press. 9.4

[45] M. Flint, E. Fernandez-Gaucherand, and M. Polycarpou. Cooperative control for UAV's searching risky environments for targets. In *42nd IEEE International Conference on Decision and Control (IEEE Cat. No.03CH37475)*, volume 4, pages 3567–3572. IEEE, 2003. ISBN 0-7803-7924-1. doi: 10.1109/CDC.2003.1271701. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1271701. 8.2.3, 9.3

[46] Vojtěch Forejt, Marta Kwiatkowska, Gethin Norman, and David Parker. Automated verification techniques for probabilistic systems. In M Bernardo and V Issarny, editors, *Formal Methods for Eternal Networked Software Systems (SFM'11)*, volume 6659 of *LNCS*, pages 53–113. Springer, 2011. 4.1

[47] Alessio Gambi, Daniel Moldovan, Georgiana Copil, Hong-Linh Truong, and Schahram Dustdar. On estimating actuation delays in elastic computing systems. In *2013 8th International Symposium on Software Engineering for Adaptive and Self-Managing Sys-*

tems (SEAMS), pages 33–42. IEEE, May 2013. ISBN 978-1-4673-4401-2. doi: 10.1109/SEAMS.2013.6595490. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6595490. 1, 1.2, 2.3

[48] Nadia Gamez, Lidia Fuentes, and Miguel A. Aragüez. Autonomic computing driven by feature models and architecture in FamiWare. In *5th European Conference on Software Architecture*, pages 164–179, Essen, Germany, September 2011. Springer-Verlag. ISBN 978-3-642-23797-3. URL http://dl.acm.org/citation.cfm?id=2041790.2041811. 2.3

[49] Anshul Gandhi, Mor Harchol-Balter, Ram Raghunathan, and Michael A. Kozuch. Autoscale: Dynamic, robust capacity management for multi-tier data centers. *ACM Transactions on Computer Systems*, 30(4), 2012. URL http://dl.acm.org/citation.cfm?id=2382556. 2.3

[50] Anshul Gandhi, Parijat Dube, Alexei Karve, Andrzej Kochut, and Li Zhang. Modeling the impact of workload on cloud resource scaling. In *2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing*, pages 310–317. IEEE, October 2014. ISBN 978-1-4799-6905-0. doi: 10.1109/SBAC-PAD.2014.16. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6970679. 1.1, 8.1.1, 8.1.1

[51] Anshul Gandhi, Parijat Dube, Alexei Karve, Andrzej Kochut, and Li Zhang. Adaptive, model-driven autoscaling for cloud applications. In *11th International Conference on Autonomic Computing*, pages 57–64, 2014. ISBN 978-1-931971-11-9. URL https://www.usenix.org/system/files/conference/icac14/icac14-paper-gandhi.pdf. 9.3

[52] David Garlan, Robert T. Monroe, and David Wile. Acme: Architectural description of component-based systems. In Gary T Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, pages 47–68. Cambridge University Press, 2000. 2.1, 7.1

[53] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley Schmerl, and Peter Steenkiste. Rainbow: architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54, October 2004. ISSN 0018-9162. doi: 10.1109/MC.2004.175. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1350726. 2.1, 7.1, 8.1

[54] David Garlan, Bradley Schmerl, and Shang-Wen Cheng. Software architecture-based self-adaptation. In Yan Zhang, Laurence Tianruo Yang, and Mieso K. Denko, editors, *Autonomic Computing and Networking*, pages 31–55. Springer US, 2009. URL http://link.springer.com/chapter/10.1007/978-0-387-89828-5_2. (document), 1, 2.1, 3.1, 7.1

[55] Jorge L. Garriga and Masoud Soroush. Model predictive control tuning methods: A review. *Industrial & Engineering Chemistry Research*, 49(8):3505–3515, April 2010. ISSN 0888-5885. doi: 10.1021/ie900323c. URL http://pubs.acs.org/doi/abs/10.1021/ie900323c. 9.3

[56] Simos Gerasimou, Radu Calinescu, and Alec Banks. Efficient runtime quantitative ver-

ification using caching, lookahead, and nearly-optimal reconfiguration. In *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems - SEAMS 2014*, pages 115–124, New York, New York, USA, June 2014. ACM. ISBN 9781450328647. doi: 10.1145/2593929.2593932. URL http://dl.acm.org/citation.cfm?id=2593929.2593932. 2.6, 9.3

[57] Sergio Giro. Optimal schedulers vs optimal bases: An approach for efficient exact solving of Markov decision processes. *Theoretical Computer Science*, 538:70–83, 2014. ISSN 03043975. doi: 10.1016/j.tcs.2013.08.020. 1.3

[58] D. Gmach, J. Rolia, L. Cherkasova, and A. Kemper. Resource pool management: Reactive versus proactive or let's be friends. *Computer Networks*, 2009. URL http://www.sciencedirect.com/science/article/pii/S1389128609002655. 2.2, 9.4

[59] Xianping Guo and Onésimo Hernández-Lerma. *Continuous-Time Markov Decision Processes*, pages 9–18. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. ISBN 978-3-642-02547-1. doi: 10.1007/978-3-642-02547-1_2. URL http://dx.doi.org/10.1007/978-3-642-02547-1_2. 2

[60] Vijay Gupta and Ignacio E. Grossmann. Solution strategies for multistage stochastic programming with endogenous uncertainties. *Computers & Chemical Engineering*, 35(11): 2235–2247, 2011. ISSN 00981354. doi: 10.1016/j.compchemeng.2010.11.013. 9.3

[61] Jason J. Haas, J. D. Doak, and Jason R. Hamlet. Machine-oriented biometrics and cocooning for dynamic network defense. In *Proceedings of the Eighth Annual Cyber Security and Information Intelligence Research Workshop on - CSIIRW '13*, page 1, New York, New York, USA, 2013. ACM Press. ISBN 9781450316873. doi: 10.1145/2459976.2460014. URL http://dl.acm.org/citation.cfm?doid=2459976.2460014. 9.4

[62] Marcus Handte, Gregor Schiele, Verena Matjuntke, Christian Becker, and Pedro José Marrón. 3PC: System support for adaptive peer-to-peer pervasive computing. *ACM Transactions on Autonomous and Adaptive Systems*, 7(1):1–19, April 2012. ISSN 15564665. doi: 10.1145/2168260.2168270. URL http://dl.acm.org/citation.cfm?id=2168260.2168270. 2.2

[63] HAProxy. The reliable, high performance TCP/HTTP load balancer. http://www.haproxy.org/, 2016. 8.1.1

[64] Julia Hielscher, Raman Kazhamiakin, Andreas Metzger, and Marco Pistore. A framework for proactive self-adaptation of service-based applications based on online testing. In *1st European Conference on Towards a Service-Based Internet*, volume 5377, pages 122–133. Springer Berlin Heidelberg, 2008. URL http://link.springer.com/chapter/10.1007/978-3-540-89897-9_11. 2.2

[65] 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*, pages 1–4, New York, New York, USA, September 2015. ACM Press. ISBN 9781450333931. doi: 10.1145/2797433.2797439. URL http://dl.acm.org/citation.cfm?id=2797433.2797439. 8.1, 8.1.2, 8.2.2

[66] C A R Hoare. Programs are predicates. In C A R Hoare and J C Shepherdson, editors, *Mathematical Logic and Programming Languages*, pages 141–154. Prentice-Hall, 1985. 5.2.1

[67] F. Hooshmand Khaligh and S.A. MirHassani. A mathematical model for vehicle routing problem under endogenous uncertainty. *International Journal of Production Research*, 54(2):579–590, January 2016. ISSN 0020-7543. doi: 10.1080/00207543.2015.1057625. URL http://www.tandfonline.com/doi/full/10.1080/00207543.2015.1057625. 9.3

[68] Nikolaus Huber, André van Hoorn, Anne Koziolek, Fabian Brosig, and Samuel Kounev. Modeling run-time adaptation at the system architecture level in dynamic service-oriented environments. *Service Oriented Computing and Applications*, 8(1):73–89, September 2013. ISSN 1863-2386. doi: 10.1007/s11761-013-0144-4. URL http://link.springer.com/10.1007/s11761-013-0144-4. 1

[69] Markus C. Huebscher and Julie A. McCann. A survey of autonomic computing–degrees, models, and applications. *ACM Computing Surveys*, 40(3), August 2008. ISSN 03600300. doi: 10.1145/1380584.1380585. URL http://dl.acm.org/citation.cfm?id=1380584.1380585. 1.2

[70] R. J. Hyndman and G. Athanasopoulos. *Forecasting: principles and practice*. OTexts, 2014. URL https://www.otexts.org/fpp. 8.1.1

[71] Stefano Iannucci and Sherif Abdelwahed. A probabilistic approach to autonomic security management. In *2016 IEEE International Conference on Autonomic Computing (ICAC)*, pages 157–166. IEEE, July 2016. ISBN 978-1-5090-1654-9. doi: 10.1109/ICAC.2016.12. URL http://ieeexplore.ieee.org/document/7573127/. 2.3, 2.5

[72] Anne Immonen and Eila Niemelä. Survey of reliability and availability prediction methods from the viewpoint of software architecture. *Software and Systems Modeling*, 7(1):49–65, 2008. ISSN 16191366. doi: 10.1007/s10270-006-0040-x. 2.1

[73] Emilio Incerto, Mirco Tribastone, and Catia Trubiani. A proactive approach for runtime self-adaptation based on queueing network fluid analysis. In *Proceedings of the 1st International Workshop on Quality-Aware DevOps - QUDOS 2015*, pages 19–24, New York, New York, USA, September 2015. ACM Press. ISBN 9781450338172. doi: 10.1145/2804371.2804375. URL http://dl.acm.org/citation.cfm?id=2804371.2804375. 9.3

[74] Mohammad Islam, Shaolei Ren, Hasan Mahmud, and Gang Quan. Online energy budgeting for cost minimization in virtualized data center. *IEEE Transactions on Services Computing*, PP(99):1–1, 2015. ISSN 1939-1374. doi: 10.1109/TSC.2015.2390231. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7006709. 1.1, 8.1.1

[75] Serena Ivaldi, Olivier Sigaud, Bastien Berret, and Francesco Nori. From humans to humanoids: the optimal control framework. *Paladyn, Journal of Behavioral Robotics*, 3(2):75–91, January 2012. ISSN 2081-4836. doi: 10.2478/s13230-012-0022-3. 1.3

[76] Daniel Jackson. *Software Abstractions: logic, language, and analysis*. The MIT Press,

2012. 5, 5.2

[77] Gueyoung Jung, Kaustubh R. Joshi, Matti A. Hiltunen, Richard D. Schlichting, and Calton Pu. A cost-sensitive adaptation engine for server consolidation of multitier applications. In Jean Bacon and Brian F. Cooper, editors, *Middleware 2009, ACM/IFIP/USENIX, 10th International Middleware Conference*, pages 163—-183, Urbana, IL, 2009. Springer. URL http://link.springer.com/chapter/10.1007/978-3-642-10445-9_9. 1.1, 8.1.1, 9.3

[78] Marcin Jurdziński, François Laroussinie, and Jeremy Sproston. *Model Checking Probabilistic Timed Automata with One or Two Clocks*, pages 170–184. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. ISBN 978-3-540-71209-1. doi: 10.1007/978-3-540-71209-1_15. URL http://dx.doi.org/10.1007/978-3-540-71209-1_15. 2

[79] Leslie Pack Kaelbling, Michael L. Littman, and Anthony R. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101(1):99–134, 1998. ISSN 00043702. doi: 10.1016/S0004-3702(98)00023-X. 9.4

[80] Donald L. Keefer. Certainty equivalents for three-point discrete-distribution approximations. *Management Science*, 40(6):760–773, 1994. 3.5, 8.1.2

[81] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1160055. 1, 2.1, 3.1

[82] Min Hyuk Kim, Hyeoncheol Baik, and Seokcheon Lee. Response threshold model based UAV search planning and task allocation. *Journal of Intelligent and Robotic Systems: Theory and Applications*, 75(3-4):625–640, September 2014. ISSN 15730409. doi: 10.1007/s10846-013-9887-6. URL http://link.springer.com/10.1007/s10846-013-9887-6. 8.2.3

[83] Cristian Klein, Martina Maggio, Karl-Erik Årzén, and Francisco Hernández-Rodriguez. Brownout: building more robust cloud applications. In *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014*, pages 700–711, New York, New York, USA, May 2014. ACM. ISBN 9781450327565. doi: 10.1145/2568225.2568227. URL http://dl.acm.org/citation.cfm?id=2568225.2568227. (document), 5, 8.1.1, 9.4

[84] Leonard. Kleinrock and Leonard. *Queueing systems*. Wiley Interscience, 1975. ISBN 0471491101. 4.3, 5.1

[85] Samuel Kounev and Christofer Dutz. QPME: A performance modeling tool based on queueing Petri nets. *ACM SIGMETRICS Performance Evaluation Review*, 36(4):46–51, March 2009. ISSN 01635999. doi: 10.1145/1530873.1530883. URL http://portal.acm.org/citation.cfm?doid=1530873.1530883. 5.1

[86] Heiko Koziolek. Performance evaluation of component-based software systems: A survey. *Performance Evaluation*, 67(8):634–658, 2010. ISSN 0166-5316. doi: http://dx.doi.org/10.1016/j.peva.2009.07.007. 2.1

[87] Jeff Kramer and Jeff Magee. Self-managed systems: an architectural challenge. In *2007 Future of Software Engineering (FOSE '07)*, pages 259–268. IEEE Computer Society, 2007. ISBN 0769528295. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4221625. 1, 2.1

[88] 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, Part B:184–206, October 2014. ISSN 15741192. doi: 10.1016/j.pmcj.2014.09.009. URL http://www.sciencedirect.com/science/article/pii/S157411921400162X. 1, 2.1, 2.2, 3

[89] D. Kumar, A. Tantawi, and L. Zhang. Estimating model parameters of adaptive software systems in real-time. *Run-time Models for Self-managing ...*, 2010. URL http://link.springer.com/chapter/10.1007/978-3-0346-0433-8_3. 8.1.1

[90] V. Kumar, B.F. Cooper, and K. Schwan. Distributed stream management using utility-driven self-adaptive middleware. In *Second International Conference on Autonomic Computing (ICAC'05)*, pages 3–14. IEEE, 2005. ISBN 0-7965-2276-9. doi: 10.1109/ICAC.2005.24. URL http://ieeexplore.ieee.org/document/1498048/. 1

[91] Dara Kusic, Jeffrey O. Kephart, James E. Hanson, Nagarajan Kandasamy, and Guofei Jiang. Power and performance management of virtualized computing environments via lookahead control. *Cluster Computing*, 12(1):1–15, October 2008. ISSN 1386-7857. doi: 10.1007/s10586-008-0070-y. URL http://link.springer.com/10.1007/s10586-008-0070-y. 2.4

[92] Marta Kwiatkowska and David Parker. Automated verification and strategy synthesis for probabilistic systems. In *Proceedings of the 11th International Symposium on Automated Technology for Verification and Analysis (ATVA'13)*, pages 5–22. Springer, 2013. URL http://link.springer.com/chapter/10.1007/978-3-319-02444-8_2. (document), 1.3, 3.3

[93] Marta Kwiatkowska and David Parker. Automated verification and strategy synthesis for probabilistic systems. In *11th International Symposium on Automated Technology for Verification and Analysis (ATVA)*, pages 5–22, Hanoi, Vietnam, 2013. Springer International Publishing. doi: 10.1007/978-3-319-02444-8_2. URL http://link.springer.com/10.1007/978-3-319-02444-8_2. 4.1

[94] Marta Kwiatkowska, Gethin Norman, and David Parker. Verifying randomized distributed algorithms with PRISM. In *Proc. Workshop on Advances in Verification (Wave'2000)*, July 2000. 4.1

[95] Marta Kwiatkowska, Gethin Norman, and David Parker. Probabilistic symbolic model checking with PRISM: A hybrid approach. In *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '02)*, pages 52–66. Springer-Verlag, 2002. URL http://link.springer.com/chapter/10.1007/3-540-46002-0_5. 1.3, 4

[96] Marta Kwiatkowska, Gethin Norman, and David Parker. PRISM 4.0: verification of probabilistic real-time systems. In *23rd international conference on Computer Aided Veri-*

fication, pages 585–591. Springer-Verlag, July 2011. ISBN 978-3-642-22109-5. URL http://dl.acm.org/citation.cfm?id=2032305.2032352. 2.6, 4.1

[97] P. Lalanda, J. A. McCann, and A. Diaconescu. *Autonomic Computing*. Springer-Verlag London, 2013. URL http://link.springer.com/content/pdf/10.1007/978-1-4471-5007-7.pdf. 1

[98] A. Lew and H. Mauch. *Dynamic Programming: A Computational Tool*. Studies in Computational Intelligence. Springer Berlin Heidelberg, 2007. ISBN 9783540370147. 5.1

[99] Jie Liu, Bodhi Priyantha, Ted Hart, Heitor S. Ramos, Antonio A. F. Loureiro, and Qiang Wang. Energy efficient GPS sensing with cloud offloading. In *Proceedings of the 10th ACM Conference on Embedded Network Sensor Systems - SenSys '12*, page 85, New York, New York, USA, November 2012. ACM. ISBN 9781450311694. doi: 10.1145/2426656.2426666. URL http://dl.acm.org/citation.cfm?id=2426656.2426666. 1

[100] LQNS. Layered Queueing Network Solver. http://www.sce.carleton.ca/rads/lqns, 2011. 5.1, 9.2

[101] Jian Lü, Yu Huang, Chang Xu, and Xiaoxing Ma. Managing environment and adaptation risks for the internetware paradigm. In Zhiming Liu, Jim Woodcock, and Huibiao Zhu, editors, *Theories of Programming and Formal Methods: Essays Dedicated to Jifeng He on the Occasion of His 70th Birthday*, pages 271–284. Springer, Berlin, Heidelberg, 2013. doi: 10.1007/978-3-642-39698-4_17. URL http://link.springer.com/10.1007/978-3-642-39698-4_17. 1

[102] Lasse Maatta, Jukka Suhonen, Teemu Laukkarinen, Timo D. Hamalainen, and Marko Hannikainen. Program image dissemination protocol for low-energy multihop wireless sensor networks. In *2010 International Symposium on System on Chip*, pages 133–138. IEEE, September 2010. ISBN 978-1-4244-8279-5. doi: 10.1109/ISSOC.2010.5625550. URL http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=5625550. 1

[103] Sam Malek, George Edwards, Yuriy Brun, Hossein Tajalli, Joshua Garcia, Ivo Krka, Nenad Medvidovic, Marija Mikic-Rakic, and Gaurav Sukhatme. An architecture-driven software mobility framework. *Journal of Systems and Software*, 83(6), 2010. URL http://www.sciencedirect.com/science/article/pii/S0164121209002842. 9.3

[104] Ming Mao and Marty Humphrey. A performance study on the VM startup time in the cloud. In *2012 IEEE Fifth International Conference on Cloud Computing*, pages 423–430. IEEE, June 2012. ISBN 978-1-4673-2892-0. doi: 10.1109/CLOUD.2012.103. URL http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=6253534. 3

[105] Julie A. Mccann and Markus C. Huebscher. Evaluation issues in autonomic computing. In Hai Jin, Yi Pan, Nong Xiao, and Jianhua Sun, editors, *Grid and Cooperative Computing*, volume 3252 of *Lecture Notes in Computer Science*, pages 597–608, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. ISBN 978-3-540-23578-1. doi: 10.1007/b100775. 2.3

[106] Andreas Metzger, Osama Sammodi, and Klaus Pohl. Accurate proactive adaptation of service-oriented systems. In Javier Cámara, Rogério de Lemos, Carlo Ghezzi, and Antónia Lopes, editors, *Assurances for Self-Adaptive Systems*, volume 7740, pages 240–265. Springer Berlin Heidelberg, 2013. URL http://link.springer.com/chapter/10.1007/978-3-642-36249-1_9. 2.2

[107] Stefan Mitsch and André Platzer. *ModelPlex: Verified Runtime Validation of Verified Cyber-Physical System Models*, pages 199–214. Springer International Publishing, Toronto, ON, Canada, 2014. ISBN 978-3-319-11164-3. doi: 10.1007/978-3-319-11164-3_17. URL http://dx.doi.org/10.1007/978-3-319-11164-3_17. 9.4

[108] 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 - ESEC/FSE 2015*, pages 1–12, New York, New York, USA, August 2015. ACM Press. ISBN 9781450336758. doi: 10.1145/2786805.2786853. URL http://dl.acm.org/citation.cfm?id=2786805.2786853. 1.5, 1

[109] 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, Wuerzburg, Germany, July 2016. IEEE. ISBN 978-1-5090-1654-9. doi: 10.1109/ICAC.2016.59. URL http://ieeexplore.ieee.org/document/7573126/. 1.5, 1, 7, 8

[110] David J. Musliner. Imposing real-time constraints on self-adaptive controller synthesis. *Self-Adaptive Software*, 1936:143–160, 2001. URL http://link.springer.com/chapter/10.1007/3-540-44584-6_12. 2.3, 9.4

[111] Athanasios Naskos, Emmanouela Stachtiari, Anastasios Gounaris, Panagiotis Katsaros, Dimitrios Tsoumakos, Ioannis Konstantinou, and Spyros Sioutas. Dependable horizontal scaling based on probabilistic model checking. In *15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 31–40. IEEE, May 2015. ISBN 978-1-4799-8006-2. doi: 10.1109/CCGrid.2015.91. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7152469. 2.5, 9.3, 9.3

[112] OPERA. Optimization, Performance Evaluation and Resource Allocator. http://www.ceraslabs.com/technologies/opera, n.d. 5.1, 8.1.1, 9.2

[113] Ivan Dario Paez Anaya, Viliam Simko, Johann Bourcier, Noël Plouzeau, and Jean-Marc Jézéquel. A prediction-driven adaptation approach for self-adaptive sensor networks. In *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 145–154. ACM, 2014. ISBN 9781450328647. URL http://hal.archives-ouvertes.fr/hal-00983046/. 1, 9.3

[114] 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, September 2016. ISBN 978-1-5090-3534-2. doi: 10.1109/SASO.2016.19. URL http:

//ieeexplore.ieee.org/document/7774394/. 9.4

[115] Manish Parashar and Salim Hariri. Autonomic computing: An overview. In Jean-Pierre Banâtre, Pascal Fradet, Jean-Louis Giavitto, and Olivier Michel, editors, *Unconventional Programming Paradigms*, Lecture Notes in Computer Science, pages 257–269. Springer Berlin Heidelberg, 2005. URL http://link.springer.com/chapter/10.1007/11527800_20. 2.2

[116] Tharindu Patikirikorala, Alan Colman, Jun Han, and Liuping Wang. A systematic survey on the design of self-adaptive software systems using control engineering approaches. In *2012 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 33–42. IEEE, June 2012. ISBN 978-1-4673-1787-0. doi: 10.1109/SEAMS.2012.6224389. URL http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=6224389. 1.2

[117] Ryan R. Pitre, X. Rong Li, and R. Delbalzo. UAV route planning for joint search and track missions—an information-value approach. *IEEE Transactions on Aerospace and Electronic Systems*, 48(3):2551–2565, July 2012. ISSN 0018-9251. doi: 10.1109/TAES.2012.6237608. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6237608. 9.3

[118] Vahe Poladian, David Garlan, Mary Shaw, M. Satyanarayanan, Bradley Schmerl, and João Pedro Sousa. Leveraging resource prediction for anticipatory dynamic configuration. In *Self-Adaptive and Self-Organizing Systems*, pages 214—-223. IEEE, July 2007. ISBN 0-7695-2906-2. doi: 10.1109/SASO.2007.35. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4274905http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4274905. 2.2, 3.2, 3.5

[119] Warren B. Powell. Perspectives of approximate dynamic programming. *Annals of Operations Research*, February 2012. ISSN 0254-5330. doi: 10.1007/s10479-012-1077-6. URL http://link.springer.com/10.1007/s10479-012-1077-6. 9.3

[120] M. L. Puterman. Dynamic programming. *Encyclopedia of Physical Science and Technology*, 4:673–696, 2002. URL http://puterman.chcm.ubc.ca/bams517_518_08/dynamicprogramming.pdf. 1.3, 5.1.1

[121] M. L. Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, Ltd, 2014. (document), 1.3, 3.3

[122] Kashifuddin Qazi, Yang Li, and Andrew Sohn. Workload prediction of virtual machines for harnessing data center resources. In *2014 IEEE 7th International Conference on Cloud Computing*, pages 522–529. IEEE, June 2014. ISBN 978-1-4799-5063-8. doi: 10.1109/CLOUD.2014.76. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6973782. 1.1, 8.1.1

[123] Rahul Raheja, Shang-Wen Cheng, David Garlan, and Bradley Schmerl. Improving architecture-based self-adaptation using preemption. In *Self-Organizing Architectures*, pages 21–37. Springer-Verlag, September 2010. ISBN 3-642-14411-X, 978-3-642-14411-0. URL http://dl.acm.org/citation.cfm?id=1880569.1880572. 3.4

[124] J.B. Rawlings. Tutorial overview of model predictive control. *IEEE Control Systems Magazine*, 20(3):38–52, June 2000. ISSN 02721708. doi: 10.1109/37.845037. URL http://ieeexplore.ieee.org/document/845037/. 2.4

[125] Magnus J.E. Richardson and Tamar Flash. On the emulation of natural movements by humanoid robots. In *IEEE-RAS International Conference on Humanoids Robots*, Cambridge, Massachusetts, USA, 2000. 1.3

[126] RUBiS. RUBiS: Rice University Bidding System. http://rubis.ow2.org/, 2009. 1.1

[127] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Transactions on Autonomous and Adaptive Systems*, 4(2):1–42, May 2009. ISSN 15564665. doi: 10.1145/1516533.1516538. URL http://portal.acm.org/citation.cfm?doid=1516533.1516538. 1, 1.2, 2.1, 2.2, 1

[128] Kristin E. Schaefer, Deborah R. Billings, James L. Szalma, Jeffrey K. Adams, Tracy L. Sanders, Jessie Y. Chen, and Peter A. Hancock. A meta-analysis of factors influencing the development of trust in automation: Implications for human-robot interaction. Technical report, Army Research Lab, Aberdeen Proving Ground, 2014. 7

[129] Bradley Schmerl, Javier Cámara, Jeffrey Gennari, David Garlan, Paulo Casanova, Gabriel A. Moreno, Thomas J. Glazier, and Jeffrey M. Barnes. Architecture-based self-protection: Composing and reasoning about denial-of-service mitigations. In *HotSoS 2014: 2014 Symposium and Bootcamp on the Science of Security*, Raleigh, NC, USA, 2014. 1, 1.2, 7, 7.2, 7.2.1

[130] D. Seto, B. Krogh, L. Sha, and A. Chutinan. The simplex architecture for safe online control system upgrades. In *Proceedings of the 1998 American Control Conference. ACC (IEEE Cat. No.98CH36207)*, pages 3504–3508 vol.6. IEEE, 1998. ISBN 0-7803-4530-4. doi: 10.1109/ACC.1998.703255. URL http://ieeexplore.ieee.org/document/703255/. 9.4

[131] Jens Steiner, Ursula Goltz, and Jochen Maaß. *Self-management within a Software Architecture for Parallel Kinematic Machines*, pages 355–371. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. ISBN 978-3-642-16785-0. doi: 10.1007/978-3-642-16785-0_20. URL http://dx.doi.org/10.1007/978-3-642-16785-0_20. 9.4

[132] Christian Stier, Anne Koziolek, Henning Groenda, and Ralf Reussner. *Model-Based Energy Efficiency Analysis of Software Architectures*, pages 221–238. Springer International Publishing, Cham, 2015. ISBN 978-3-319-23727-5. doi: 10.1007/978-3-319-23727-5_18. 2.1

[133] Ryohei Suzuki, Fukiko Kawai, Chikashi Nakazawa, Tetsuro Matsui, and Eitaro Aiyoshi. Parameter optimization of model predictive control by PSO. *Electrical Engineering in Japan*, 178(1):40–49, January 2012. ISSN 04247760. doi: 10.1002/eej.21188. URL http://doi.wiley.com/10.1002/eej.21188. 9.3

[134] SWIG. Simplified Wrapper and Interface Generator. http://www.swig.org/, 2017. 8.2.2

[135] Andrew Symington, Sonia Waharte, Simon Julier, and Niki Trigoni. Probabilistic target detection by camera-equipped UAVs. In *2010 IEEE International Conference on Robotics and Automation*, pages 4076–4081. IEEE, May 2010. ISBN 978-1-4244-5038-1. doi: 10.1109/ROBOT.2010.5509355. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5509355. 8.1.2

[136] Genci Tallabaci and Vítor E. Silva Souza. Engineering adaptation with Zanshin : an experience report. In *Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 93–102. IEEE Press, May 2013. ISBN 978-1-4673-4401-2. URL http://dl.acm.org/citation.cfm?id=2663546.2663563. 1

[137] B. Trushkowsky, P. Bodík, A. Fox, and M. J. Franklin. The SCADS director: Scaling a distributed storage system under stringent performance requirements. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies (FAST'11)*, pages 163—-176, San Jose, California, 2011. USENIX Association. URL http://static.usenix.org/legacy/events/fast11/tech/full_papers/Trushkowsky.pdf. 2.4

[138] Sebastian Vansyckel, Dominik Schafer, Gregor Schiele, and Christian Becker. Configuration management for proactive adaptation in pervasive environments. In *2013 IEEE 7th International Conference on Self-Adaptive and Self-Organizing Systems*, pages 131–140. IEEE, September 2013. ISBN 978-0-7695-5129-6. doi: 10.1109/SASO.2013.28. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6676500. 1

[139] András Varga and Rudolf Hornig. An overview of the OMNeT++ simulation environment. In *Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems Workshop*, Marseille, France, 2008. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering). ISBN 978-963-9799-20-2. URL http://dl.acm.org/citation.cfm?id=1416222.1416290. 8.1.1

[140] Michael J. Veth. Advanced formation flight control. Technical report, Air Force Institute of Technology, 1994. 8.1.2

[141] Norha M. Villegas, Gabriel Tamura, and Hausi A. Müller. Dynamico: A reference model for governing control objectives and context relevance in self-adaptive software systems. *Software Engineering for . . .* , 2013. URL http://link.springer.com/chapter/10.1007/978-3-642-35813-5_11. 1

[142] William E. Walsh, Gerald Tesauro, Jeffrey O. Kephart, and Rajarshi Das. Utility functions in autonomic systems. In *International Conference on Autonomic Computing, 2004. Proceedings.*, pages 70–77. IEEE, 2004. ISBN 0-7695-2114-2. doi: 10.1109/ICAC.2004.1301349. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1301349. 1

[143] Chen Wang and Jean-Louis Pazat. A two-phase online prediction approach for accurate and timely adaptation decision. In *2012 IEEE Ninth International Conference*

on Services Computing, pages 218–225. IEEE, June 2012. ISBN 978-1-4673-3049-7. doi: 10.1109/SCC.2012.26. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6274147. 2.2

[144] Ji Zhang, Zhenxiao Yang, Betty H. C. Cheng, and Philip K. McKinley. Adding safeness to dynamic adaptation techniques. In *Proceedings of the ICSE 2004 Workshop on Architecting Dependable Systems*, Edinburgh, Scotland, 2004. 2.3

[145] Jiheng Zhang and Bert Zwart. Steady state approximations of limited processor sharing queues in heavy traffic. *Queueing Systems*, 60(3-4):227–246, November 2008. ISSN 0257-0130. doi: 10.1007/s11134-008-9095-4. URL http://dl.acm.org/citation.cfm?id=1484953.1484959. 8.1.1

[146] Tao Zheng, Murray Woodside, and Marin Litoiu. Performance model estimation and tracking using optimal filters. *IEEE Transactions on Software Engineering*, 34(3):391–406, May 2008. ISSN 0098-5589. doi: 10.1109/TSE.2008.30. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4515874. 8.1.1, 9.3

[147] Parisa Zoghi, Mark Shtern, and Marin Litoiu. Designing search based adaptive systems: a quantitative approach. In *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems - SEAMS 2014*, pages 7–16, New York, New York, USA, 2014. ACM Press. ISBN 9781450328647. doi: 10.1145/2593929.2593935. URL http://dl.acm.org/citation.cfm?doid=2593929.2593935. 9.3, 9.3