Combining Static Policies and Machine Learning for Dynamic Adaptation

Francisco Duarte
francisco.c.duarte@tecnico.ulisboa.pt

Instituto Superior Técnico
(Advisor: Professor Luís Rodrigues)

Abstract. Among the approaches that have been proposed to support dynamic adaptation, one can find two distinct techniques that appear to be antagonistic: the use of static models or policies specified by human operators and the use of fully automated techniques based on machine learning. Static policies are able to capture the valuable knowledge of experts into an intelligible model, which may be hard to extract when using completely automated approaches. Furthermore, static policies also provide a higher representation flexibility. However, expert-defined models and policies are unable to deal with unforeseen scenarios and are hard to keep up-to-date as the system evolves. This work aims at combining the advantages of static policies and machine learning tools as complementary techniques to drive the dynamic adaptation of systems. The approach consists in using the expert's knowledge to bootstrap the adaptation process and use machine learning to revise, refine, and update the adaptation policies at run-time.

1 Introduction

As computer systems become more complex, they also become harder to manage by humans. Current systems are composed by many components, each of these with multiple deployment and configuration options: to find the right system configuration that maximizes some high level business goal may be extremely hard. Furthermore, these systems operate in dynamic environments where the workloads are subject to change, faults occur, and components need to be frequently updated. This makes the task of managing a complex system error-prone and time consuming.

In this context, the idea of automating, even if partially, the adaptation process becomes extremely appealing. If successfully implemented, the approach has the potential to offer a faster and better response to events that may negatively affect the behavior of the system. Furthermore, automated adaptation can also contribute to reduce significantly the operational costs associated with system maintenance, by allowing the system to be managed by smaller teams that are assisted by automatic tools.

Autonomic computing was proposed as a solution to this problem [1]. It refers to systems that are able to manage themselves, under varying conditions, in order

to achieve some high-level goals. The MAPE-K control loop was presented as a solution to the self-adaptation problem. It is an external control mechanism responsible for Monitoring the system and its context, Analyzing the data, Plan and Execute changes to the managed system, leveraging Knowledge about it and its context [2].

A common way of representing a system is through the use of architectural models [3] that represent the system as a set of components, connectors (interconnections between components) and their properties. These models provide a global perspective of the system, information about the components and connectors, and they also support the definition of integrity constraints [4]. The architectural model can be used in the Knowledge that the MAPE-K control loop has about the system, given that it provides information about the managed system, its components and constraints that when violated, trigger adaptations.

Self-adaptive systems are subject to uncertainty [5]. Here we will consider the uncertainty associated with the adaptations actions' effects, e.g., activating a server may not have the expected result, as it may suffer from an unexpected problem (disk failure). The information related with this kind of uncertainty may be useful in some cases, e.g., when it is necessary to know the worst possible scenario.

Human involvement is essential in order to build trust in the system and its ability to adapt, and may prove to be useful in certain situations, e.g., reparations. This involvement is greatly facilitated if an intelligible model is available and, in our case, if the uncertainty under consideration is explicitly represented in this model.

Among the approaches that have been proposed to support self-adaptation, one can find two distinct techniques that appear to be antagonistic: the use of static models or policies specified by human operators and the use of fully automated techniques based on machine learning. We will focus mainly on the use of human defined static policies to adapt the system, when considering the first technique. Static policies (rules that select the adaptation action based on the system's current state, and we make the assumption that they provide the action's expected effect) are able to capture the valuable knowledge of experts about the system into an intelligible model of the adaptation logic, these kind of models may be hard to extract when using completely automated approaches. Furthermore, static policies provide a higher representation flexibility than the automated approaches, given that models extracted from automated approaches have a fixed representation, e.g., decision trees, or artificial neural networks. This flexibility can be leveraged to represent, explicitly, the uncertainty mentioned before.

Unfortunately, expert-defined policies are unable to deal with unforeseen scenarios and are hard to keep up-to-date as the system evolves. Fully automated approaches that use on-line learning, however can keep the system updated, based on the system's history, and are robust when dealing with new situations. This means that fully automated techniques can achieve a more accurate prediction of the real behavior of the system. On the other hand, fully automated

approaches that use machine learning to find the correct adaptation strategies, require a large training set of observations, usually collected from long and comprehensive training phases to provide meaningful results.

To the full extent of our knowledge there is no solution that leverages the knowledge an expert has about a system, provides an intelligible model and explicitly represents uncertainty to facilitate the human involvement, deals with unforeseen scenarios and uses the history of the system to update the model, achieving higher accuracy on the prediction of the system's behavior.

This work aims at combining the advantages of static policies and machine learning as complementary techniques to drive the dynamic adaptation of systems. The approach consists in using the expert's knowledge to specify adaptation policies at design-time and use machine learning to refine and update such policies at run-time.

In detail, we plan to leverage languages such as Stitch [6] and the one presented by Cámara et al. (2014) [7], that allow experts to capture their knowledge into a set of possible *strategies* to drive adaptation. *Strategies* are composed by a set of adaptation actions, in the form of condition-action-effect rules. The idea is to bootstrap dynamic adaptation by following the knowledge provided by the experts to build *strategies* and then use data from the running system as feedback to improve them in an automated manner.

The rest of the report is organized as follows. Section 2 briefly summarizes the goals and expected results of our work. Section 3 describes a simple system that will be used throughout the rest of the report as an example, and some terminology. Section 4 presents a description of the problem, a general method of how to solve it and the challenges of such solutions. Section 5 presents all the background related with our work. Section 6 describes the proposed architecture to be implemented and Section 7 describes how we plan to evaluate our results. Finally, Section 8 presents the schedule of future work and Section 9 concludes the report.

2 Goals

We address the problem of supporting automated self-adaptation of a system operating on a dynamic environment. We assume that dynamic adaptation can be performed by applying a set of *adaptations*. Examples of adaptations are the activation or decommission of components, changing the values of configuration parameters of some components, or changing the way components interact. Each individual adaptation has a set of possible *impacts* on the system's behavior, that are functions of the system's state. Given a system's state, one should select the set of adaptations that should be applied to the system to achieve its goals.

In order to make the right choices, it is fundamental that the *impact functions* accurately capture the adaptations' effects. We could aim at designing a system where all this information could be specified by an expert, based on an architectural model, taking advantage of the knowledge he has about the system. This approach provides an intelligible model of the system and allows the ex-

plicit representation of the uncertainty associated with the adaptation's effects. However, it is not easy for an expert to describe the impact of actions on a system accurately and the adaptation model would be static. The other approach would be to learn a model of the system automatically, based on data monitored from it. Usually, it is harder to extract an intelligible model using this approach and uncertainty is not represented explicitly (different effects are combined into a single one). Furthermore, they need to go through a training phase, during which the system can be far from its goals (because the learner would have to try sub-optimal configurations in order to learn about the system). Therefore, with this project we aim to:

> *Goals:* Design and implement a set of mechanisms that, starting from a set of expert-defined adaptation policies, can automatically use data collected on-line to refine the *impact functions* associated to such *adaptations*, keeping the same structure of the policies to maintain its representation and ease of interpretation, while achieving a better accuracy.

We believe that this goal can be achieved by the following approach. A number of condition-action-effect rules (adaptation policies) is collected from experts. From these rules a dataset of synthetic samples that represent the conditions and effect of the adaptation rules is created. Machine learning tools are used to infer the rules back from the synthetic dataset (at this point the set of rules should be equivalent to those provided by the expert). Monitor the system and collect samples from observed adaptations. These samples are added to the dataset and machine learning tools are again used on the augmented dataset to update the rules inferred previously. If new adaptation effects are learned, these are used to improve the original set of rules produced by the expert.

The project will produce the following expected results:

> *Expected results:* The work will produce i) a specification of the algorithms used to refine the knowledge provided by the expert; ii) an implementation of a self-adaptive system that uses these algorithms, iii) an extensive experimental evaluation using a concrete case study.

3 Case Study

In this section we describe the case study that will be used to assist the description of the systems presented in the related work, and also introduce some terminology we will be using throughout the report.

To illustrate the concepts and approaches covered in this report we will often use a simplification of the ZNN case study extracted from the work of Cheng et al. (2012) [6]. This case study considers a web-based application that is supported by a pool of servers. Clients make their requests to a load balancer that forwards them to a given active server. The aggregate load on servers, i.e., the number of request per unit of time, may change dynamically.

Goals capture the high level objectives that the system should try to achieve during its execution. In our case we assume that the goal of the provider is to keep the service with low latency while minimizing the costs, which are proportional to the number of servers instantiated at a given point in time. Therefore, our goal metrics are the average time interval the user waits until a response to its request arrives and the number of active servers. By *goal metrics* we mean measurable performance indicators of the system operation that are related with the system goals. These metrics should be monitored by the system.

Utility is a quantitative evaluation of a certain system state, based on the system goals. It is derived by combining the goal metrics. A possible representation of the utility is a linear function of the goal metrics, where the weights associated with each variable correspond to the preferences among the goals (higher weight correspond to a more important goal).

The system must perform elastic scaling, by adapting the number of servers to the experienced load. To further simplify the problem we will consider that there is no limit to the number of servers in the available server pool.

In this example, there are two adaptations that can be performed on the system: to activate or deactivate a single server. The *effect* of an adaptation is the result the adaptation has on the system. Activating one server will increase the number of active servers by one. The *impact* of adaptation is the observed change in the goal metrics caused by an adaptation action. In this case, the impact of activating a new server would be a decrease of the average response time (presumably).

Naturally, each of these adaptations may be performed multiple times in sequence. In order to drive the adaptation, the system has to maintain information about the number of servers that are active (from which the current cost may be directly derived), about the service latency, measured as the average time interval between the client request and the reception of the corresponding reply, and about the load on the system (number of request per unit of time). This information represents the current state of the system.

4 Background

4.1 Basic Definitions

A system is said to be Self-Adaptive if it has the ability to monitor its context and to automatically change its behavior such that a predefined set of goals, that capture human needs and preferences, are respected [8]. The ability to adapt without requiring human intervention has the potential to provide faster response to changes, to reduced downtime, and to reduce administration costs [9].

The work of Salehie & Tahvildari (2009) [9] differentiates the process of engineering self-adaptivity into the system at the development phase (Making Adaptation), which is usually used with static policies based systems, and using artificial intelligence to learn a model of the system and use it to adapt the system (Achieving Adaptation). We will analyze both methods during the related work section.

4.2 Approach

Self-Adaptiveness is one of the properties of an autonomic computing system [9]. The main building block of a self-adaptive system is an *autonomic feedback control loop* [10]. The autonomic feedback control loop usually used in autonomic computing consists of the MAPE-K loop illustrated in Figure 1. The loop executes four different steps, that share (K)nowledge about the system [2, 9, 11]:

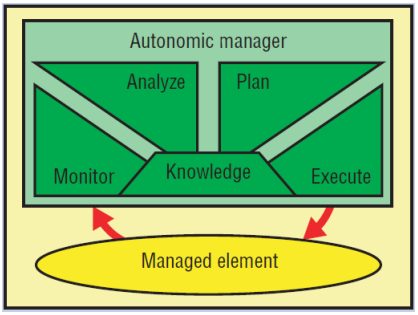

Fig. 1. MAPE-K loop (picture taken from [2])

- (M)onitor: Collects information about the system, and aggregates the collected data into symptoms if they need to be analyzed. In our case study, this task consists in monitoring the response time to user's requests.
- (A)nalyze: Receives the data collected in the previous step, and analyzes it, to check if changes are required. In case change is required, it passes a request to the next step of the loop. In our example this step consists in assessing if the response time is higher than the target threshold (in which case a new server needs to be activated) or if it is much lower than the threshold (in which case a server may be removed to reduce the cost).
- (P)lan: This phase is where the sequence of actions (or single action) is chosen to adapt the system to the desired state. The constructed plan is then passed to the next phase. A possible plan, in our case study, is to activate the number of servers required to lower the latency.
- (E)xecute: Receives as input the plan of adaptation and applies it to the managed system. Following the example, this step would connect the new server.

The (K)nowledge component of the MAPE-K loop is responsible for storing important information about the system – actions it may perform on the system, their effects/impact, when it should adapt the system, what are the goals that should be achieved, policies (possibly), among others.

Architectural models provide a high-level view of the system, based on components, interconnections (connectors) and their properties. Components correspond to the computational elements of the system and connectors to interactions between components. The model can be extended with information about its elements (e.g., properties, constraints). This approach is appropriate to use in the Knowledge component given that it provides an abstracted view of the system, a general model (one model describes a class of systems, thus may be used in systems of the same class) and the constraints associated with its elements help to identify allowed changes and when the system needs to be adapted, during the adaptation process [4].

The steps of the loop are also called collect, analyze, decide and act in some works [10, 12], but their meaning remains the same.

This work will focus on the Knowledge component and how it can be learned and updated. In the related work, we will also consider the analysis and planning components, by stating how other components implement it, in order to support our claim.

4.3 Challenges

Below, we identify the main challenges that must be addressed when building a self-adaptive system:

- *Achieve and maintain goals:* The system should adapt in order to achieve its goals, even when dealing with a highly dynamic environment.
- *Adapt effectively and timely:* If the system is adapted every time there is a small change in the environment, it may become useless for its main purpose because it is always busy adapting, so a trade-off must be considered. Furthermore, it is necessary to choose between reactive and proactive adaptation, because some systems cannot afford to have their goals violated, even if transiently [9, 12].
- *Consider delayed effects:* The impact of an adaptation action may not be immediately observed.
- *Consider uncertainty:* Sometimes a given action may have different outcomes, and which of the effects takes place cannot be predicted based on the observable variables. In this case we say that the outcome is uncertain [8].
- *Strive for generic solutions*: Ideally, the solution proposed should be able to be used in different self-adaptive systems without a lengthy process of modifications [9].

To this set of challenges we add some that we want to solve, in order to attain our defined goals:

- *Modeling the adaptations:* Plans may be defined by experts or be built at run-time using planners. Plans must be built based on the knowledge of the system and the state it is in. To be able to build a plan, the impact and cost of each action must be known, in order to know which actions will lead the system to the state the planner wants to achieve [9]. So impact and effects of actions must be known and modeled.

– *Support human involvement:* We consider that the adaptation model of the system should be *easily interpretable* (policies, impact) by a human operator, as this will facilitate human involvement in the adaptation system, which is fundamental to build trust on the system and to improve its manageability [9, 10, 12]. By *easily interpretable*, we mean that (1) it should be easy to extract information by reading the policies or model, and (2) it should provide enough information to understand the system, e.g., when considering uncertainty, all the different expected outcomes should be presented, instead of a combination of every possibility.
– *Keep the model up-to-date:* The model of the system should be kept up-to-date because we are considering highly dynamic environments, which can influence the behavior of the system, and new situations can arise frequently. A possible solution to this challenge is to use learning techniques to improve the accuracy of the model – effects of actions on the system and their impact on goal metrics.

5 Related Work

5.1 Static High-level Policy-based Adaptation

It is often the case that these systems, where self-adaptation is appropriate, are currently handled by human operators that adapt the system by executing a sequence of adaptation actions. This means these operators already have a mental model of the system and how it could be adapted to some situations.

The human operator must be able to interpret the model of the managed system, because it may be useful when he must intervene, e.g., when a hardware fault occurs. Also, if the model is easily interpretable by the operator, it is easier for him to trust it, as he will be able to know how it captures the system's behavior [3, 10]. The following approaches are the ones that offer the models that are easier to interpret by the operator, as they are intelligible and represent uncertainty explicitly.

In this section we present work that leverages the knowledge collected from human operators to define adaptations based on static high-level policies. This kind of policies consists of a set of rules of the form "if the system is in a given state, that could be improved or does not respect the requirements, then it should make this change in order to adapt". Then in the analysis phase, if the system is in a state that corresponds to the precondition of any of these rules, the consequent of the rule is used as the plan of adaptation.

This is the most simple method, despite the possible conflicts that may arise among rules (more than one precondition can be true). To solve these conflicts a human operator must be kept in the loop [3].

5.1.1 Stitch

Cheng et al. (2012) [6] propose a language that can be used to represent the

operator's knowledge about adapting a given system. Stitch is used to define adaptation strategies, based on the architectural model of the system.

The purpose of this work is to capture the expert's knowledge about the system and how it should adapt to different conditions. Here adaptations are based on expert defined strategies – sequence of steps the expert would take to adapt the system to the current conditions.

The intuition behind the Stitch language is that the system can choose how to adapt itself as an expert would adapt it. So when adaptation should be considered, the system will see which of the defined strategies results in the state with the highest utility. Then the strategy's steps are executed.

Adaptation serves the purpose of keeping a certain quality of service level even when confronted with dynamic environments. When choosing the best strategy the system has to know which strategies are available, which can be used at that point, the objectives of the organization (such as always provide high fidelity, or save as many resources as possible) and its priorities (e.g., what objectives are more valuable to the organization). To represent this knowledge Stitch supports the specification of business QoS concerns, represented as quality dimensions; the priorities among the dimensions, which are specified through an utility function based on the quality dimensions; and how actions affect goal metrics (quality dimensions). The latter is achieved by defining the impact an action has on each goal metric (impact vectors – each cell corresponds to the change on one goal metric).

One characteristic that Stitch tries to mimic from the way an expert executes the actions is the variability of the path taken. An expert can go through different paths for the same strategy considering the result of previous steps. If a step fails - effect is not observed within a time-window – the expert may try something else. To support different paths within the same strategy, every step has an applicability condition saying if the step should be executed based on the state of the system (determined by the success or not of previous steps) and an expected result so it is able to check if it had the desired effect. The result of a given action may not be instantaneous, and for this reason, Stitch also supports the definition of a delay window in which the result of an action may be observed. If the strategy did not solve the problem, the next cycle of adaptation will try to solve it.

Stitch defines strategies as decision trees, where each node is a tactic and each tactic has operators. An operator is a configuration command. A tactic has a collection of operators, its expected effects and impact on the goal metrics, guarded with an applicability condition. It is a step of adaptation used by strategies. In a strategy each step is a conditional execution of some tactic, thus the path is dependent on the tactics' effect, as it was already stated. The effect of a tactic may not be immediate, and therefore each tactic has a delay time-window – defined by the expert – in which its effect must be observed or it is considered that the tactic has failed. A possible tactic, in our case, is to activate a server. Strategies also have applicability conditions that state when a given strategy may be applied, in order to reduce the search space when searching for

the most appropriate strategy, e.g., the server must be off-line before activating it.

This language also supports the explicit representation of the uncertainty associated with the outcome of an action. This is represented with probabilities attached to each possible path after the execution of an action. The path taken depends on the outcome of the previous action, and is not influenced by the probabilities. The probability can be defined by experts and represents the number of times each of the paths is taken. The probabilities are used to compute the utility of a strategy, by stating the probability of reaching each of the possible states. The utility of a strategy is then a function of the utility of each possible final state and the probability associated with each one of them. Reducing the response time, in our case, is possible only by activating a new server, so the probability of this condition is 1.

Listing 1.1. Example of Strategy represented in Stitch

```
// latencyViolation is a boolean variable, true if the
//latency is above a certain threshold
strategy ReduceResponseTime [latencyViolation] {
  // hiLatency is a boolean variable, true if the latency
  //is considererd to be high
  // hiLoad is a boolean variable, true if the load is
  //considererd to be high
  t1: ([Pr] hiLatency & hiLoad) -> enlistServer(1) @[2000/*ms*/] {
    t1a: (!hiLoad) -> done;
    t1b: (!success) -> do t1;
  }
  tn: (default) -> fail;
}
```

The example in listing 1.1 represents the characteristics presented in this section. The $[Pr]$ represents the probability of that condition being chosen, in this case it would be 1. However, if we considered more conditions the probabilities would be distributed among them. The strategy is considered if a constraint is violated (response time above a threshold), and a server is activated if the latency and the load of the system are high. The default clause is for the case where all the previous conditions are false.

Strategy selection is the process of choosing the best strategy to follow when the system is in an adaptation condition – state for which adaptation should be considered. It starts by computing the impact, per goal metric, of each strategy, and the overall utility is computed using the goal metrics weighted by the utility preferences. In our case, this corresponds to having a linear function based on the number of active servers and the average response time, where the weights would have to represent the preferences, for example it is more important to provide a service with a low latency.

5.1.2 Expert Defined Impact Models

The work done by Cámara et al. (2014) [7] describes a solution intended to improve Stitch [6], regarding the description of the impact of actions.

In Stitch, each action has an impact vector, where each value corresponds to the change in a goal metric, e.g., cost. The vector's values are the combination of the different possible impacts an action can have on each goal metric. This is a simplistic approach that does not consider that the outcome of a tactic

may differ. The outcome of executing an action is uncertain, e.g., activating a server can be successful or not. It also depends on the context at that time, e.g., activating a server has a lower impact when there are already a lot of active servers than when just one additional server is active.

In order to chose a strategy it is important to have a good estimate of the impact each applicable strategy will have, if applied, to be able to chose, at runtime, the best for the current context. If we have high latency we do not want to disconnect a server.

To improve the description of the impact models, the proposed approach is to create a declarative impact specification language that is able to represent the uncertainty in the outcome of actions and the dependency of the current context.

Stitch represents the uncertainty with the different possible paths the system may take at a given state during the execution of a Strategy, not with the different effects an action may have on the goal metrics. So the different effects are represented implicitly and are harder for the operator to analyze. This work tries to mitigate this problem. To represent the uncertainty of the outcome, the models are based on Discrete-time Markov Chains (DTMCs), which allows the representation of different outcomes, of an action, with some given probability. The impact model is defined in terms of probabilistic expressions ($\{[p_1]\alpha_1 + [p_2]\alpha_2\}$, which means that α_1 will happen with probability p_1, or α_2 will happen with probability p_2).

The outcome of an action, depends on the context the action takes place in. This dependency is represented by expressing the outcome as a function of the state, instead of a fixed value.

So the language described supports the specification of the impact of an action on the goal metrics. Each action is then described with the possible outcomes that may be observed by executing it, the likelihood of observing that outcome, the expected impact on the goal metrics and the effects that are observed on the system.

Listing 1.2. Example of definition of Action

```
impactmodel enlistServer
// m is the number of active servers
m > 0 -> {[0.95] {foreach s:S | s.isActive'=true &
                    {[0.7] forall c:ClientT | c.expRspTime'=f(c.expRspTime)
                   + [0.3] forall c:ClientT | c.expRspTime'=g(c.expRspTime)}}
        + [0.05] {forall c:ClientT | c.expRspTime' = c.expRspTime &
                  forall s:ServerT | s.isActive' = s.isActive }}
```

The example in listing 1.2 shows the representation of the uncertainty ([probability]), and the dependency of context (functions f and g).

In order to compute the utility of Strategies, we can think of the possible paths and outcomes of such strategy as a tree with normal and chance nodes – chance nodes are the ones where the choice of edge to follow is external to the system – that alternate in consecutive depth levels. Each normal node represents a state, usually followed by edges that represent possible actions to perform in said state, e.g., connect a server. Chance nodes are followed by edges that correspond to each of the possible different outcomes of the action chosen. Each edge

has a probability associated with it. When it comes to normal nodes the probability is equally split between the possible actions (edges). Edges that connect chance nodes to normal nodes have a probability associated that corresponds to the likelihood of the outcome, represented by that edge.

Given the tree and based on DTMCs, computing the strategy's utility is a matter of summing the utilities of the states associated with each leaf node, each one weighted by the probability of reaching said state. The probability is given by the product of the probabilities in the path to that node.

Experimental results showed that probabilistic impact models predictions were more accurate than the approach seen in Cheng et al. (2012) [6].

5.1.3 Automated Planning for Self-Adaptive Systems

A problem from the previous approaches is that strategies – sequence of actions to adapt the system – are static. This problem burdens the expert, in the sense that he has to think of every possible context the system may be in and every sequence of steps that may solve a given problem, which is a difficult task [13]. This turns out to be a much more complex problem if we consider that contexts that were not considered before may occur.

A generic solution is proposed by Gil (2015) [14], for this problem of static strategies. This work proposes the creation of plans in runtime, based on the available actions, their impacts on the goal metrics and the utility function. This is a dynamic decision making approach [9].

The process consists of using each action, its precondition (applicability condition) and impacts to create operators in a standard planning language, and use a planning algorithm to create plans that are supposed to lead the system from the current state, to a goal state – a state where every metric is within its goal values – if applied.

In order to account for the inherent uncertainty associated with the impact of each action, the author considers that each action can have more than one outcome, each with a certain probability. This can be represented by the language defined by Cámara et al. (2014) [7] and described earlier in section 5.1.2. For this reason, for each state that may result from the previous action/step, a planner is used to generate new adaptation paths. The result is a tree, where each node is a state, and edges represent outcomes from a given action, with probabilities associated.

After the generation of this plan, the adaptation system should just use this tree to choose the action corresponding to the state it is in.

This kind of approaches rely heavily on the accuracy of the model of adaptation (impact model – probabilities and impact functions). If an inaccurate model is used, the sequence of actions derived from the planning process will result in a completely different outcome than the one expected, or the complete invalidation of the selected plan. This is due to the accumulation of the inaccuracy and uncertainty over the action sequence. Therefore, having an accurate and updated model is important for systems like the one just described.

5.1.4 Discussion

Policy based adaptation provides a higher flexibility when it comes to choosing the representation of the system. This can be seen by the explicit representation of the uncertainty used in Stitch and the impact modeling language presented. These approaches present more information than the one that is provided by using learning techniques (to be discussed next) where the uncertainty is represented implicitly. Therefore, the approaches presented here provide more knowledge about the system to an operator.

Representing the uncertainty explicitly also allows systems to use this extra information in a different way than how it is usually treated (the impact of an action is defined by the different outcomes and the number of times each one occurs – information implicitly represented by a single impact function). An example of such application would be a planner that considers the worst possible outcome of each action, e.g, manages critical systems, so that the final plan results in a state that is at least as good as a certain threshold. This approach is not possible without access to the possible outcomes explicitly.

The work described in section 5.1.2, when compared to Stitch [6], presents an approach that provides more and easily interpretable information, to the operator. It also facilitates approaches, such as the one described of automated planning [14], by providing a more explicit representation of the impact and effect of actions.

Even though the expert defined models are better when it comes to representation of knowledge, it still has the problem of being static. The model is built during the development phase and is not changed, even if it does not model the behavior of the system correctly.

For these reasons we will consider the approach of expert defined impact models [7] and how to improve the accuracy of the model defined by experts, by learning from data collected from the running system.

5.2 Machine Learning based Adaptation

The next sections will describe techniques that learn the system's behavior, based on data collected from it, and use the learned model to manage the system.

Machine learning consists of a class of algorithms that learn patterns, and make predictions or decisions based on data. Examples of such algorithms are reinforcement learning, decision trees and artificial neural networks based algorithms [15]. The results are models of the collected data that can be used to make predictions, or a set of rules to guide the decision making process, if that is the case. Usually, these predictive or decision making models can be used as black boxes, in the sense that one can feed them with data and they will return the expected output or decision.

In the following sections we will consider: (1) Prediction based approaches, where decisions are made by predicting the system's needs in the future based on its current state and (2) decision making approaches, where an action is chosen based on the current state

The state of the system is usually represented as its current configuration and representation of the environment. It is also possible to add information to the state's representation by adding past states (history of the system). However, this leads to a huge growth on the number of states the adaptation system has to consider.

Unlike the previous approaches (expert defined policies) techniques based on learners derive their models from data collected, in our case from a running system – training phase. This has the potential of leading to a more accurate model, since it deals with the system directly instead of the idea an expert has on the behavior of the system.

However, it is not easy to extract the learned models, as an easily human interpretable model, mainly because it is not a main concern with these approaches, and for this reason they provide low flexibility to the way the model is represented.

A learned model that is not refined after it is created presents the same disadvantage than the expert defined models – loses accuracy and does not deal well with unforeseen scenarios. The techniques presented in the next sections continue to refine the learned models with collected data from the system, after the model is created.

5.2.1 Fuzzy Control

Fuzzy sets are an approach to approximate the set theory to the way humans think. For example, using the classical set theory, for someone to be classified as tall, a precise threshold, that captures the height above which a person is considered tall, must be defined. However, most humans do not reason in terms of these precise thresholds but, instead, use subjective values. Therefore, classical set theory does not represent the way people usually think. In the real world it is common to have classes of objects that cannot be clearly separated, as is the case of tall and short. To solve this problem fuzzy set theory uses classes with grades of membership. An object, instead of belonging or not to a given class (binary classification), has a degree of belonging, given by the membership function corresponding to that set (captured by a value in [0, 1]) [16].

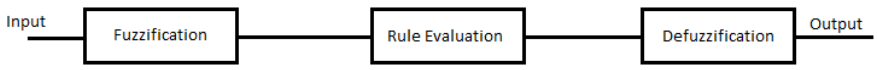

Fig. 2. Fuzzy Inference Process

This notion of fuzzy sets can be leveraged to create a rule base, i.e., a set of rules, that maps from an input space to an output space. These rules can be

built by human experts, or using existing data (input/output) to automatically learn them [17]. Each rule is of the form:

$$\texttt{if } x_1 \texttt{ is } A_1 \wedge x_2 \texttt{ is } A_2 \wedge \cdots \wedge x_k \texttt{ is } A_k \texttt{ then } Y$$

Where x_i is a variable, A_i corresponds to a membership function (fuzzy set) and Y corresponds to the output. The mapping corresponds to the notion of a Fuzzy Inference System (FIS). The mapping process is represented in Figure 2.

- The first step of the process is the fuzzification of the inputs. Using fuzzy set theory, each input is used to compute the degree of membership to each fuzzy set, using its membership function.
- The second step is the evaluation of the fuzzy rules. Here, the firing strength of each rule is computed based on the degree of truth of the precondition of that rule. The firing strength is then combined with the output of that rule. The output of every rule is combined and sent to the last step.
- The last step, defuzzification, receives the output of the rules' evaluation and converts this result to a real value, a value that makes sense in the real world (not fuzzy).

The use of Fuzzy Logic is very common in the field of self-adaptive systems, for many reasons, including the lack of requirements for previous knowledge about the system, its robustness when confronted with noisy data, its ability to quickly adapt to changes and the fact it does not need a lengthy training phase. It is also known to be robust when approximating non linear functions and dealing with uncertainty.

We will now describe some approaches that use data collected on-line, to build an inference system, in order to decide how to adapt the managed system.

Fuzzy-modeling: We use the work of Xu et al. (2008) [18] to illustrate the fuzzy modeling approach, which consists in using data collected via monitoring, to construct a fuzzy inference system. Such system can then be used to implement reactive adaptation [9].

The approach works as follows. Each sample from the data that was monitored and respects the defined goals (the system should only be trained with samples where its goals were achieved) is considered to be a point in an N dimensional space, where N is the number of observable and relevant variables in the system. Therefore, each point is a tuple $\langle$ $input_1$, $input_2$, ..., $output_1$, $output_2$,...$\rangle$. The inputs correspond to the values of the variables that represent the state of the system. The outputs are the values that were observed, and that the system wants to predict in the future. In order to predict what the system needs to reach a goal state, it filters the samples that are far from the defined goal. When applied to our example, the inputs are the response time and the load, and the output is the number of servers being used.

Based on these samples, the rules and membership functions are induced, using a fuzzy clustering algorithm (in particular, Xu et al. (2008) [18] uses a form of clustering called subtractive clustering [19]), which will aggregate similar

points in the same cluster. Each cluster therefore represents the behavior of the system in a given general situation. As a result, the fuzzification and computation of the firing strength of a rule is done by computing the distance from a sample to the cluster center of that rule (by considering only the dimensions corresponding to the input variables). Clustering the samples also avoids getting a large number of fuzzy rules. In order to increase the accuracy of this model, a first-order Sugeno-type model can be used, where the output is considered to be a linear function of the input variables [20]. During the normal execution of the system the model is continually updated based on new monitored samples (Fuzzy set parameters and output functions).

The result of the previous step is a set of rules that map from the input to the output. These rules are then used to predict the behavior of the system which will determine the adaptation that is going to be applied. A possible example during the execution of the system is, based on an observed load, and on the response time to user's requests, the model states that n servers should be active.

Fuzzy Neural Networks: We now describe how ideas from neural networks can be used to implement a fuzzy controller (in an approach that is called a fuzzy neural network). Inducing a fuzzy inference system, using neural networks, is usually done using an Adaptive Network based Fuzzy Inference System (ANFIS) [21]. This presentation, however, is based on the work of Lama & Zhou (2010) [22]. Using a neural network to model the fuzzy controller allows the learning techniques that build and update neural networks to be used, in this case, to build and update a fuzzy controller as it learns from the running system. An advantage over the system described earlier is the flexibility of the neural network.

Using this approach, the controller is designed using a four layer neural network. These layers roughly correspond to steps in the fuzzy controller: fuzzification, rule evaluation and defuzzification. In layer one, each node corresponds to one input variable, and its purpose is to pass its signal to the next layer. In layer two, each node corresponds to a fuzzy set, and its membership function is used to determine the degree to which an input value belongs to this set. This layer corresponds to the fuzzification step. In layer three, each node represents the precondition of one fuzzy logic rule. Each node outputs the firing strength of that rule, i.e., the degree to which the rule is satisfied. The fourth and last layer represents the defuzzifier. It receives the results from the third layer and outputs its combination, already defuzzified.

Initially the network only has input and output nodes: the fuzzy neural network requires no knowledge about the system before it starts. By processing samples, the network will determine the parameters necessary for a fuzzy controller (membership functions and rule base) in the form of nodes and the connections between them. These nodes are added dynamically using a learning process that will be described next.

When an input is received and the current network does not model it correctly – the firing strength of the existing rules is below a certain threshold – then it needs to be updated. This update is done in what has been called the Structure

Learning Phase. This process adds a new node to the network in layer two and the associated rule node in layer three. The membership function is only generated if there is not already a similar one. If a similar one does exist, the corresponding node is used. The new link between nodes will have a constant, or random weight assigned, and this weight will be updated by another procedure called the Parameter Learning Phase. This last phase uses the well-known backpropagation learning algorithm [15], to update the weights of the network according to the gradient of the error associated with the samples.

We illustrate how this approach could be applied to our case-study. We could use, as the input variables, the current response time and load, and, as the output, the number of servers that should be active in order to achieve the goal latency. The backpropagation algorithm would have to tune the network parameters to minimize the error of the predicted output, based on the observed one.

5.2.2 Optimizing the Exploration Phase

The presented Fuzzy control based approaches described a way of predicting the needs of the system, based on the current state. However, they left little knowledge about the impact that actions had on the goal metrics, and did not necessarily maximize the utility, given that it was based on "heuristics". By heuristics we mean that these systems only train with samples that result in a good enough utility and are not concerned with further exploring for other alternatives, therefore, it does not achieve the global maximum.

Exploration in complex computer systems usually result in a state explosion due to the presence of a potentially large number of input variables. This can be problematic if one wants to build the system's model on-line (using observations from the running system). In this case, it is typically prohibitive to randomly explore the entire search space, as the system would likely deliver unacceptable performance.

Analytical models that describe systems, and the impact a certain configuration has on the goal metrics, may be learned automatically or be written by an expert.

In this section we present some work that aims at optimizing the exploration phase, by minimizing the time spent in undesirable regions of the search space. This is done by leveraging analytical models of the systems, previously defined or learned on-line, and use them to accelerate the training phase.

iTuned: The work by Duan et al. (2009) [23] describes a process to search for the best state a system could be in, given the current context in which it is inserted. To help the search process it builds an analytical impact model of the system. That is later used, as was already said, to find the optimal configuration.

The goal of its authors was to perform an educated search for the best configuration. Using results from previous experiments they could extrapolate the utility of new states without exploring them, thus avoiding searching the entire search space and have a planner choosing which states to explore next. This approach leads to a smaller computational cost, and raises the chances of getting

good results early in the search, given that only a sub-set of the search space is explored.

The algorithm behind this tool consists of an initialization phase followed by a cycle in which the state of the system for the next experiment is chosen and tested. The cycle terminates when there are no prospects of finding a better configuration or when the user terminates it manually.

In the first step of the algorithm, initialization phase, iTuned has no knowledge regarding the impact model, so it has to do some experiments to collect samples and add them to the database. The way the samples are chosen are out of the scope of this report, and we consider that they use a process as simple as random sampling. An experiment is represented by $X =< x_1 = v_1, ..., x_n = v_n >$. The outcome of the experiment is a sample represented by $< X, y >=< x_1 = v_1, ..., x_n = v_n, y = p >$, where x_i represents a characteristic of the system, v_i is its value and y is the utility measured for that state.

To select the next experiment, iTuned uses Adaptive sampling, a novel feedback-driven algorithm. It analyzes samples collected so far to approximate a surface – each point of this surface is a sample $< X, y >$ - and chooses what should be the next experiment to be tested based on the likelihood of getting a better result – the next experiment should be the point X with the highest value of y on the mentioned surface. This surface is built using a model called Gaussian process Representation of a response Surface (GRS). This model uses the available samples to update the mean and variance of the Gaussian random variable that models the utility estimate.

With the next experiment selected, the configuration settings are tested and the result – $y(X)$ – is used to update the GRS model.

The number of parameters that is considered during the search is an important factor, because the larger it is, the longer the search takes. With this in mind the authors proposed that some parameters do not need to be considered, as their effects on the utility of the system is negligible. The filtering can be done with knowledge from an expert, or by computing the information gain of each parameter (information gain – changing the value of a variable results in a change in y).

This approach could be applied to our example if we consider that a parameter, load, is not under the control of the system, and therefore cannot be modified in order to make certain experiments. This causes the search space to be restricted to certain areas at each time. The input space could be the number of active servers, the load and the response time, and the output the utility function that combines the response time and the cost associated with the active servers.

This work requires no initial knowledge, and after some experiments finds a better configuration than the initial one, if there is one. It is also able to consider only a subset of the possible states by reducing the search space when looking for more promising configurations. However, it still requires that each of these considered states to be tested, in order to find their utility value. This test may be a lengthy process if we consider that an adaptation does not take

place immediately, and that this change may not alter the performance metric instantaneously.

Although it is not the main concern of this work it is possible to extract a graphical representation of the GRS, which provides information of how certain configurations affect the metric goals.

FUSION: Elkhodary et al. (2010) [24] presents FUSION, a framework that learns the impact of an adaptation on the system's user defined goals, in order to plan the sequence of actions to perform when adaptation is needed – a goal is not being satisfied.

The work done here is different than the previous one, because it leverages knowledge from experts and it uses machine learning as a way of learning the analytical model.

This framework uses knowledge from experts to reduce the number of possible configurations, thus reducing the search space, when looking for better states. The knowledge collected is in the form of relationships between features. A feature may depend, or have a mutual exclusion relation with another feature. Experts are also responsible for defining the QoS objectives (goals) using a metric (measure that can be obtained from a running system) and a utility function (represents users' preferences).

FUSION uses machine learning – model trees – to induce an analytical model, based on the monitored data. When a goal is not being satisfied, it uses the induced model to adapt the system to satisfy its goals. Each process is performed in a separate loop. The process of learning the impact model is performed in the learning cycle, and the adaptation in the adaptation cycle.

During the learning cycle, FUSION discovers the impact each feature has on a given metric. This process can be divided into two parts, the Observe and Induce.

Observe checks if the impact model is accurate, based on the last observations. This is done by computing the difference between the predicted value, given by the model, and what was observed. If the accuracy is below a certain threshold, it is assumed the system is not being correctly modeled. So Induce is asked to tune the model based on the latest observations.

Induce uses the observed samples to build impact functions – function that have features of the system as a domain, and output the impact on a given metric. The learning algorithm used is the M5 model tree – a decision tree that allows the use of linear regression models in the tree leaves. The first step is to use a significance test to discard variables that do not influence a given metric (already done by M5). Then use M5 to derive the functions that model each goal, based on the features of the system. The context may influence the impact of the system and for that reason it is also considered in these functions. This results in different functions for the same goal, each one representing a different context.

A possible function that could be induced for our case study, where s is the number of servers, l the response time measured and M_G1 the impact on the

goal metric that corresponds to the response time.

$$M_G1 = \begin{cases} -20s + 2l + ... & s < 7 \\ 0.5s + 2l + ... & s \geq 7 \end{cases}$$

The adaptation cycle initiates when a constraint is being violated, instead of every time a better state is achievable, in order to decrease the interruptions of the usual operation. This could also be done in the previous described systems, by defining constraints and initiating the planning phase only if they were violated. Then FUSION generates an optimization problem, where it wants to find the configuration that increases the utility of the system, while no constraint can be violated, and the restrictions, e.g., mutual exclusion, defined by the experts must be respected.

The last step is to change the system to the new state, the one found in the previous step. During this step the main concern is to respect the restrictions between features, e.g., two mutual exclusive features cannot be enabled simultaneously. For this reason a plan of adaptation actions is built, in order to get the system to the new state, where at any given time, every user defined restriction is respected.

This can be adapted to our case study by considering two goals, the latency and the cost (M_G1 and M_G2). The inputs would have to be the response time, the load and the number of active servers.

Bootstrapping: The work done by Didona & Romano (2015) [25] describes a technique that combines machine learning and analytical modeling to create a predictor, which requires a short training phase and achieves a high prediction accuracy.

It uses a user defined analytical model, of the variable that it wants to predict in the future, to create synthetic samples that are used to train the machine learner. Using such samples eliminate the need to collect them from a running system, which may reduce the training phase considerably.

Building an analytical model capturing the dynamics of complex systems is well known to be a difficult task. It is, often, also necessary to make some assumptions about the system, to be able to deal with its complexity, which leads to models that are possibly inaccurate. Therefore, when new samples are collected from the system to keep the model up to date, the synthetic samples may be contradicted by the new ones, ultimately changing the predicted values.

To deal with the possible inaccuracy of the analytical model this work proposes four approaches to update the dataset with the a new collected sample: (1) *merge*: collected samples are added to the data set and weights are used to give more relevance to real samples; (2) *replace based on nearest neighbor*: for each new real sample it finds the nearest sample, synthetic or real, and replaces it; (3) *replace based on nearest region*: replaces synthetic samples within a certain range for the new real sample; (4) *replace based on nearest region 2*: replaces the output of the nearest sample with the output of the new sample.

5.2.3 Reinforcement Learning

The learning approaches seen until now make their decisions by predicting the needs of the system, based on its state, e.g., how many servers are needed if the response time is high and the load is also high. This section describes an approach that maps states to actions, e.g., activate or deactivate a server.

Reinforcement learning is based on the notion of rewards received as the consequence of executing an action in a given context. In order to collect information about the actions that maximize the reward received, the learner has to try these actions in different contexts. With the notion of reward per action in each state it learns policies that adapt the system [26]. This introduces the problem of knowing when to explore – select an action that may not be the optimal one, in order to collect more information about the system – and when to exploit - based on the current knowledge, select the action that is expected to maximize the reward [27].

The result of this process will be a set of adaptation policies. Thus, it may be considered to be in the class of policy-based adaptation.

Reinforcement Learning is used in this field, because it is able to learn about a system without an initial model of it and with little knowledge about the system itself. However, it can use an initial model to facilitate the training phase. Another advantage is the fact that reinforcement learning approaches are capable of handling delayed effects of actions [28].

A problem of this kind of learners, is that it needs a big training phase to learn an accurate model [3].

Hybrid RL approach: Tesauro et al. (2006) [28] describe a reinforcement learning approach to learn management policies, i.e, map from a state of the system to a management decision (adaptation action).

In order to minimize the exploration phase, which can lead to poor results in a running system during the training phase, a scenario usually not affordable, the system uses a set of policies in the beginning (which encapsulates expert's knowledge), and trains the learner off-line.

The set of policies is used in the running system, while samples are collected. Samples are of the form $< s, a, r, t >$, where s represents the state of the system, a the action taken and r the immediate reward at time t. The collected samples are then fed to a function approximator – here neural networks are used. In the case of the simple ZNN, the state is the response time to users' requests, the load and the number of active servers, the actions are activating or deactivating a server, and the reward may be the difference of the utility, or even the utility itself, since the point is to maximize it.

As a result of this training period, a model of the system is created. A model defines the utility/reward that the system should attain, by performing a given action on a specific state. Leveraging this model, a set of policies may be induced - if the system is in a given state, the action performed should be the one that maximizes the expected reward.

After collecting a big set of representative samples – necessary to get an accurate model – the learned model can be used instead of the initial policies.

In each cycle of adaptation, the system checks the state it is in, then for each possible action it may execute, checks the reward associated with it. Then it executes the action that gives the best reward. New samples are still collected to update the reward function approximator.

If we consider that a reward may not be immediate, an we want to consider delayed rewards, then the samples can have a reward at $t + 1$, and so on.

This approach minimizes the exploration phase, common to reinforcement learning approaches, that will increase the knowledge of the adaptation system about the adapted one. But choosing sub-optimal actions, in a deployed system, may result in costs that are not compensated by the model learned.

5.2.4 Discussion

Approaches that learn how to model a system by observing the way it behaves are now the most common techniques given that they require little, or no, initial knowledge, which results in a more generic solution [13]. Further, these approaches promise to reduce the costs of the teams that manage these systems while keeping a model that is more accurate and up-to-date than the alternative, static policies. A problem of these techniques is the fact that they were made to remove the need for a human to be involved, and for this reason, the models they learn are not usually easy for a human to interpret or have a flexible representation. When it comes to the trust a human has on a management system, the policy based adaptation systems have an advantage since, the human operator will trust more on something that he can easily analyze.

A usual problem with machine learning techniques is the time that is necessary for the training phase. Reinforcement learning is an example of a technique that needs lengthy training phases. Given that the system will be active during the training phase, before a good and accurate model is learned, the system will probably behave badly and will do bad adaptation choices. A way to mitigate this problem is to use some knowledge about the system, in order to facilitate the training phase, as was seen in Tesauro et al. (2006) [28] and section 5.2.2.

For the reasons presented, we want to use the automatic learning approaches to refine and update an expert defined model of the system. All of the presented learner based approaches deal with the uncertainty associated with the real world (but do not represent it explicitly), keep an updated model of the system, adapt quickly to changes, adapt if needed (or can be modified to do so) and can be modified to consider delayed effects.

In section 5.1.4 we chose to use the work described by Cámara et al. (2014) [7], combined with an architectural model of the managed system to model the adaptation actions' impact. To update and refine this model we will try to use the techniques seen in the approaches based on Fuzzy Logic first.

6 Architecture

We have previously stated that we will be leveraging the language defined by Cámara et al. (2014) [7] and described in section 5.1.2, combined with an

architectural model of the system, and use a learner to improve the initial model. In this section we present the set of challenges that we consider relevant to this problem, and how to solve them based on our assumptions:

- *Modeling the adaptations:* We have seen that it is important for many applications to have a model of the adaptation actions, their impact and effects. As it was done before, we plan on using the language defined by Cámara et al. (2014) [7], to describe this model, based on an architectural view of the system.
- *Support human involvement:* A concern that has been growing in relevance, is the trust the human operators have on the system. To solve this problem we consider that keeping the human involved is essential. For this reason it is important that the operator understands the model that the self-adaptation system has, about the managed system. We think that using an architectural model of the system (a common way of representing the system) and using a language such as the one presented in Cámara et al. (2014) [7] (that provides human readable information about the system model and explicit representation of the uncertainty associated with the adaptation's impact) attains that goal of understandability.
- *Keep the model up-to-date:* We consider to be dealing with highly dynamic environments, which may result in a different behavior, from the one expected, and in conditions never considered (unforeseen). For this reason, having an up-to-date model is essential if we want an accurate model. We have seen the importance of having an accurate model in 5.1.3. To solve this problem we plan on using techniques seen in the systems based on automatic learning, presented in the previous section, particularly the ones presented in 5.2.1.

Approaches based on Fuzzy Logic accomplish good results as a way of approximating non-linear functions and allow for hand written (appropriate given that we want to use expert's knowledge to bootstrap the system) or learned (we also want to update the model of the system) rules. Additionally, a common way of implementing Fuzzy Inference Systems is to define the output associated with an input as a linear function of the input variables. This corresponds to the notion of context dependency presented by Cámara et al. (2014) [7]. Another advantage of using this approach is that Fuzzy Logic allows more than one output for a given input/state. This may be a relevant detail for our work, since the work of Cámara et al. (2014) [7] explicitly models uncertainty by having multiple outputs (possible outcomes) to each input/state.

6.1 Proposed Approach:

An expert expresses the possible adaptation actions (and its guard conditions, effects, impacts and probabilities), the strategies (applicability conditions and body) as we have seen in section 5.1.2. We make an additional assumption that is quite frequently made in this field: the impact functions provided for an action

are restricted to linear functions. Hence, we can leverage existing techniques to describe the model, e.g., first-order Sugeno-type fuzzy inference system.

Then we use the adaptation actions' properties and express them as a set of points in an N dimensional space, where N is the number of relevant variables to define the state and the expected impact. For each outcome of a given action we generate a set of samples (points), proportional to the probability associated with that outcome.

We planned on using a clustering algorithm to correlate the input (variables of each samples corresponding to the input parameters) to the output values, and use them to induce a linear function that fits the samples – this could be achieved using techniques presented earlier, e.g., subtractive clustering. However, this process has the problem of not being able to guarantee multiple different outputs for the same input. If we generated samples from a rule that stated that for a given input X the outcome could be $f(X)$ or $g(X)$, the use of clustering does not guarantee that two clusters will be created, or that the points would be attributed to the proper cluster. This happens for two reasons: (1) clustering is based on distance metrics, e.g., euclidean, which groups elements that are nearby, based on the point (X, y), where y is the output value, instead of (X, f), and (2) if we consider that each sample has multiple inputs and one output, the output may have little influence on the distance between points, which would result, possibly, in the grouping of every outcome on the same cluster.

For these reasons we will take a different approach. We will consider clusters with crisp borders, instead of the fuzzy membership, in order to be able to assign a point to a given cluster. Another change is that we will use a different representation of the sample. Instead of having (X, y) we will use (X, f), where f corresponds to the function that better models the sample (with the lowest prediction error) – we can make the analogy with classification (the function is the class it belongs to). The classes are created, initially using the functions described by the experts. We have to make sure that different classes cannot exist in a single cluster – this problem can be solved by attributing big weights to the function parameter, or by using semi-supervised learning.

From the managed system we collect samples (after executing actions and waiting for its effect) and add them to the set described earlier.

The addition of samples is done by computing the cluster that better represents them. Usually, this would be achieved by computing the distance between the cluster center and the sample. However, we are dealing with functions in one of the dimensions (the output dimension is a function and not a point in space). Therefore, we will compute the "distance" to each cluster by checking the rule that better represents the point. We start by comparing the input part of the sample with the rules' conditions – each condition is a combination of input metric ranges. The result will be the set of rules to which the sample can belong to. Each rule predicts a certain outcome. After that step, we compute the difference between the observed value and the prediction made (prediction error), and assign the sample to the cluster that corresponds to the rule that had the lowest prediction error.

From this process we consider that two cases may occur: (1) The sample belongs to a single cluster, in which case it is assigned to that cluster. (2) The sample does not belong to any cluster, in the sense that no rule's precondition encloses the sample (a state never seen), or the prediction error is too high (an outcome never considered). Ideally, we would create a new rule to represent these samples.

We consider that it is possible to detect the samples that fall in the second case. However, creating a new rule presents a bigger challenge. A state never seen or considered can be detected by checking the conditions of each rule and if none applies we consider it is a new state. To detect a new outcome for a given state, we could define a prediction error threshold, above which the sample does not belong to that cluster. If a sample does not belong to any cluster because of the prediction error, then we may considered that it represents a new outcome.

When enough points fall in this category (sample not represented by existing rules) we can use them to compute a new rule and outcome function. However, we think the new function does not provide as much information as the other ones, or at least not as precise. If we consider that all the points belong to this hypothetical cluster, then we can compute the output function – $h(X)$ – but this new function can actually model the samples inaccurately. For example if the points are accurately modeled by two different functions – $f(X)$ and $g(X)$, then defining a function $h(X)$ that tries to model both may not be the best approach (we lose information).

The collected samples that are added to the clusters are used to refine the rules' conditions, impact functions and the outcomes' probabilities.

The condition of a rule can be refined by updating the ranges to better represent the set of samples (increase or decrease ranges, based on the cluster's samples).

To update the impact functions we will use a similar process to the one used in fuzzy inference systems, specifically, first-order Sugeno-type models (outputs are linear functions of the input variables). This corresponds to solving the linear least-squares estimation problem [19], using the samples from the cluster.

The probabilities are computed based on a counting process:

$$P(outcome|state) = \frac{\#(state, outcome)}{\#(state)}$$

So computing these probabilities based on the collected and synthetic samples will result in updated probabilities.

We depart from the notion, that expert's knowledge may not be perfectly accurate. However, synthetic samples created based on that knowledge are present in each cluster, initially, which may skew the center of the cluster when new samples are added. For this reason our data set will be updated with samples collected from the running system, using one of the techniques described by Didona & Romano (2015) [25].

7 Evaluation

Our work will try to define an adaptation model of the system that is both easily understood by a human operator, and accurate on the previsions it makes. We will try to accomplish this goal by combining two different methodologies, which normally are considered as alternative approaches and are not used in synergy.

We depart from the point that the language described in 5.1.2, based on an architectural model is already easily understood. Therefore, what we want to evaluate is if our approach is able to increase the accuracy of the model by keeping it up-to-date, based on the monitoring of the system.

We expect to get a model that is less accurate than we would get if we used automated approaches, due to the constraints we are placing on our representation. However we expect that our solution decreases, significantly, the number of observations required to get an accurate model, when compared to automated approaches (assuming that the initial model is reasonably accurate).

The metrics we will consider in our evaluation are the average error of the predicted impact, when compared to the observed one, and the size of the training set needed to achieve an accurate model, which should tell us how fast the systems converge to a model.

To evaluate our proposal we will:

- Implement the system described in section 3. Use it to understand how the system behaves in different conditions.
- Based on the previous step, act as the expert, and describe the impact model of the system (adaptation actions and strategies).
- Implement the solution proposed in section 6.
- Apply an automated approach (using machine learning techniques) to our case study.
- Run the system in the same highly dynamic environment (simulate different workloads by altering the number of requests per second), using the three approaches. During the execution of the system, we will measure the error of the predictions it makes concerning the impact of actions and save the number of samples the system needed until it converged in an accurate model – the latter is not necessary for the hand written impact model approach.
- Compare the prediction errors of the different approaches and verify if the results are the ones expected.
- Compare to the samples needed for the model to converge between the automated approach and the one we proposed. We expect a faster convergence to an accurate model using our approach given that the expert's knowledge serves as a good starting point to define it.

8 Scheduling of Future Work

Future work is scheduled as follows:

– January 9 - March 29: Detailed design and implementation of the proposed architecture, including preliminary tests.
– March 30 - May 3: Perform the complete experimental evaluation of the results.
– May 4 - May 23: Write a paper describing the project.
– May 24 - June 15: Finish the writing of the dissertation.
– June 15 Deliver the MSc dissertation.

9 Conclusions

Self-adaptive systems are becoming a common way of dealing with the ever increasing complexity of computer systems. However, this is still a field of study with many open issues. We focused on the description of impact models – how adaptation actions impact the goals of the system.

In this report we have presented some relevant work made on this field, focusing on static policies based adaptation systems, and systems that learned the adaptation model at runtime.

We have seen that static policies based systems have some disadvantages, such as its inability to deal with unforeseen scenarios and to update its model when confronted with a dynamic environment. Automated approaches are presented as a possible solution to these disadvantages, however they also have some problems, particularly they tend to exclude the human operator from the adaptation process, which can lead to issues related with the trust the operator has on the system, which is an increasingly important issue to be considered.

We propose a solution that aims at combining both approaches, in order to get an adaptation system that is able to deal with a dynamic environment and includes the human operator in the adaptation process.

We also presented the architecture of our proposed solution and how we intend on evaluating it.

Acknowledgments We are grateful to Professor Antónia Lopes, Professor Paolo Romano and Richard Gil Martinez for fruitful discussions and comments during the preparation of this report.

References

1. Horn, P.: Autonomic computing: Ibm's perspective on the state of information technology. (2001)
2. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. Computer **36**(1) (2003) 41–50
3. Huebscher, M.C., McCann, J.A.: A survey of autonomic computing-degrees, models, and applications. ACM Computing Surveys (CSUR) **40**(3) (2008) 7
4. Garlan, D., Cheng, S.W., Huang, A.C., Schmerl, B., Steenkiste, P.: Rainbow: Architecture-based self-adaptation with reusable infrastructure. Computer **37**(10) (2004) 46–54

5. Esfahani, N., Malek, S.: Uncertainty in self-adaptive software systems. In: Software Engineering for Self-Adaptive Systems II. Springer (2013) 214–238
6. Cheng, S.W., Garlan, D.: Stitch: A language for architecture-based self-adaptation. Journal of Systems and Software **85**(12) (2012) 2860–2875
7. Cámara, J., Lopes, A., Garlan, D., Schmerl, B.: Impact models for architecture-based self-adaptive systems. In: Proceedings of the 11th International Symposium on Formal Aspects of Component Software (FACS2014)
8. De Lemos, R., Giese, H., Müller, H.A., Shaw, M., Andersson, J., Litoiu, M., Schmerl, B., Tamura, G., Villegas, N.M., Vogel, T., et al.: Software engineering for self-adaptive systems: A second research roadmap. In: Software Engineering for Self-Adaptive Systems II. Springer (2013) 1–32
9. Salehie, M., Tahvildari, L.: Self-adaptive software: Landscape and research challenges. ACM Transactions on Autonomous and Adaptive Systems (TAAS) **4**(2) (2009) 14
10. Brun, Y., Serugendo, G.D.M., Gacek, C., Giese, H., Kienle, H., Litoiu, M., Müller, H., Pezzè, M., Shaw, M.: Engineering self-adaptive systems through feedback loops. In: Software engineering for self-adaptive systems. Springer (2009) 48–70
11. : An architectural blueprint for autonomic computing. Technical report, IBM (June 2005)
12. Cheng, B.H., de Lemos, R., Giese, H., Inverardi, P., Magee, J.: Software engineering for self-adaptive systems: A research roadmap
13. Maggio, M., Hoffmann, H., Santambrogio, M.D., Agarwal, A., Leva, A.: A comparison of autonomic decision making techniques. (2011)
14. Gil, R.: Automated planning for self-adaptive systems. In: Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on. Volume 2., IEEE (2015) 839–842
15. Mitchell, T.M.: Machine Learning. 1 edn. McGraw-Hill, Inc., New York, NY, USA (1997)
16. Zadeh, L.A.: Fuzzy sets. Information and control **8**(3) (1965) 338–353
17. Guillaume, S.: Designing fuzzy inference systems from data: an interpretability-oriented review. Fuzzy Systems, IEEE Transactions on **9**(3) (2001) 426–443
18. Xu, J., Zhao, M., Fortes, J., Carpenter, R., Yousif, M.: Autonomic resource management in virtualized data centers using fuzzy logic-based approaches. Cluster Computing **11**(3) (2008) 213–227
19. Chiu, S.L.: Fuzzy model identification based on cluster estimation. Journal of intelligent and Fuzzy systems **2**(3) (1994) 267–278
20. Takagi, T., Sugeno, M.: Fuzzy identification of systems and its applications to modeling and control. Systems, Man and Cybernetics, IEEE Transactions on (1) (1985) 116–132
21. Jang, J.S.R.: Anfis: adaptive-network-based fuzzy inference system. Systems, Man and Cybernetics, IEEE Transactions on **23**(3) (1993) 665–685
22. Lama, P., Zhou, X.: Autonomic provisioning with self-adaptive neural fuzzy control for end-to-end delay guarantee. In: Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2010 IEEE International Symposium on, IEEE (2010) 151–160
23. Duan, S., Thummala, V., Babu, S.: Tuning database configuration parameters with ituned. Proceedings of the VLDB Endowment **2**(1) (2009) 1246–1257
24. Elkhodary, A., Esfahani, N., Malek, S.: Fusion: a framework for engineering self-tuning self-adaptive software systems. In: Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering, ACM (2010) 7–16

25. Didona, D., Romano, P.: Using analytical models to bootstrap machine learning performance predictors. In: IEEE International Conference on Parallel and Distributed Systems (ICPADS). (2015)
26. Sutton, R.S., Barto, A.G.: Reinforcement learning: An introduction. Volume 1. MIT press Cambridge (1998)
27. Auer, P.: Using confidence bounds for exploitation-exploration trade-offs. The Journal of Machine Learning Research **3** (2003) 397–422
28. Tesauro, G., Jong, N.K., Das, R., Bennani, M.N.: A hybrid reinforcement learning approach to autonomic resource allocation. In: Autonomic Computing, 2006. ICAC'06. IEEE International Conference on, IEEE (2006) 65–73