Using Machine Learning to Revise Adaptation Models Under Non-Determinism

(extended abstract of the MSc dissertation)

Francisco Duarte
Departamento de Engenharia Informática
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. On the one hand, different adaptation models have been proposed as a mean to capture, in an intelligible way, the valuable knowledge that experts have about the system behavior and how to manage it. However, expert-defined models are typically incomplete, often inaccurate and hard to keep up-to-date as the system evolves. On the other hand, the use of machine learning (ML) has been proposed to find, in a fully automatic manner, the correct adaptation strategies. However, it is not trivial for ML to cope with non-determinism, in particular with scenarios where a given adaptation may have different outcomes due to factors that have not been taken into account in the original model. In this dissertation we present RAMUN, an approach that aims at combining the advantages of static models 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 update the adaptation models at runtime. The revision process is built to take non-determinism into account. The approach has been experimentally validated in a system that performs elastic scaling of RUBiS, a prototype of an auction web application.**

I. INTRODUCTION

As computer systems become more complex, they also become harder to manage by human operators. Current systems are composed by many components, each of these with multiple deployment and configuration options. Furthermore, these systems operate in dynamic environments where the workloads are subject to change, faults occur, and components need to be frequently updated. Coping with such a dynamism requires frequent system adaptations, which makes the task of administering 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 prompter and more accurate reactions 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. The MAPE-K control loop [1] is a well known approach to build self-adaptive systems, which consists in: Monitoring the system and its context, Analyzing the data, Planning and Executing changes to the managed system based on available Knowledge about it and its context. We will focus on the Knowledge component, in particular, in learning an adaptation model to support the planning step.

The decision process embodied in self-adaptive systems involves comparing alternative adaptations at runtime [2], [3], [4], [5], [6]. Therefore, it requires an adaptation model, which supports the process by either mapping states to actions or by predicting actions' impacts.

Furthermore, self-adaptive systems are subject to non-determinism [7]. Here we will consider the non-determinism associated with the adaptations actions' effects, e.g., activating a server may not have the expected result, as it may be deployed in an already occupied physical machine. The information related with this kind of non-determinism may be useful in some cases, e.g., when it is necessary to plan for the worst possible scenario. In order to reason about the effects of non-determinism, the model should explicitly embed information regarding the non-determinism associated with adaptation actions. We argue that a highly desirable property of such models is their intelligibility for human operators. This facilitates human involvement, which we deem essential in order to build trust in the system and its ability to adapt correctly [8], [5].

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 techniques that rely on adaptation models specified by human operators [3], [4] and the use of fully automated techniques based on machine learning [9], [10], [11].

The first approach bypasses a training phase by capitalizing on expert's knowledge and the model's specifications are flexible. However, the resulting models are often incomplete and inaccurate [12]. On the other hand, automated techniques can keep the model updated based on its history at the cost of the model's representation flexibility and a training phase.

To the full extent of our knowledge there is no previous approach that combines the following key characteristics: i) leverages the knowledge an expert has about a system; ii) uses the history of the system to update the model, achieving higher accuracy on the prediction of the system's behavior; iii) provides an intelligible model and explicitly represents non-determinism to facilitate the human involvement;

In this thesis we focus specifically in modeling the impact of adaptations on the system's properties, which is a key element used by many adaptive systems to determine their behavior. The aim is to combine the advantages of adaptation models and machine learning as complementary techniques to drive the dynamic adaptation of systems.

This dissertation proposes a novel approach to revise adaptation models under non-determinism, or RAMUN for short, that works as follows. An impact model is collected from experts. From these rules a dataset of synthetic samples that represent the conditions and effects of the actions is created. This set is then extended with samples collected by monitoring the effects of adaptations on the system in operation. The extended dataset – containing both samples synthesized from the human provided impact model and gathered from the actual system – is then analyzed via the k-plane algorithm [13]. This information is then processed to extract a piece-wise linear model, which is then translated into an impact model written to a text file, using the language proposed by Cámara et al. [4]. This text file can be read by an operator or be used by an adaptation system to predict action's impacts.

RAMUN has been experimentally validated in a system that performs elastic scaling of a web application implemented using RUBiS. After collecting data from this system, we proceeded to compare our solution to model the *impact* of a concrete adaptation action (in this case, activating a server) with a state of the art, regression tree based learner [14], which does not explicitly consider non-determinism.

II. EXAMPLE

In order to illustrate the key concepts and techniques involved in the design of RAMUN, we will use as an example a web application, where the service provider has the goal of keeping a low response latency, while reducing costs related to the number of active servers. An active server is a Virtual Machine (VM) deployed in a data center that is able to receive and respond to clients' requests. We assume a standard setup, where all requests are made to a proxy, which then distributes them among a pool of active servers.

In this application, the adaptation actions that are available are the following: i) connecting a server, which aims at decreasing the response time experienced by users (by distributing the load among more servers), but also increasing the system's operational cost and; ii) disconnecting a server, decreasing the operational costs, but, depending on the load, potentially increasing the response time observed by end users.

To simplify the presentation, at this stage we will only consider the average response time experienced by users. We denote the observed average response time before an adaptation simply as *rsp* and the average response time after an adaptation as *rsp'*.

We assume that when a new server is launched its assigned VM may have 1 or 2 CPU cores, according to the data center current capabilities.

When a new server is spawned or deactivated, the adaptation system does not know how many cores it had assigned. Which means, there is no way of knowing what impact each action will have on the average response time given that a server with two cores will be able to deal with more requests, than the alternative.

III. RELATED WORK

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, which can be used to build an adaptation model. In such a situation (autonomous adaptation) 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 [15], [16].

In the literature, we are able to find different approaches to this problem. In particular, approaches that map states to actions [3], [10], e.g., if the average response time is between 300 and 400ms launch a new server, and others that map state and action to an impact on the system [4], [9], [11], e.g., if the average response time is between 300 and 400ms when a new server is launched, it is expected that the average response time lowers to 200ms.

When using the former approach, the adaptation model is tightly linked to the business' goals, which can change at any moment, e.g., a service provider may decide to reduce its target average response time. Consequently, any change leads to the invalidation of the current model and the creation of a new one. For example, if the target response time lowers, connecting a new server when the average response time is between 300 and 400ms is probably not enough to reach the new goal. The latter approach, mapping states to impacts, is decoupled from the business' goals. Therefore, it is more robust to changes at that level. Thus, we will focus on this approach that learns impact models.

Considering our example and that we use impact models to support the planning process, whenever the adaptation process is triggered, the adaptation action is chosen based on the available actions at the time and their expected impact on the system, in our example, on the average response time and operational cost. However, an action may have more than one expected impact, given that the number of cores is unknown to the adaptation system. Considering we could identify the impact of both cases and their probability we will describe three different adaptation policies we could use for our example's case, which we will use in the rest of this section as examples:

1) Combine predicted impacts and use their average (weighted by their probabilities) as the expected result;

2) Use each different expected impact as input to an utility function, which returns a value based on the system's operational cost and the average response time, and then combine the results with a weighted average of every utility value;

3) Consider the worst case scenario, i.e., impact with lowest utility for each action, if the system does not achieve its goals for a certain period of time. Thus, planning pessimistically. Otherwise, use one of the polices described previously.

In order to support the policies described, namely the last one, the impact model has to explicitly consider non-determinism. Otherwise, it is not possible to consider different possibilities, in particular the worst case.

The impact model should be: i) accurate, in order to avoid performing an unfit adaptation action that increases the system's cost instead of decreasing; ii) readable and easily understandable, to allow human operators to get involved, thus increasing their trust in the model; iii) able to explicitly consider and represent non-determinism, by capturing more than one possible impact for a given action, which allow a wider variety of policies to be used.

An impact model may be defined by an expert, or automatically learnt by a tool. In the first case, it is possible to leverage the expert's knowledge to build an impact model, thereby bypassing a learning phase. Additionally, the model's structure has no restrictions, other than, it needs to be computationally parsable, which allows to explicitly represent non-determinism in the impact model. The model could be written in a language such as the one presented by Cámara et al. [4], which is easily interpretable by a human operator and allows the explicit representation of non-determinism. With this approach any of the example's policies could be used. However, expert's knowledge is not enough to create an accurate impact model. The model is static, which means it will be outdated if the system or environment evolves, which is usual in real world scenarios. However, even if both the environment and the system were immutable, the expert's knowledge may not suffice to describe a complex system, its behavior and interactions with the environment.

Learning a model with the assistance of learning tools, on the other hand, will lead to more accurate models, as it observes the real behavior of the system at runtime, and it allows to update the model by observing new information. However, this technique needs a period of training, in which the impact model would not correctly represent the system's behavior and that could lead to costly decisions in the adaptation process. In the literature, it is possible to find a great variety of learning tools capable of learning accurate models, e.g., Artificial Neural Networks or Fuzzy Inference Systems. However, these usually output their results in the form of structures that are not easily translatable into a human readable model, or do not provide enough information to an operator. Other works, avoid this problem by using machine learning tools that employ or are translatable to human readable models, such as Model Trees (Regression Decision Trees). However, none of these techniques explicitly considers or represents the non-determinism present in self-adaptive systems, which means that their predictions are an implicit combination of every possible outcome. Therefore, these tools could provide a model that already combines the expected impacts, which is compatible with the first policy. If instead of using these tools to predict the impact (average response time), they were used to predict the utility after the action's execution, according to a given function, then it would also be compatible with the second policy. However, if the utility function changes, then the model is no longer valid. Lastly, the third policy cannot be used because non-determinism is not explicitly considered.

Sykes et al. [6] presented an alternative approach that is able to consider the actions' impacts to be non-deterministic. However, this work and others that learn probabilistic models [17], assume that the state space is discrete, which is not applicable if the goal is to predict the value of a continuous variable, such as the average response time.

In order to automatically infer an impact model able to fulfill the requirements listed before we chose to use subspace clustering, namely, k-plane [13]. This is a data mining tool, which returns the K (hyper-)planes that best represent a set of points in a multi-dimensional space. These planes may overlap in the input space, which means that a single input region may correspond to more than one plane. We will map each plane to a linear impact function, which may facilitate the comprehension of the model by operators. However, applying k-plane to this problem is not enough. The result would be a set of impact functions with endless input spaces. However, we want to write a set of rules, in which each impact function has a valid input space and a probability of happening. Furthermore, to translate the result to the language presented by Cámara et al. [4] we need to assign impact functions to input regions – one input region has one or more impact functions. Therefore, we have to identify these regions where more than one impact function is valid.

IV. RAMUN

RAMUN's purpose is to learn an impact model based on observations from the real system. This has the potential to improve the model's accuracy but also to update the model in face of upgrades to the managed system (for instance, when a machine is replaced by a new model, the impacts of adaptations need to be updated). The update process is continually repeated, in order to ensure that an accurate and updated model is being used by the adaptation system.

A different, separate, model is created and learnt for each different adaptation action. Therefore, while in the exposition we address only the action of adding a new server, a similar procedure can be executed for other adaptation actions, like, deactivating a server.

Although this technique may be used without an initial impact model, we will consider one was provided beforehand, by an expert or derived from experiments on previous deployments of the system. Based on this initial impact model the first step is to create a dataset composed of synthetic samples, which will be used to bootstrap the learning process.

After that initialization process the system begins a loop, which will result in an updated impact model at the end of each iteration. At the beginning of each cycle new samples, resulting from the monitoring of the managed system, are added to the dataset. The extended dataset is then provided to k-plane algorithm in order to update the impact functions. However, note that the model can be updated in different aspects, namely: 1) the linear impact functions corresponding to each possible outcome; 2) the input space in which each linear function is considered to be valid; 3) the probability associated with each outcome and 4) new outcomes, not captured in the initial model, may be identified and added to the impact model. Therefore, k-plane by itself is not enough to update the entire impact model.

In the following sections each step of our technique will be described in detail.

A. Initialization

Before RAMUN starts executing its model refinement loop, it creates an initial dataset composed of synthetic samples that are generated based on impact functions provided in the original model. The purpose of creating the synthetic dataset is to ensure that the adaptation system is still able to predict, with some accuracy, the impact of actions while the new model is being learnt. This way, even if not enough samples for a given outcome are observed in the operational system during the first iterations of the refinement loop, the knowledge captured in the original model is preserved and taken into account. As it was mentioned before, this step can be skipped, at the cost of taking longer to learn a valid model and the possibility of performing "bad" adaptations on the system meanwhile. Furthermore, we assume that the initial model has the same information as the one that is provided by the models presented by Cámara et al. [4]. In particular, impact functions, with their respective probabilities of happening and the input region where they are valid.

This procedure should take into account the following information, which is given in the original model, and should not be lost in the process, namely: 1) the functions that represent the impact of performing an action in a given state, 2) the probabilities associated to each impact function and 3) the ranges of values where each function is valid.

As noted before, in our example the input and output we are considering is the average response time before and after the adaptation action, respectively. Thus, the points in the dataset are tuples that include both values. To generate the synthetic samples it uses the impact functions present in the original impact model. Since the function f_i associated with an impact i defines that a (possible) effect of the adaptation action over a given system property is defined by $output = f_i(input)$, a synthetic tuple will be formed as $\langle input, f_i(input) \rangle$.

To preserve the information provided by the probabilities associated with each impact function, the number of samples generated for each function is proportional to its probability. This is done by generating $P_i * C$ samples per impact function f_i, where P_i is the probability associated with impact i and C is a constant, a positive number large enough to ensure the creation of multiple samples for each possible impact.

Our approach requires that the interval of valid input ranges for each of the dimensions is defined for every impact function. Using this valid input space i, when generating samples for impact function i, RAMUN uniformly samples the region, i.e., it ensures that the samples considered for each impact function are all equally spaced.

B. Sample Collection

The first step of the proposed loop is to collect new samples to extend the existing dataset. In order to do so, it is necessary to create samples, based on the monitoring of the managed system and add them to the dataset.

A sample is a tuple that represents the state of the managed system before the execution of a given adaptation action and the effect that action had on the target property. In our example, the only property of the system's state is the average response time experienced by users, as is the case for the target property. Which means that a tuple for this system would be $\langle rsp, rsp' \rangle$.

Therefore, a set of system's properties, deemed relevant for the adaptation's planning process, is continually monitored. The values each property takes, at any given point, represents the system's current state. For instance, consider that during the system operation a new server is activated and the observed response time decreases from $70ms$ to $50ms$, then a new sample $\langle 70, 50 \rangle$ would be created and added to the dataset associated with the "enlist server" adaptation action.

This step should be repeated until a certain amount of new samples is collected (e.g., 10% of the current dataset size or 100 samples). Note that while we consider the number of new samples to be what triggers another iteration of the refinement loop, other conditions may be also be used. For instance, one can keep collecting samples until the prediction error of the model provided by the original samples becomes too high (above some predefined threshold). The result of adding new samples to the dataset, considering our running example, is represented by the dots in in Figure 1.

C. Inferring Impact Functions

This step should find a set of impact functions that best represents the dataset collected so far. For this task we will use the k-plane algorithm [13] as a building block, which returns a set of K planes for a given dataset. We consider each plane to correspond to a linear impact function. Note however, that there is no way to know *a priori* how many impact functions the refinement procedure is expected to

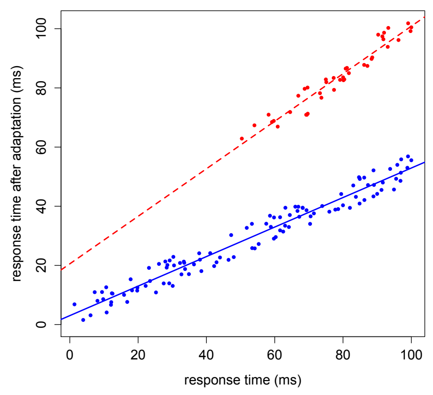


Figure 1: Dataset and impact functions.

find. In fact, as we have pointed out before, one of the purposes of the refinement algorithm is to unveil new impacts that have not been considered in the original model. Thus, we cannot derive K deterministically from the original model.

Our goal is to find the smallest possible K that captures all the the relevant impacts without cluttering the model with redundant functions. For this purpose, we iteratively run the k-plane algorithm for different values of K, starting with $K = 1$, and increasing K by one at each iteration. In order to assess when the loop should stop, we first split the dataset, which results in a training subset (90% of original dataset) and a test subset (10%). The k-plane algorithm is run against the training subset and validated against the test subset. The test subset is used to compute the model's expected fit error, by measuring the average error between each sample in the subset and its closest plane. For each sample the error is computed as:

$$\text{Error} = \frac{abs(\text{real} - \text{predicted})}{\text{predicted}}$$

And we consider the expected error for new samples to be the average value of every computed error. This process is repeated 10 times, each with a different 10% of the dataset as test set and the error is an average of the 10 runs. The iterative loop stops when the computed error is lower than a given threshold (T), thereby finding an adequate value of K.

After K is chosen, the k-plane algorithm is applied again to the entire dataset, to obtain a result that is as much accurate as possible, given the available samples. The result is the set of planes, which represent different impact functions. Following the previous example, the result of this step is represented in Figure 1.

D. Compute the Validity Ranges

As a result of the previous step, we derive K planes, that capture K different impact functions for a given adaptation. However, we cannot assume that each plane is valid in the entire input space. Therefore, we have to define what constitutes as a valid input region, which we consider to be the region that includes the samples used to infer the impact function. For instance, when looking at the results presented in Figure 1, two impact functions are depicted, however, for one of these functions, samples only exist for the interval $]50, 100]$ of the pre-adaptation response time. Therefore, there is no evidence that this impact function applies when the adaptation is performed in scenarios where the response time is outside this interval. Following this example, the goal of this step is to capture that the adaptation may behave differently when the response time is in the intervals $[0, 50]$ and $]50, 100]$.

For each impact function, RAMUN will use the samples used to infer the function, to compute the range for each input dimension. Considering a single dimension at a time, the range will be between the minimum and maximum value taken by samples, which belong to that function. After this step we will have a range for each dimension, which defines the valid region for an impact function – a hypercube surrounding the function.

However, we want to explicitly consider non-determinism, which means the hypercubes, computed previously, may overlap. Non-determinism is only present if the input space overlaps, otherwise it would be possible to distinguish them by, at least, an input variable. In order to represent the non-determinism we have to split the input regions such that it is possible to know what are the functions that are valid in a given input region.

Splitting the ranges in a single dimension is done by joining and ordering the region's limits of every plane in a list, removing duplicated values. In our example this would result in $[0, 50, 100]$. Then, we pair every two samples, which results in $(0, 50)$ and $(50, 100)$.

For higher dimensions, hypercubes only intersect if the ranges of every dimension intersect. As an example for the case of considering higher dimensions (2), we will consider two impact functions that are valid in $(0-100, 0-100)$ and $(50-150, 50-150)$, respectively.

In order to split higher dimension hypercubes we have every hypercube definition, e.g., $(0-100, 0-100)$, in a list. Then, considering one dimension at a time, we select the list's first element and check if the hypercube intersects with another one on the list and if it has different ranges for that dimension. If the ranges are equal for that dimension there is no point in splitting them. However, if two intersect and are different both items are removed from the list. Then, considering only the range for the dimension under consideration, we split it as we did for the single dimension example and combine the result with the remaining ranges of each item. For our example, and considering that the current dimension being considered is the first one, the result of the split would be: $(0-50, 0-100)$, $(50-100, 0-100)$, $(50-100, 50-150)$ and $(100-150, 50-150)$. Lastly, we add every new "hypercube" to the list's end, excluding duplicates and repeat until no hypercube intersects or if

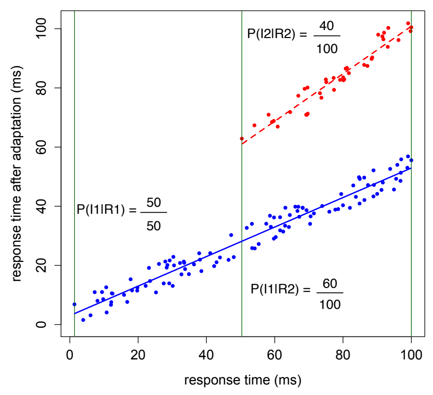


Figure 2: Probability and Valid Regions

it does intersect it has the same range for the current dimension, e.g., $(50-100, 0-100)$ and $(50-100, 50-150)$. Then, we repeat the process for the next dimension.

In the case of our running example the result would be the vertical lines in Figure 2.

E. Compute Impact's Probability

Now that the impact functions have been inferred and their valid input regions have been identified, what remains is to identify in which regions we can observe the presence of non-determinism, i.e., different impact functions observed in the same region, and to derive how likely it is to observe each of the possible adaptation's impacts, i.e., the probabilities associated with each impact function.

For each region, derived from the previous step, we will identify every sample that belongs to it. Then, using these samples, we identify which impact functions are valid within the region. The presence of more than one impact function in the same region indicates that when the input values belong to this region, then, there is no way of discerning which impact function predicts the correct impact, i.e., the result is non-deterministic. When such case arises we will compute the likelihood of each impact function being the correct one.

These probabilities are computed using the following formula:

$$P(\text{impact}_i|\text{region}) = \frac{\#(\text{region}, \text{impact}_i)}{\#(\text{region})}$$

This computes the probability of the impact of a given action being the one predicted by impact function i (i.e., impact$_i$) if the input value is within region.

The probability of each impact function in a given region is computed by counting the number of samples that belong to that region (denoted by $\#(\text{region})$) and the number of samples in that region that are closer to the i^{th} impact function (denoted by $\#(\text{region}, \text{impact}_i)$). This is illustrated in Figure 2, where the corresponding fraction is depicted next to each plane.

F. Dataset Curation

Assuming the system is continually working for an indefinite period of time, it is not reasonable to save every sample collected or generated since the system started. Keeping all samples would eventually result in a large memory occupation, and also, every sample would have the same relevance, which only makes sense if the system and environment do not change over time, which we assume to be rarely true. Also, the synthetic samples are generated based on previous knowledge of the behavior of the system that is most likely incomplete or wrong. Therefore, the final step of each iteration is to perform dataset curation. Alternatively, this could be done outside the loop, periodically.

One possible strategy to manage the dataset consists in just accumulating all points collected, resulting in the disadvantages mentioned earlier; eventually the old or synthetic points used to bootstrap the system will become a minority and have a negligible impact on the model. However, it is also possible to eliminate synthetic points as new ones are added to the dataset. This last strategy is used, for instance, by Didona & Romano [12], although their work does not address non-determinism explicitly. They assume that synthetic points are less accurate than observed points. Finally, we assume a possible strategy to be that of assigning a decaying weight to each sample. However, this method could have an impact on the discovery of uncommon events, as well as the probability computation.

Our approach therefore considers the existence of a *dataset curation* phase, at the end of each iteration. This procedure consists of eliminating or downgrading samples from the dataset that are no longer considered relevant. The policy to select this points is orthogonal to the contributions of this paper. In fact, how fast synthetic points are replaced by new points depends on the estimated accuracy of the original model which, in turn, depends on the techniques that have been used to create such model, something that is completely outside the control of RAMUN. Thus, RAMUN is agnostic to the policy used to perform the dataset curation. In all our experiments, reported in the evaluation section, we simply accumulate all points.

V. EVALUATION

For the evaluation we have experimented with a concrete instantiation of the abstract problem presented in Section II. Namely, we have applied RAMUN to learn the impact model used to support the elastic scaling of a RUBiS [1] deployment. RUBiS is a well known auction website similar to eBay.

A. Experimental Testbed

We have used RUBiS version 1.4.3 deployed on a virtual machine with 512MB of RAM, running Ubuntu 14.04 in a cluster of workstations each with a 2.13GHz Quad-Core Intel(R) Xeon(R) processor and 32GB of RAM, connected by a private Gigabit Ethernet. We used Autobench [2] to

[1]Rice University Bidding System: http://rubis.ow2.org
[2]Autobench: http://www.xenoclast.org/autobench/

generate different workloads and drive httperf [18] to issue the requests. To distribute the load among servers, we have used HAProxy 1.6 [3] running on a separate virtual machine.

As in our abstract example, the adaptation action that needs to be modeled is the activation of a server. Naturally, the real deployment is slightly more complex than the simplified example used before. In the deployed system we have monitored the following system metrics that are used as inputs in the impact model: number of active servers, request rate and response time. As in our example, the model estimates the impact of the adaptations in the observed average response time.

We will consider that our VMs are launched in a data center. However, this data center is shared by different service providers. According to Liu [19], GoGrid's [4] smallest VM has half a CPU core guaranteed when launched. However, if no other VM is using the other half, the first one uses the entire core. Following this scenario, in our evaluation we consider that each VM has one CPU core guaranteed. However, and following the GoGrid's example, if there is another available core (unused by other VMs) the VM uses both cores. We assume that, when a server is activated, the configuration is selected by the cloud provider and is outside the control of the elastic RUBiS application. Therefore, even if the probability of activation of each of the configurations may be known by the cloud provider, it must be treated as a non-deterministic event when modeling the system. For the experiments we considered that the service provider had a lot of available cores and the probability of activating a server with only 1 CPU core was 40%. Furthermore, we consider that the number of cores is chosen when launching the VM and is not changed afterwards.

B. Data Collection

All the experiments and comparisons provided in the evaluation have used the following methodology. We have collected experimental data, using the deployment described in the paragraphs above, under different configurations, by changing both the number of servers, between 1 and 4, and the workload to which the system is subject, a request rate between 250 and 10000 requests/second, in steps of 50. For each configuration we used Autobench to generate the different workloads, distributed among four client machines, and measure the system's response time. For each combination of configuration and workload we did 10000 requests, in five separate runs. The resulting response time for each run is the average between the 10000 requests and the value used for our dataset was the average between the five runs. Unfortunately, the cluster that we have used to run the experiments is a shared facility, and the collected results are slightly affected by other experiments that run concurrently on the cluster, introducing some amount of variance that was hard to control or reproduce. To amortize this unintended variance, we have executed five different runs of each experiment and we have used the average between the five runs as the final value to include in the dataset.

The collected samples correspond to connecting a new server, without knowing if this server has 1 or 2 CPU cores. In our experiments the servers that were connected before always had 2 CPU cores, in order to lower the number of cases, thus facilitating the results' presentation. Furthermore, samples which had a response time higher than 600ms were removed because we consider that a real system would not collect those samples as it would have to be adapted before reaching that point. Finally, our dataset had a total size of 688 samples.

Then, the collected data has been used to feed RAMUN. Furthermore, when comparing the system with competing approaches, exactly the same dataset has been provided to the different tools such that the differences in outcomes are entirely due to the artifacts of each approach (and not to fluctuations in the data collected).

C. Noise

There are a number of parameters that are critical to the performance of RAMUN. One of the key parameters of the approach is the error threshold (T) used to decide when to stop using additional planes in the model (i.e., to select the value of K). Roughly speaking, this parameter establishes with how many planes the learnt model is "accurate enough".

Obviously, the accuracy of the model also depends on the accuracy of the measurements used to create the dataset. Unfortunately, when observing physical phenomena, it is often impossible to obtain completely accurate measurements [20]. There is always an error, many times known beforehand. We now perform an experiment that allows us to assess the consequence of the measurement's error used to populate the dataset in the accuracy of the model and, in particular, in the selection of the stopping threshold (T). It is important to note that if RAMUN aims at an accuracy that cannot be reached with the given dataset and current value of K, it can falsely detect non-determinism where there is only noise.

All data used in this section has been obtained experimentally, therefore, is already subject to some reading error. To stress RAMUN, we have used our experimental data to create "polluted" datasets, where an additional error was added. We have created polluted datasets adding a random error to our measured values. To add a synthetic error we used random generation for the normal distribution, with mean equal to the original value, and standard deviation equal to half the original value multiplied by the intended error. Therefore, guaranteeing that the added error, with probability of 95% will be between the intended error and its symmetric. The intended errors used were 5%, 10%, 15%, 20%, 30%, 35%, 40%, 45% and 50% All the polluted datasets have been created from datasets where just 2 CPU cores servers have been activated and the request rate was between 400 and 10000 requests per second. Therefore, for the "unpolluted"

[3]Haproxy: the reliable, high performance tcp/http load balance: http://www.haproxy.org/

[4]GoGrid: https://my.gogrid.com/

Table I: Number of impact functions found

T\Error	0	5	10	15	20	25	30	35	40	45	50
0.2	2,3	3	3,4	3,4	3,4	3,4	4,5	3,4	4,5	5,6,7	6,7
0.25	2,3	2,3	3	3,4	3,4	3,4	3,4	2,3	3	3,4	3,4
0.3	1	1	1	1	1	1	1	2,3	2,3	3	3
0.4	1	1	1	1	1	1	1	1	1	1	1

Listing 1: Simplified representation

```
impact model(connect server):
1 < s < 3 & 250 < req < 800 & 3.655 < rsp < 595.775:
 [1] rsp’ = −34.994∗s + 0.025∗req + 0.360∗rsp + 74.506
1 < s < 3 & 800 < req < 9950 & 4.255 < rsp < 595.775:
 [0.62] rsp’ = −34.994∗s + 0.025∗req + 0.360∗rsp + 74.506
 [0.38] rsp’ = −135.084∗s + 0.014∗req + 0.488∗rsp + 502.873
1 < s < 3 & 9950 < req < 10000 & 3.655 < rsp < 595.775:
 [1] rsp’ = −135.084∗s + 0.014∗req + 0.488∗rsp + 502.873
```

dataset, there should not be detected non-determinism in the phenomena being modeled as we restricted our dataset to samples that should correspond to one impact. In Table I we tested different error thresholds with datasets that had different levels of "pollution". To serve as a baseline we also present these results with the original dataset. As our approach has a random component we ran it 10 times for each combination (dataset, threshold) and show the number of planes found. A cell with more than one number means that for that combination the number of planes was not always the same and every result is shown.

As it can be observed in Table I, when the added error starts increasing, the number of impact functions found also increases. Furthermore, if the error threshold is small (0.2, 0.25) then even the "unpolluted" dataset is considered to have more than one impact. Which means that non-determinism is being found where it should not exist. However, if we try to avoid this scenario by using a high error threshold, e.g., 0.4, we might not find different impact functions when we should, because it learns a "good enough" model with one impact function.

The absolute values presented here are specific to this experiment and should not be applied to every context. However, the results of the experiment show that the error threshold should be chosen according to the system's behavior and the monitoring process. The system's behavior should be taken into account because the variance in the data collected affects the number of planes found, as we can see by our "unpolluted" dataset. The monitoring process and the error it possibly adds to the dataset should also be considered, because this will also affect the number of planes found, as can be seen by the "polluted" datasets experiments.

For the rest of the experiments, we will use an error threshold of 0.3 because it is the smallest tested threshold which finds only one impact function in the original dataset.

D. *Learnt Impact Model*

An advantage of RAMUN over competing approaches, such as model trees, is that it explicitly represents non-determinism in the learnt model. To demonstrate this feature, we used our solution with data collected from connecting a new server with 2 CPU cores and 1 CPU core.

We assume there was no impact model provided beforehand, therefore, the model was built from scratch.

In order to simplify the model's presentation, in Listing 1 we present a model equivalent to the one proposed by Cámara et al. [4] in a simpler representation, which only presents information relevant for this dissertation. In particular, the simplified representation does not represent effects on the system, such as the number of active servers increasing by one. It does not represent entities such as clients or servers because we are only considering the impact on the client's perceived response time. Lastly, the impact functions are represented inline.

In this Listing, *s* corresponds to the number of active servers connected before the action, *req* to the number of requests per second on average, *rsp* and *rsp'* to the average response time before and after the action was executed, respectively. This model shows that this approach is able to learn an impact model based on real data while explicitly accounting for non-determinism. Furthermore, the probabilities follow very closely what we expected to find as, approximately, 40% of the dataset corresponds to connecting servers with only 1 CPU core.

E. *Fitting with Non-Determinism*

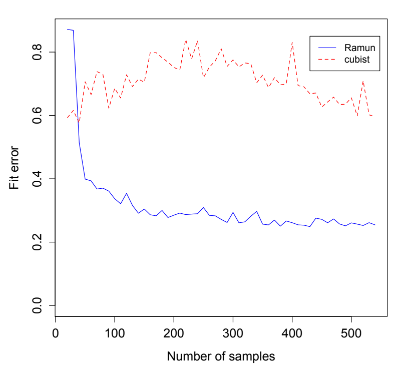
By explicitly considering non-determinism we expect that the model learnt using RAMUN will fit better a dataset, where non-determinism is present, than a tool which does not consider it. As in the previous section, non-determinism is present because the server that is activated may have 1 or 2 CPU cores and the systems does not know which one it will get. To test this assumption we compare RAMUN with *cubist*. We will test how each fits the dataset by dividing the dataset in 10 subsets and using cross validation. Furthermore, we will use different sizes of training subsets, sampled from the entire training set ($9/10$ of entire set), in order to present how fast they stabilize in a good model.

In Figure 3a we can observe that in this dataset, where non-determinism is present, a solution like cubist, which does not explicitly account for non-determinism, does not fit the dataset as well as RAMUN. We can also observe that when we increase the number of samples used to learn the model, the fitting error actually increases when using cubist. We assume this happens because the increasing dataset is skewing the model to something in between both impacts, therefore increasing its distance to more samples.

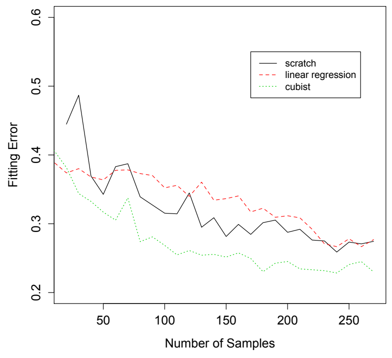
F. *Initial Model's Effect on Learning Phase*

The initial model chosen to support the system during the learning phase will impact the learning process. If the initial impact model does not model the system's behavior correctly, this may hinder the early impact models, and delay the model's convergence.

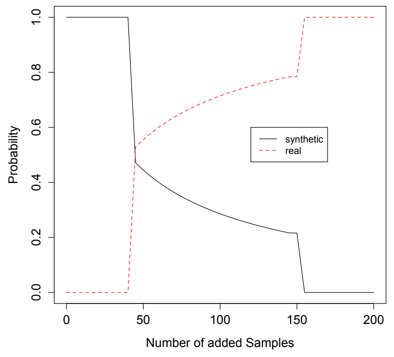
In order to show the effect of the initial impact models' correctness we will use our solution to learn a new impact



(a) Fit comparison



(b) Fitting with Initial Models



(c) Probability of synthetic and real impact functions

Figure 3: Evaluation Results

model, starting with different initial models. As initial models we will use one that is inferred with a linear regression, and the other will be inferred using *cubist*. To serve as a baseline we will also learn the model from scratch, as we did previously. To train these models we will use the same partial dataset, which is 50% of our dataset, randomly sampled. The remaining half will be used to learn and evaluate a model as we did previously.

As we can see in Figure 3b, using a model to initialize the dataset results in a faster convergence to a good model. However, if we use a not so good initial model, as the one given by the linear regression, the learnt model may achieve better results in the beginning, but in the long run it may be hindered because the dataset now contains samples that do not represent the system's behavior correctly.

The size of our initial dataset is of 50 samples. However, the size of the initial synthetic dataset depends on the managed system. If we use many samples as the initial dataset, then, the model will be accurate in the beginning of the managed system's lifetime (considering it is a good initial model). However, these samples will affect future models when it has enough samples collected from the system. On the other hand, if we use few samples to initialize the dataset, the model's initial accuracy may be bad but with fewer new samples it achieves a better model. Therefore, the size depends mainly of the sample collection speed. If samples are collected frequently, then we can use a smaller initial dataset because it will have enough samples soon. If the collection rarely happens then the initial dataset should be bigger.

G. *Outdated Samples*

We propose that an initial model be used to bootstrap the initial dataset. However, we also state that we do not trust this initial model's correctness. In this section, we will show why we think it is still valid to use an initial model, even if it is not correct. This experiment is particularly relevant to support the validity of skipping the data curation step presented in the previous chapter. In order to do so, we will consider the same dataset we considered for the Noise experiment, i.e., a dataset that should only detect one impact. Furthermore, we consider that our initial model populates the dataset with 40 samples derived from an impact function that is parallel to the real impact but with an added 250ms.

Our experiment will depart from the initial dataset and gradually add samples from the real dataset and measure the probability of the impact functions found. Because we are only considering two impacts we will have at most two different impact functions.

We expect that by collecting new samples that belong to a real impact, this impact function will be enough to capture the system's behavior with a low error. Therefore, the initial impact function will no longer be considered.

In Figure 3c we can see that our expectations were true for this case. However, we must note that this happened for this particular experiment, and there are some details we have to consider. First, the distance between impact functions will have a great impact on these values. In particular, if the distance is bigger, i.e., the initial model is less correct, the number of new samples needed to find a new impact function is smaller, but the number of samples needed so the initial impact function is no longer considered is bigger. Furthermore, the number of synthetic samples also influences these results. Assuming we use the same distance between impact functions, if we use more synthetic samples, finding a new impact function will take longer, as will disregarding the initial function. Lastly, this idea can also be applied to outdated samples instead of synthetic. By outdated samples we mean, samples that are no longer valid. For example, after an upgrade to a system, the existing samples may not be valid anymore.

In this experiment we used an initial impact function that was parallel to the real function. However, this does not affect our conclusions which are equally valid if the functions are not parallel.

VI. Conclusions

In this thesis we proposed Ramun, a novel approach that dynamically learns and refines an impact model, which is used to predict the impact of adaptation actions. The initial impact model may be provided by an expert to bootstrap the process. Our solution's key aspects are its ability to explicitly capture and learn non-deterministic effects associated with complex adaptation actions, and the fact that it infers impact models that can be easily interpreted by human operators, while keeping the model up to date. The first property allows for enhancing the model robustness in presence of exogenous factors that cannot be easily measured or captured in the model. The model's readability allows to keep human administrators in the loop – a property that we argue is essential to build trust in the self-adaptive system. Its ability to update the model is essential to deal with an evolving environment.

We evaluated Ramun with a well known benchmark, RUBiS, and using as baseline, a state of the art regression tree learner that does not model non-determinism associated with adaptation actions' effects.

As future work we would like to explore more efficient alternatives to the exhaustive search for the ideal value of K during the impact function inference step. Furthermore, we would explore the possibility of considering higher order functions, as impact functions, without compromising the model's readability. Finally, we would like to use our solution as support for an adaptive system's planner, such as the one presented by Gil [21].

Acknowledgments

This work was partially supported by PIDDAC and by Fundação para a Ciência e Tecnologia (FCT) through projects with references PTDC/EEI-SCR/1741/2014 (Abyss) and UID/CEC/50021/2013. Parts of this work have been performed in collaboration with other members of the Distributed Systems Group at INESC-ID, namely, Richard Gil and Professors Paolo Romano and Antónia Lopes.

References

[1] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, 2003.

[2] D. Garlan, S. W. Cheng, A. C. Huang, B. Schmerl, and P. Steenkiste, "Rainbow: Architecture-based self-adaptation with reusable infrastructure," *Computer*, vol. 37, no. 10, 2004.

[3] S. W. Cheng and D. Garlan, "Stitch: A language for architecture-based self-adaptation," *Journal of Systems and Software*, vol. 85, no. 12, 2012.

[4] J. Cámara, A. Lopes, D. Garlan, and B. Schmerl, "Adaptation impact and environment models for architecture-based self-adaptive systems," *Science of Computer Programming*, vol. 127, no. C, 2015.

[5] M. Salehie and L. Tahvildari, "Self-adaptive software," *ACM Transactions on Autonomous and Adaptive Systems*, vol. 4, no. 2, 2009.

[6] D. Sykes, D. Corapi, J. Magee, J. Kramer, A. Russo, and K. Inoue, "Learning revised models for planning in adaptive systems," in *Proceedings - International Conference on Software Engineering*, San Francisco, CA, USA, 2013.

[7] N. Esfahani and S. Malek, "Uncertainty in self-adaptive software systems," in *LNCS*, vol. 7475, 2013.

[8] B. H. C. Cheng, R. De Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic, G. Di Marzo Serugendo, S. Dustdar, A. Finkelstein, C. Gacek, K. Geihs, V. Grassi, G. Karsai, H. M. Kienle, J. Kramer, M. Litoiu, S. Malek, R. Mirandola, H. A. Muller, S. Park, M. Shaw, M. Tichy, M. Tivoli, D. Weyns, and J. Whittle, "Software Engineering for Self-Adaptive Systems: A Research Roadmap," in *Softw. Eng. Self-Adaptive Syst.*, vol. 5525, 2009.

[9] S. Duan, V. Thummala, and S. Babu, "Tuning Database Configuration Parameters with iTuned," *ReCALL*, vol. 2, 2009.

[10] J. Xu, M. Zhao, J. Fortes, R. Carpenter, and M. Yousif, "Autonomic resource management in virtualized data centers using fuzzy logic-based approaches," *Cluster Computing*, vol. 11, 2008.

[11] P. Lama and X. Zhou, "Autonomic provisioning with self-adaptive neural fuzzy control for end-to-end delay guarantee," in *Proceedings - 18th Annual IEEE/ACM International Symposium on MASCOTS*, Miami, Florida, USA, 2010.

[12] D. Didona and P. Romano, "On Bootstrapping Machine Learning Performance Predictors via Analytical Models," in *ICPADS*, 2015.

[13] P. S. Bradley and O. L. Mangasarian, "k-Plane Clustering," *Journal of Global Optimization*, vol. 16, 2000.

[14] M. Kuhn and N. Coulter, "Cubist Models For Regression," *R package Vignette R package version 0.0*, vol. 18, 2012.

[15] M. C. Huebscher and J. a. McCann, "A survey of autonomic computing-degrees, models, and applications," *ACM Computing Surveys*, vol. 40, 2008.

[16] Y. Brun, G. Di Marzo Serugendo, C. Gacek, H. Giese, H. Kienle, M. Litoiu, H. Müller, M. Pezzè, and M. Shaw, "Engineering self-adaptive systems through feedback loops," in *LNCS*, vol. 5525, 2009.

[17] A. Filieri, L. Grunske, and A. Leva, "Lightweight adaptive filtering for efficient learning and updating of probabilistic models," in *Proceedings - ICSE*, vol. 1, 2015.

[18] D. Mosberger and T. Jin, "Httperf—a Tool for Measuring Web Server Performance," *ACM SIGMETRICS Performance Evaluation Review*, vol. 26, 1998.

[19] H. Liu, "A measurement study of server utilization in public clouds," *Proceedings - IEEE 9th International Conference on DASC*, 2011.

[20] A. J. Ramirez, A. C. Jensen, and B. H. C. Cheng, "A taxonomy of uncertainty for dynamically adaptive systems," in *ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, 2012.

[21] R. Gil, "Automated planning for self-adaptive systems," in *2015 IEEE/ACM 37th IEEE ICSE*, vol. 2. IEEE, 2015.