Learning Adaptation Models Under Non-Determinism

Francisco Migual Caramelo Duarte

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. Dr. Luís Eduardo Teixeira Rodrigues

Examination Committee

Chairperson:	Prof. Dr. José Carlos Alves Pereira Monteiro
Supervisor:	Prof. Dr. Luís Eduardo Teixeira Rodrigues
Member of the Committee:	Prof. Dr. Pedro Miguel Frazão Fernandes Ferreira

October 2016

Acknowledgements

I would like to thank my advisor, Professor Luís Rodrigues for taking the time to guide and motivate me over the last year. I also would like to thank Richard Gil, Professors Paolo Romano and Antónia Lopes for our fruitful discussions and for the support they provided during the preparation of this thesis.

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.

Lisboa, October 2016
Francisco Duarte

For my family and friends,

Resumo

Entre as abordagens propostas na literatura para suportar adaptação dinâmica, é possível encontrar duas técnicas distintas para realizar a modelação do sistema. Por um lado, existem modelos que capturam o conhecimento de peritos sobre o comportamento do sistema e de como geri-lo, de forma inteligível. No entanto, os modelos estão normalmente incompletos, imprecisos e tornam-se desatualizados à medida que o sistema evolui. Por outro lado, foi proposta a utilização de aprendizagem automática para encontrar as estratégias de adaptação corretas. Porém, aprendizagem automática requer um grande conjunto de treino, de observações do sistema, normalmente recolhidas durante longos e exaustivos períodos de treino para conseguir bons resultados. Para além disso, estas últimas em geral são aplicáveis apenas a sistemas deterministas, não sendo capazes de capturar no modelo aprendido, por exemplo, que uma ação de adaptação pode ter resultados distintos devido a características que não são diretamente observáveis. Nesta dissertação apresentamos uma abordagem que combina as vantagens dos modelos estáticos e das ferramentas de aprendizagem automática, de forma a criar modelos de sistemas não-deterministas, que são simultaneamente precisos e compreensíveis, e que podem ser obtidos num curto período de tempo. A abordagem consiste em partir do conhecimento do perito para obter um modelo do comportamento do sistema e usar ferramentas de aprendizagem automática para atualizar modelos de adaptação em tempo de execução. Para além disso, este processo aprende um modelo que tem em conta não-determinismo. Esta abordagem foi experimentalmente validada num sistema que realiza o re-dimensionamento dinâmico de uma instalação do RUBiS, uma conhecida aplicação de gestão de leilões na rede.

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, ML requires a large training set of observations, usually collected from long and comprehensive training phases to provide meaningful results. Furthermore, it is not trivial for ML to capture non-determinism in the learnt model, 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 thesis we present 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 revise, refine, and update the adaptation models at run-time. 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.

Palavras Chave
Keywords

Palavras Chave

Adaptação Autonómica

Modelo de Impacto

Não-Determinismo

Aprendizagem Automática

Agrupamento de Subespaço

RUBiS

Keywords

Self-Adaptation

Impact Model

Non-Determinism

Machine Learning

Subspace Clustering

RUBiS

Contents

List of Figures

List of Tables

Acronyms

ML Machine Learning

RUBiS Rice University Bidding System

VM Virtual Machine

ms milliseconds

DTMC Discrete Time Markov Chain

CPU Central Processing Unit

1 Introduction

This work addresses the problem of updating/learning an impact model that represents the system's behavior in the context of supporting self-adaptation of a system operating on a dynamic environment. This model must be accurate and readable by human operators to ensure their trust on the model itself. Furthermore, it should account for the non-determinism inherent to self-adaptation actions, and explicitly represent it in the model. For this purpose, we propose the use of a learning technique to learn a model which is then translated to a human readable impact model, with non-determinism explicitly represented.

1.1 Motivation

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, therefore, to identify the optimal 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. 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 decision process embodied in self-adaptive systems involves comparing alternative adaptations at runtime based on the system's current state and, possibly, on the action's expected impact (Garlan, Cheng, Huang, Schmerl, and Steenkiste 2004; Cheng and Garlan 2012; Cámara, Lopes, Garlan, and Schmerl 2015; Salehie and Tahvildari 2009; Sykes, Corapi, Magee, Kramer, Russo, and Inoue 2013). Therefore, it requires an adaptation model, which supports the decision making process by either mapping states to adaptation actions or by predicting actions' impacts on the system and choose the action that leads to the system's best state.

Furthermore, self-adaptive systems are subject to non-determinism (Esfahani and Malek 2013). Here we will consider the non-determinism associated with the the adaptations actions' effects. For example, activating a server may not have the expected result, as the underlying and unobservable conditions may differ, e.g., launching a server in a physical machine, which may already be occupied by other instances. 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.

Some adaptation systems operate as black boxes, which may result in a lack of trust, by the operators, in its ability to correctly adapt systems (Cheng, De Lemos, Giese, Inverardi, Magee, Andersson, Becker, Bencomo, Brun, Cukic, Di Marzo Serugendo, Dustdar, Finkelstein, Gacek, Geihs, Grassi, Karsai, Kienle, Kramer, Litoiu, Malek, Mirandola, Muller, Park, Shaw, Tichy, Tivoli, Weyns, and Whittle 2009; De Lemos, Giese, Müller, Shaw, Andersson, Litoiu, Schmerl, Tamura, Villegas, Vogel, Weyns, Baresi, Becker, Bencomo, Brun, Cukic, Desmarais, Dustdar, Engels, Geihs, Göschka, Gorla, Grassi, Inverardi, Karsai, Kramer, Lopes, Magee, Malek, Mankovskii, Mirandola, Mylopoulos, Nierstrasz, Pezzè, Prehofer, Schäfer, Schlichting, Smith, Sousa, Tahvildari, Wong, and Wuttke 2013; Salehie and Tahvildari 2009). For this reason, it is necessary that the internal models used in the adaptation process are intelligible and available for the operator to analyze and understand.

In order to accommodate the previously described requirements the goal of this work is to design a technique to learn an impact model that is: *a)* accurate, so that it does not lead to unfit adaptations that increase the system's cost instead of decreasing; *b)* able to explicitly represent non-determinism, which provides more knowledge about the system's behavior and; *c)* readable and easily interpretable, to allow human operators to get involved thus increasing their trust in the model.

1.2 Contributions

This work addresses the automated learning of the impact models that support adaptation systems. Specifically, we aim at learning models that predict the impact an adaptation action will have on the managed system, while keeping the model readable, accurate and aware of non-determinism. In this context, this thesis makes the following contribution:

> The dissertation proposes a technique capable of learning adaptation action's effects on the system's parameters and representing them on a readable model, while explicitly considering non-determinism.

1.3 Results

The results produced by this thesis can be enumerated as follows:

- A Python implementation of a data mining tool capable of clustering data into subspaces, k-plane;
- A prototype of RAMUN, the proposed technique, using Python.
- An experimental evaluation of the prototype using a system that performs elastic scaling of a web application implemented using RUBiS.

1.4 Research History

During this thesis, we benefited from the fruitful collaboration of Professor Paolo Romano, Professor Antónia Lopes and Richard Gil.

This work is part of a collaboration among the Distributed System group at IST, the Group of Software Systems at FCUL, and David Garlan's research group at CMU. The goal is to automatically learn impact models that consider non-determinism and represent them in a human readable manner, written using the language proposed by Cámara, Lopes, Garlan, and Schmerl (2015). The impact model is then supposed to support the planning of adaptation actions, using Gil's work (Gil 2015), which will leverage the non-determinism representation in its plan generation process.

In the beginning, the main focus of this work was to study the state of the art solutions used to support self-adaptation systems. In particular, solutions that used static expert defined models and, alternatively, the ones that used machine learning to support the adaptation process. As a result of that study this work would combine both approaches therefore combining their advantages.

1.5 Structure of the Document

The rest of this document is organized as follows. For self-containment, Section 2 provides an introduction to RAMUN. Chapter 3 describes the architecture and the algorithms used by RAMUN and Chapter 4 presents the results of the experimental evaluation study. Finally, Chapter 5 concludes this document by summarizing its main points and future work.

2 Related Work

In this chapter we will survey the relevant related work found in the literature, regarding self-adaptive systems, adaptation models and learning tools. In order to support the presentation of the related work, we will start by presenting an abstract system that could benefit from self-adaptation, which will be used as an example throughout the rest of this and the next chapter. We will also present and explain some concepts and details relevant for this thesis.

2.1 Motivating Example

In order to illustrate the key concepts and techniques presented in this chapter, we will use as an example an abstract web application. The service provider responsible for this application has the goal of keeping a low response latency to its users, while reducing costs related to the number of active servers. The number of active servers affects the users' experienced response time, i.e., if a system has more connected machines serving requests, it is able to handle a larger request rate with a low average response time, due to its parallelization using the different servers. We assume a standard setup, represented in Figure 2.1, where all requests are made to a proxy, which then distributes them among a pool of active servers.

2.2 Self-Adaptive Systems

Complex computer systems which are subject to a dynamic *environment* need to adapt to their current state, by changing their configuration, parameters or available resources, i.e., by performing an

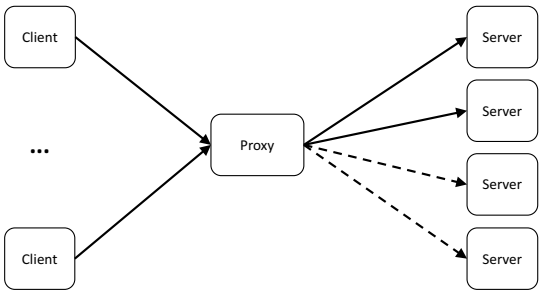

Figure 2.1: Service setup

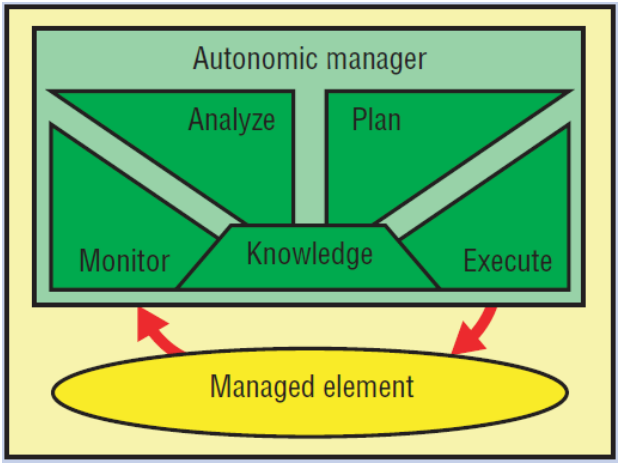

Figure 2.2: MAPE-K loop

adaptation action, in order to satisfy some high-level goals. Self-adaptation has been presented as a way of speeding the process and unburdening human operators by automating this management process.

In this context and throughout the rest of the dissertation, the term *environment* is used to name external factors, i.e., factors not pertaining to the system, but that affect the system's behavior or performance. These factors may be observable by monitoring tools, e.g., users' request rate, or unobservable, therefore, unknown to the system, e.g., network infrastructure. Although, these factors have an influence on the system, in our case the web application example, the choice of which variables should be monitored is outside the scope of this work. Ultimately, it should be the operator selecting the relevant monitored values.

Adaptation actions correspond to changes done to system's properties or configuration in order to change its behavior and achieve its goals. These actions are defined at development time and used at runtime.

In our example, the high-level goals are to provide a service with a low average response time, and maintain low operational costs. Furthermore, the adaptation actions we consider to be available in our example are the following:

a) connecting a server, which aims at decreasing the response time experienced by users (by distributing the load among more servers), but also increases the system's operational cost;

b) disconnecting an active server, decreasing the operational costs, but potentially, depending on the load, increasing the response time observed by end users.

2.2.1 MAPE-K Adaptation Model

The main building block of a self-adaptive system is the *autonomic feedback control loop* (Brun, Di Marzo Serugendo, Gacek, Giese, Kienle, Litoiu, Müller, Pezzè, and Shaw 2009). A common example of such control loop is the MAPE-K loop illustrated in Figure 2.2. The loop is composed by four

different steps, which share a Knowledge component that holds information about the system and the environment (Kephart and Chess 2003; Salehie and Tahvildari 2009; IBM 2005):

1) Monitor collects information about the system and environment, and aggregates the collected data into symptoms, if they need to be analyzed. In our example, this would correspond to monitoring the average response time to user's requests.

2) Analyze receives the data collected in the previous step, and analyzes it, to check if changes are required. In which case, it passes a request to the next step of the loop. In our example this consists of assessing if the response time is above or below the target range.

3) Plan is the step where the sequence of actions (or single action) is chosen to adapt the system to the desired state (state where goals are satisfied). The constructed plan is then passed to the next phase. A possible plan, in our case, is to activate the number of servers required to lower the average response time.

4) Execute receives as input the plan of adaptation and applies it to the managed system. Following the previous example, this step would connect the new servers.

The Knowledge component of the MAPE-K loop is responsible for storing relevant information about the system, such as actions it may perform, their impact on system's properties, when the system should be adapted, what are the goals that should be achieved, adaptation policies, and others.

This work will focus on the Knowledge component. In particular, on adaptation models that support the planning process, by mapping system and environment's states to adaptation action(s), or predicting the impact of executing a given action on the system's state, i.e., impact model.

2.2.2 Adaptation Knowledge

The knowledge component encompasses details, used by the four steps in the adaptation process, about the system's behavior, the environment and how it could be managed. It may hold information about the system's behavior, such as the interaction between different system's components, how the system behaves under different configurations and parameterizations. Considering our example, it could be important to know that more connected servers leads to lower response times and less servers to higher response times. The knowledge of the environment and how changes in it affect the system is also relevant as the system will often try to adapt to the current environment in order to achieve its predefined goals. In our example it may be relevant to know that changes in the workload, e.g., request rate, will affect the response time. It can also contain the knowledge about the system management, which holds the information to what changes can be made to the system in order to adapt it, i.e., what are the available adaptation actions. In our example these adaptation actions would correspond to connecting and disconnecting a server from the active pool.

Finally, it may have an adaptation model that is used to support the *Plan* step of the process, which is the main focus of this dissertation and we will cover in more detail in the next section.

2.2.3 Adaptation Model

In the literature it is possible to find different types of adaptation models, as well as writing specifications and techniques to build them. The model can support the adaptation process by either predicting, based on the managed system and environment's state, the new configuration the system should be in, the set of actions to adapt the system (adaptation plan) or each action's impact on the managed system. If we consider our example, the predictions could be, respectively: *a)* for the current state two servers should be active/connected; *b)* considering this state and the current system's configuration the plan is to connect two new servers; *c)* in this state, if a server is connected then the average response time will decrease 100ms. We will group the first two approaches, as both predict which changes must be performed. Therefore, we will only compare the last two approaches, i.e., it either predicts the action to be performed or the impact of performing an action.

In this dissertation *impact* will be used as the consequence of performing a given adaptation action on the system, in terms of monitored variables. The monitored variables can be related to the system or its environment. In order to simplify the example's presentation, at this stage, the only monitored variable is the average response time and the impact of the adaptations is the change in the average response time after the action is performed. We denote the observed average response time before an adaptation simply as *rsp* and the average response time after an adaptation as *rsp'*.

Therefore, we will compare both approaches to build and use adaptation models and discuss each one's advantages and disadvantages. One that receives as input the system's current state and outputs the adaptation action that should be performed to adapt the system (Condition-Action). Alternatively, we present impact models, which receive as input the system's current state and an adaptation action that can be performed, and outputs the expected impact on the system (Condition-Action-Effect).

2.2.3.1 Map to Plan

The work presented by Xu, Zhao, Fortes, Carpenter, and Yousif (2008) is an example of an adaptation system that learns a model, which maps the current system to an adaptation action. Part of their proposal is to use a learning technique to build a model, which will be used to support the process of autonomic resources allocation, by predicting the resource needs for individual virtual containers, based on the performance guarantees provided to their clients. In this particular work they used Fuzzy Inference Systems to learn the system's behavior, however, in this section we will not focus on the tool.

The learning tool builds a model based on input and output pairs. The input is the system and environment's state. The output is the amount of resources used at the time. The model is then used at run-time by providing it as input, the current state, and its output will be the necessary resources to achieve the system's goals.

Because there are system's goals and the resource allocation process is used to achieve them, input and output pairs that did not achieve these goals, i.e., the resources were insufficient for that particular state, are not considered. Therefore, the learning tool only uses valid data pairs.

In our example, we could use this solution by considering *rsp* as input and the number of active servers as output. Using request rate as input would be more appropriate for this example as the *rsp*

does not necessarily dictate the number of servers that should be active. However, the pair would only be fed to the learning tool if it achieved the system's goal. For example, if the response time (*rsp*) was below a threshold.

2.2.3.2 Map to Impact

The work proposed by Elkhodary, Esfahani, and Malek (2010) is a framework that learns an impact model, which is later used to support the planning process. Instead of learning a model which maps the current state to a plan, it learns a model that is able to predict the value of a given performance metric under certain conditions. Then, an external planner uses this information to generate the plan at run-time.

Applying this solution to our example would be similar to the previous solution. However, the output would not be the number of active servers, but the response time after activating a server, *rsp'*. Then, the planner would decide if the system needed another active server. This solution will be presented again, in more detail, further ahead.

The first solution, which learns a model that maps the current system and environment's state to an adaptation action, eliminates the need for complex planning algorithms because they already output a plan for the current state. However, because these plans are made so the managed system achieves some business goals (only considers states where the goals where achieved), the models are tightly linked to them. Therefore, if the business goals change, the model is invalidated because the plans were made to achieve a state that is no longer the goal. This is a problem if we consider that these goals are not immutable. In fact, they usually have to change as the technology evolves to keep up with the increasingly strict requirements, e.g., lower latency when accessing a website.

On the other hand, the second solution learns models which predict adaptation action's impact on the system and environment. This will add complexity to the planning process as it will have to plan how to adapt the managed system based on its properties, environment, available actions and their expected impacts. However, if the monitored and target properties do not change, these models have the advantage of being completely decoupled from the business goals. If the model predicts the impact an action has on the system and environment's properties, it will still be valid if the business goals change because the impact will not be affected by these changes. Therefore, the disconnection between the model and the business goals facilitates reuse, by an equivalent system with different business goals, and it is not invalidated as often.

We will focus our work on the second approach, i.e., impact models, because of their robustness to change. Furthermore, we can consider the added complexity as a trade-off for the extra planning flexibility (higher variety of policies, which could be used interchangeably) which tools like the one presented by Gil (2015) can take advantage of.

Impact Model and Policies

Impact models predict the impact of an adaptation action on the system's properties. In order to use this model to adapt the system we also need a planner and an adaptation policy, which will output a sequence of adaptation actions (possibly with a single action) that guides the system to a goal state. In this context, the policy defines how the planner uses the impact model.

Although, policies and planning are outside the scope of our work, we will give an example of three *policies* that state how the planner could use the impact model's predictions, as a motivation for impact models that capture non-determinism and that will be used in the next sections as examples. For these policies we assume the impact model captures non-determinism by presenting different possible impacts with their respective probability of happening. We consider the input to be the system's state before the adaptation action is performed and the adaptation action itself. The policies we will consider are:

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

b) 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;

c) 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.

Using our example, lets assume our impact model predicts that the impact of connecting a new server in a given state is the decrease of the average response time by, either 100ms, with 80% probability or 50ms, with 20%.

Using this information alongside the first policy, the planner would consider the impact of activating one server to be a decrease of 90ms ($100 * 0.8 + 50 * 0.2$) on the average response time.

The second policy requires an utility function. We will assume that the utility values for both states, decrease of 100 or 50ms, are 5 and 2, respectively. Therefore, the utility value used would be 4.4 ($5 * 0.8 + 2 * 0.2$).

An application of the last policy would be to track how much time the system did not achieve its goals, and if it exceeds some threshold, e.g., 5 seconds in a 100 seconds time window, it considers the worst case scenario for every action, which would be the decrease of only 50ms in our example. Thus, this would be useful in systems in which high response times causes the loss of users and consequently, profit. When considering the worst case scenario the adaptation system will probably connect more servers than the ones that are needed, i.e., it will over provision.

2.3 Manually Defined Models

Self-Adaptation is the process that aims at automating the manual adaptation process, i.e., changes done by human operators in order for the system to meet certain requirements. Therefore, the human operators already have a mental model of the system's behavior, based on their previous experience with it.

In the literature it is possible to find different solutions that use the manually defined model approach. In this section we present and discuss the work of Cámara, Lopes, Garlan, and Schmerl (2015),

which leverages the human operator's knowledge to build a static impact model that supports the representation of non-determinism.

2.3.1 Expert Defined Impact Models

The work done by Cámara, Lopes, Garlan, and Schmerl (2015) describes a specification of probabilistic models for architecture-based self-adaptive systems, i.e., they describe a way of writing impact models that explicitly represent non-determinism.

According to this specification the impact models are based on Discrete-time Markov Chains (DTMCs), which allows the representation of different outcomes, for a given input, with associated probabilities. In the case of impact models, the input corresponds to the system's state and an adaptation action. The output on the other hand represents the system's new state.

In particular, each adaptation action has an impact model. For each impact model there are a set of rules. Each rule has a condition, which is one of the ways the model accounts for context variability, because the impact of an action depends on its current observable context. In our example, the impact of connecting a new server is different if the number of active servers, before the action is executed, is 2 instead of 3.

Following the condition is the set of possible outcomes, each one with a given probability associated. Which means that for a given state the model considers multiple possible outcomes, therefore it explicitly considers non-determinism.

The outcome is composed by the action's effect on the system and its impact on the system's metrics. In the case of our example, and considering the action to be connecting a new server, the effect would be having one more active server, and the impact could be the average response time decrease of 50ms (*rsp'* = *rsp* $- 50$). The other way the model accounts for context variability is by representing the impact on the metrics as a function of the current state's properties.

The outcomes' probabilities represent the likelihood of verifying that a given outcome happens instead of the alternatives. Furthermore, these probabilities may be used to compute a plan's (set of adaptation actions) utility, i.e., a value that allows to compare different plans and choose the best. The chosen utility function is outside the scope of our work.

2.4 Automatically Learn Models

In order to unburden system's operators, several papers proposed automated approaches to assist in the adaptation process. The idea is to collect measurements, selected before the system's deployment, from the running system and feed them to a learning tool. Given that these tools deal directly with data collected from the system and because most of them learn complex models to fit non-linear functions, they are able to infer accurate models, if enough data is collected.

Therefore, we can use these techniques to learn impact models by providing as input system's measurements before the execution of an action, and as output the measurements after the action is

performed. These tools could then be used to predict the impact of performing an action based on the system's current state.

2.4.1 Learning Cycle

A concrete example of this approach is the work done by Elkhodary, Esfahani, and Malek (2010), where they presented FUSION. FUSION is a framework that learns the impact of an adaptation on the system's user defined goals, and uses this information to output adaptation plans. In practice this framework is a concrete implementation of the MAPE-K loop.

Their work uses machine learning tools (in their case, a model tree learner) to learn how a given state affects the system's goals during the learning phase. Then, the learnt model is used to adapt the system when needed. It is worth mentioning that in their case the system's state is given by a set of features, which are system's properties that can be enabled or disabled, e.g., send a web page with/without images in order to provide a better service or reduce latency, respectively. Furthermore, the goals are metrics collected from the system alongside an utility function, which should be maximized during the adaptation process.

The model is learnt during the *Learning Cycle*. This cycle is composed by two steps. It monitors the system (*Observe*), by collecting system's metrics, normalize the collected data and verifies if the model is still accurate given the latest data collected. The accuracy is the difference between the predicted utility value and the observed one. If it is not accurate, it triggers the next step, *Induce*. During this step, both the newly collected and remaining data are fed to the learning tool. The dataset is composed by pairs of system's metrics as input, and the utility function as output. Given the new dataset a new impact model will be learnt. The process is repeated in a cycle in order to deal with the system and environment's evolution. When considering the MAPE-K loop, *Observe* corresponds to the Monitor and Analyze steps. The *Induce* learns a model that will be added to the Knowledge component that assists the loop's steps.

This could be applied to our example by considering the system's features to be the active servers. For example if the first and second server were connected the input would be $(1, 1, 0, 0)$, considering we had four servers. The output would be the value given by a function based on the average response time and the number of active servers (cost).

When the adaptation process is triggered, the learnt impact model is used by a planner to choose the best state, i.e., which features should be enabled and disabled in order to maximize the system's utility.

This framework is supposed to be generic enough to be able to support the adaptation of different systems without major changes, and that is why the authors decided to use features as states and enabling and disabling them as actions. However, we argue that this comes with a loss of accuracy and flexibility, because the model does not consider environment properties or continuous system properties. On the other hand, the Learning Cycle idea can be easily used with continuous properties, instead of features, and a variety of learning tools. Furthermore, it is possible to add environment's properties to the input space.

2.4.1.1 Probabilistic Rule Learning

Sykes, Corapi, Magee, Kramer, Russo, and Inoue (2013) highlighted that taking into account non-determinism and having an updated impact model is important for an adaptive system. For this reason, they proposed the use of a non-monotonic probabilistic rule learner to update an impact model based on observations from the running system. Their model considers non-determinism and is readable, which means it achieves the goals we set out for this thesis.

Their approach is similar to FUSION's. They start by collecting data from the running system, then use this data to infer a model, which predicts how the environment is affected by the system's actions. In practice this corresponds to recording the resulting state after the execution of an adaptation action. In this work they only consider discrete states, therefore, an action either succeeds or fails. Furthermore, they encode the collected observations as a logic program in terms of the state before the action, the action itself, its success and at what time it occurred. In order to learn the non-monotonic probabilistic set of rules, they used NoMPRoL (Corapi, Sykes, Inoue, and Russo 2011). This tool, which is based on solving an inductive logic programming task, searches for a set of logic rules that are able to explain the observed data, in this case the impact of adaptation actions observed so far.

From the previous step results a set of hypotheses, each hypothesis is a set of rules. Therefore, the next step is to find which hypothesis, most likely, explain the observations. For each state, in the set of observations, they split the observations into history and state. Then, the most likely hypothesis is the one that explains the most observations on the given state, based on the considered history.

This approach is able to capture non-determinism because in this set of rules the input may yield different results. In a given state, the same action may succeed some times and fail others and this is captured in the set of rules. Therefore, they also compute each rule's probability of being true, to take it into account during the planning phase.

Although this approach results in an accurate and readable impact model that also explicitly represents non-determinism, its inability to consider continuous states makes it unfeasible to use with complex systems. Many metrics that compose states in complex systems are continuous and should not be discretized. In our example the average response time is a continuous variable, and its discretization would result in loss of information. Therefore, this tool could not be used for our solution.

2.4.2 Relevant Machine Learning Tools

Although the work presented by Sykes, Corapi, Magee, Kramer, Russo, and Inoue (2013) achieved the goals we have established, i.e., it inferred accurate and readable models, which represented non-determinism, its inability to work with continuous variables makes it inapt for our solution. Therefore, we will use the the idea presented by Elkhodary, Esfahani, and Malek (2010) to build an impact model which could be used to support an adaptation system. However, this could be used with different machine learning tools and impact models' specifications. In particular, our goal is to use a tool that: *a)* is able to learn an accurate model, which is common among machine learning tools; *b)* outputs a model, which should be translatable into a human readable impact model, to allow human involvement; *c)* is able to

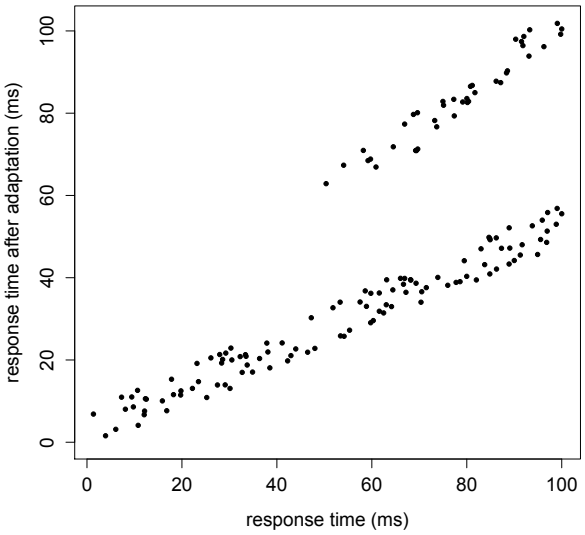

Figure 2.3: Example Dataset

explicitly consider and represent non-determinism, in order to account for the inherent non-determinism present in complex systems.

In order to present each tool's results we will use as input a simple dataset based on our example use case. A representation of this dataset is shown in Figure 2.3. This data was generated for explanation purposes to mimic a hypothetic scenario and does not represent real data.

2.4.2.1 Model Trees

Model Tree (Quinlan 1992) is the tool Elkhodary, Esfahani, and Malek (2010) used on their work. This tool is able to achieve high accuracy through the use of piece-wise linear functions based on rules. By using models based on a collection of rules, of the type *IF input THEN output*, where *input* is a set of conditions on input variables and *output* is a linear function of the input variables, they are able to achieve the goal of being intelligible. Thus, striking the balance between accuracy and readability. A model like this one would be easily translated to the specification seen previously. The validity conditions would correspond to the input conditions and the outcome to the rule's output.

However, this tool is not able to represent non-determinism. Instead, in the presence of non-determinism (more than one output for the same input) it will average every outcome and compute a single linear function. In Figure 2.4 the example dataset is represented alongside the result of applying *cubist*, a model tree learner, to the same dataset. In this Figure it is possible to observe that we lose information by computing the outcomes' average.

In our example it would be easy to use Model Trees to learn an impact model, but its inability to account for non-determinism means that it cannot be used to support an adaptation system with every policy described in the previous section. The first two policies could still be used, because this tool would output a value that is already some kind of average between the different possible results. If we wanted

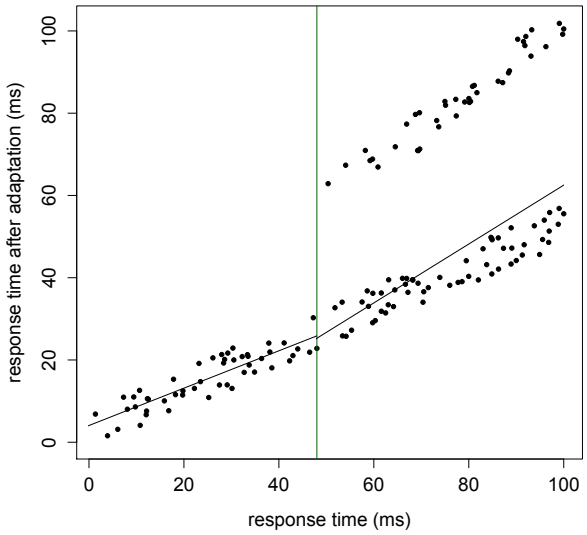

Figure 2.4: Model Tree Output

to use the first policy, the model learnt would output the state that results from the action's execution. In our example the output should be *rsp'*. For the second policy the model would output the utility value for the resulting state. In practice, the difference between both policies is that the data fed to this tool would be pairs where the input would be the state before the action's execution and the output would be the new state (*rps* , *rsp'*) and the new state's utility (*rsp* ,*utility(rsp')*), respectively for the first and second policies. In particular, this tool could not be used to support the third policy as it requires the knowledge of alternative impacts for each state, i.e., the model would have to account for non-determinism.

2.4.2.2 Fuzzy Inference System

Fuzzy Inference Systems (Chiu 1994; Guillaume 2001) also build accurate rule-based models. However, these rules use fuzzy set theory to map inputs to outputs.

Fuzzy set theory tries to approximate the set theory to the way humans think. Instead of fixed thresholds that define if a value belongs to a given set, e.g., average response time is bigger than 300ms, it uses degrees of membership, e.g., average response time is *high* with a certain degree. Therefore, the rules are of the form *IF* x_1 *IS* A_1 *AND* x_2 *is* A_2 *... THEN output*. In our example, the input condition could be the following for a given rule *IF res is High*.

Sugeno-type Fuzzy Inference Systems (Takagi and Sugeno 1985) have as outputs linear functions of the input variables, which results in a set of rules similar to the ones inferred with a Model Tree. Furthermore, because it uses degrees of membership, a given input can trigger more than one rule (more than one input condition may be true), which means that for a given input, several outputs can be considered, with some degree. This could indicate that it is possible to capture non-determinism with this model.

A model can be learnt using a dataset like the one we have been using. Then, using this dataset, the

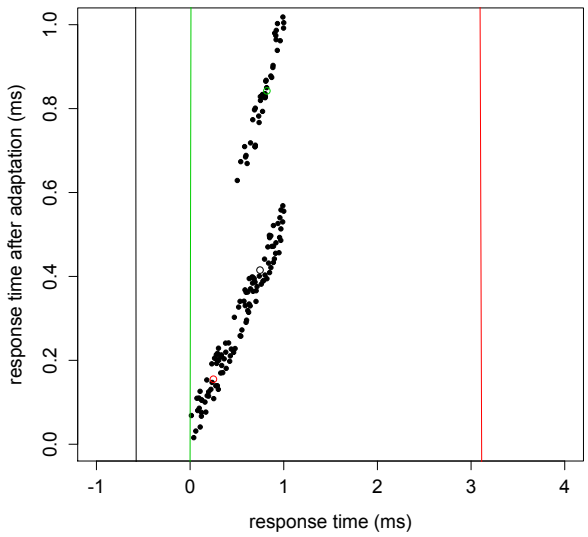

Figure 2.5: Fuzzy Inference System Output

first step would be to find the rules' conditions. This is achieved using fuzzy clustering, which results in a set of input for fuzzy rules. Lastly, for each input condition, or rule, a linear function will be inferred. The output linear function is computed by doing a weighted linear regression using the dataset. The weights, are based on each points' distance (degree of membership) to the cluster center being considered (Chiu 1994).

However, the adaptation system would have to account for degrees of membership, as every input would trigger multiple rules, and not all of them would be relevant. Also, because the linear function, which is the output of every rule, is inferred based on multiple data points and their degree of membership to the rule's input fuzzy set, the linear functions do not provide the information a human operator would expect, i.e., the linear function will not fit a subset of points, but instead will use the whole set, each point weighted by its membership value to that input fuzzy set. In Figure 2.5 we present the set of rules, given by this tool, as linear functions. The plot is scaled in order to show every linear function. The lines correspond to outputs and the hollow points to fuzzy cluster centers, furthermore, the lines correspond to the output of the cluster center of the same color. These results were obtained by using an implementation of the work presented by Chiu (1994). We can see that the functions do not fit correctly the dataset, therefore, we do not consider that it is able to explicitly represent non-determinism, because the rules do not make sense by themselves, only by combining them.

The policies that could be used with model trees can also be used with Fuzzy Inference Systems receiving similar datasets as the ones described for that tool. We consider that it does not support the use of the third policy because, as we said, the learnt rules do not have meaning individually, only their combination is a valid prediction.

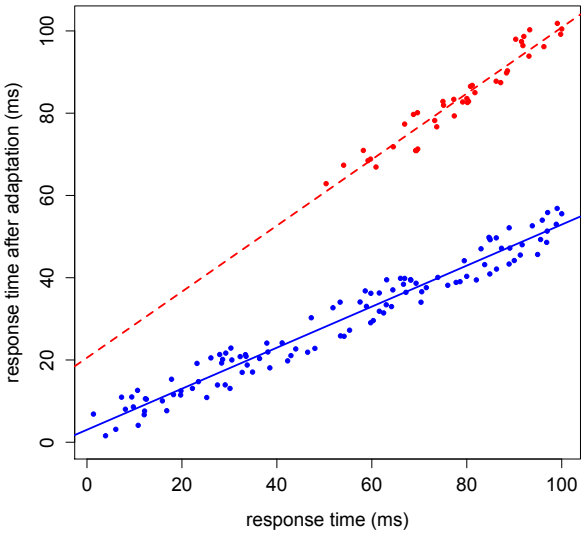

Figure 2.6: k-plane Output

2.4.2.3 Subspace Clustering

Subspace clustering is a technique that clusters data points into subspaces and, for each one, it infers a lower-dimension subspace that fits the cluster's data points, e.g., find two-dimensional subspaces that fit three-dimensional data points. In practice, this corresponds to finding hyper-planes that fit the dataset.

A particular example of this technique is the k-plane algorithm (Bradley and Mangasarian 2000), a data mining tool, which returns K hyper-planes that best represent a set of data points in a higher-dimensional space.

This algorithm receives as input the set of points and a value K, which indicates how many clusters it wants to find. It starts by creating K different random hyper-planes. Then, it assigns each point in the dataset to its closest plane. Afterwards, for each new collection of points (points that were closer/assigned to a given plane), a new plane is inferred by minimizing the sum of the squares of the distance between each point and the new plane. The process is repeated until the planes stabilize, i.e., every plane stays the same to the previous iteration.

Therefore, the result will be a set of K planes that fit the dataset. Applying this approach to our problem, we could map the state's input variables to the input dimensions, the impact on a property as the output dimension, and the impact functions to the planes found, which could be described with linear functions of the input variables. Furthermore, the use of linear functions to represent impact functions, instead of complex functions, facilitates the comprehension of the model by operators and follows the two previous systems output format.

This algorithm alone does not achieve the goals we established. However, it is able identify linear functions that fit a set of points. The linear functions inferred have no defined range, i.e., they have no limits. The application of this tool to our example may be seen in Figure 2.6.

Regarding the example policies we have been considering for different tools, neither of them can be supported by this k-plane algorithm. The first two policies could not be used because this tool does not implicitly average multiple outcomes, as the previous ones do, and impact functions do not have probabilities assigned to average them explicitly. The last policy cannot be used because we do not know where each plane, or impact function, is valid. Therefore, we would consider every impact function in every situation.

2.5 Discussion

Expert defined impact models are very flexible when it comes to its representation because their only requirement is that they have to be computationally parsable. Given that we argue that the model should be readable and able to explicitly represent non-determinism, this flexibility is a major advantage.

The work presented by Cámara, Lopes, Garlan, and Schmerl (2015) provides a specification of an impact model, which can be defined by experts and achieves the goal of being both readable and able to explicitly represent non-determinism. Therefore, the model specification they presented fits our goals. Furthermore, because this specification allows the explicit representation of non-determinism, it is possible to use it to support any of the policies presented earlier.

However, if the model is built by a human operator during the development phase and it is not updated, then it will not be, or remain, accurate. The inaccuracies are caused by the operator's inability to capture complex system's behaviors and due to the environment and system's constant evolution, which will invalidate outdated impact models.

For these reasons, our goal is to use the specification presented by Cámara, Lopes, Garlan, and Schmerl (2015), coupled with an automated tool capable of learning or updating the impact model. We presented some tools able to learn models, which could be used to learn impact models. However, for this work, we need a tool that is: *a)* able to learn an accurate model, in order to make correct predictions and do not compromise the planning process; *b)* readable, in order to involve the human operator into the adaptation loop; *c)* and able to explicitly represent non-determinism.

The ability to represent non-determinism is essential to support every policy presented earlier.

Learners that average different outcomes into a single impact function, such as Model Trees, are able to support the first policy because the learner implicitly averages the possible results. In order to support the second policy, the learner would have to learn to predict the utility value, instead of the impact. However, this approach links the learnt model to the utility function. Therefore, if the utility function changes, the model is invalidated, which could be a problem for systems with changing goals, because every time the utility function changed a new model would have to be learnt. Finally, because these tools do not represent alternative outcomes for a given state, the third policy is not applicable.

We consider that learning a model using a Fuzzy Inference System does not result in a readable model, as seen previously.

k-plane is a tool able to infer a set of linear functions that fit a dataset. Furthermore, data points with the same input and different outputs may "belong" to different impact functions, therefore, supporting

the representation of non-determinism. However, this tool does not output valid input regions, i.e., in which regions may the impact function be considered. For these reasons we will use this algorithm as a starting point and use it as a building block to our solution. Using this algorithm we are able to find the impact functions necessary to fit the dataset, then, because we ultimately want to write the model in the language proposed by Cámara, Lopes, Garlan, and Schmerl (2015), we have to identify each function's input region. Lastly, we will identify input regions where non-determinism exists, compute the impact's probabilities and translate to a readable model.

There are many other tools that were not considered during this chapter, such as clustering algorithms, Artificial Neural Networks (Dietterich 2009) and Support Vector Machines (Cortes and Vapnik 1995). Artificial Neural Networks output a complex model, which would be hard to translate into a readable model. Clustering algorithms learn how to split the space in order to separate "different" points. However, we want to consider impact functions as linear functions, which could eventually intersect. So separating points according to their euclidean distance to a cluster center is not the best approach. Furthermore, we did not consider supervised learning algorithms, such as Support Vector Machines, as we assume there is no available knowledge of which points belong to which impact, or even if two points were the result of one impact or two.

Summary

Due to the increasing complexity of systems, adaptation processes are becoming more relevant for real world systems. In this chapter we introduced some concepts that will be used throughout the rest of this dissertation and surveyed systems and tools that could help us create a technique able to learn readable adaptation models, which are also aware of non-determinism.

The next chapter will introduce the architecture and implementation details of our system.

3 Ramun

In this chapter we propose an approach to drive the dynamic adaptation of systems by learning an impact model, which achieves the goals specified in Chapter 1.

We start by providing an overview of an approach to revise adaptation models under non-determinism, or RAMUN for short, and then describe in more detail each step. To support RAMUN's presentation, we will use as a running example the activation of a server, such as we did in the previous chapter.

3.1 Overview

Algorithm 1: RAMUN's high level description

```
Input: impacts: impact model (set of impact functions)
dataset ← initialization(impacts)
while TRUE do
    dataset ← collect_data(dataset)
    planes ← infer_functions(dataset)
    ranges ← get_ranges(dataset, planes)
    probabilities ← compute_probabilities(dataset, planes, ranges)
    dataset ← data_curation(dataset)
end
```

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. This periodic procedure is captured in Algorithm 1.

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: *a)* the linear impact functions corresponding to each possible outcome; *b)* the input space in which each linear function is considered to be valid; *c)* the probability associated with each outcome and *d)* 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.

3.2 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, Lopes, Garlan, and Schmerl (2015). 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: *a)* the functions that represent the impact of performing an action in a given state, *b)* the probabilities associated to each impact function and *c)* 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.

3.3 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 in Figure 3.1.

3.4 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 (Bradley and Mangasarian 2000) 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 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

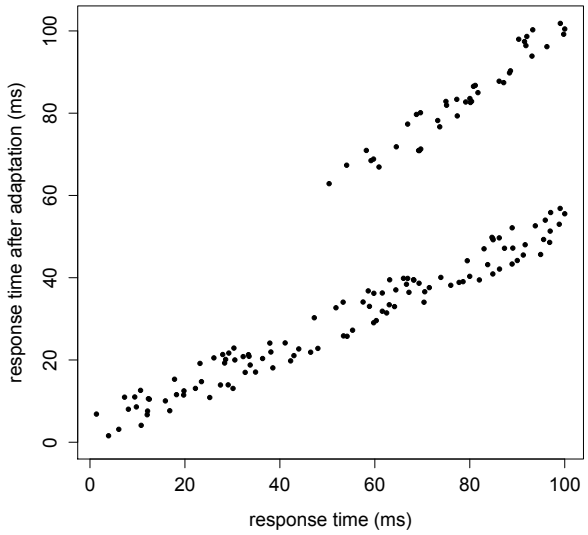

Figure 3.1: Dataset after collecting new samples

average error between each sample in the subset and its closest plane. For each sample the error is computed as:

$$\text{Error} = \frac{abs(\text{real} - \text{predicted})}{abs(\text{predicted})} \tag{3.1}$$

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 (10-fold cross-validation). 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 3.2.

3.5 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 3.2, 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

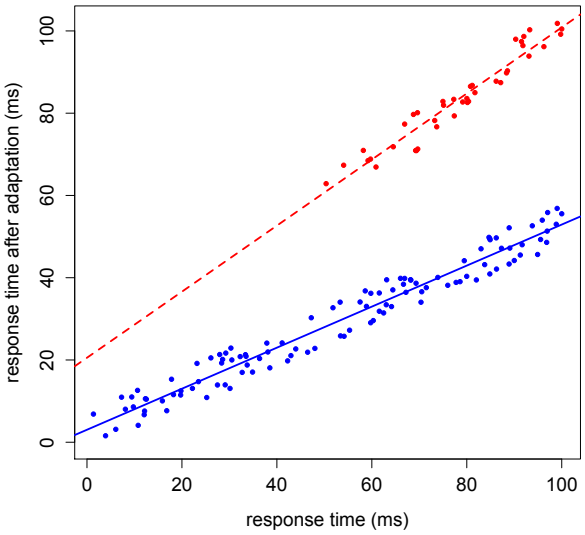

Figure 3.2: Dataset and impact functions.

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

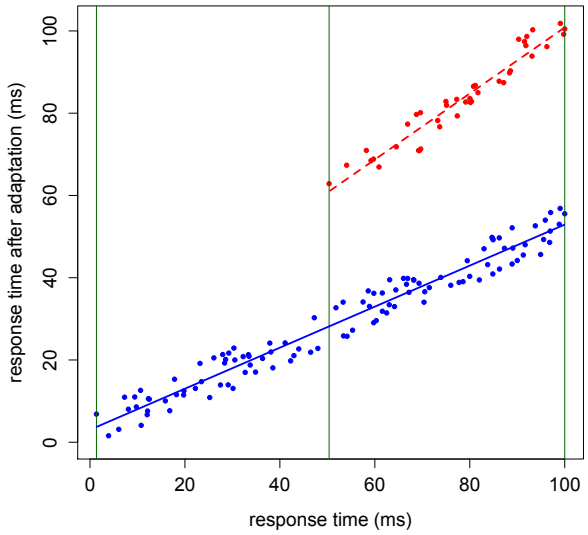

Figure 3.3: Validity ranges represented as green vertical lines.

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 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 one presented in Figure 3.3.

3.6 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:

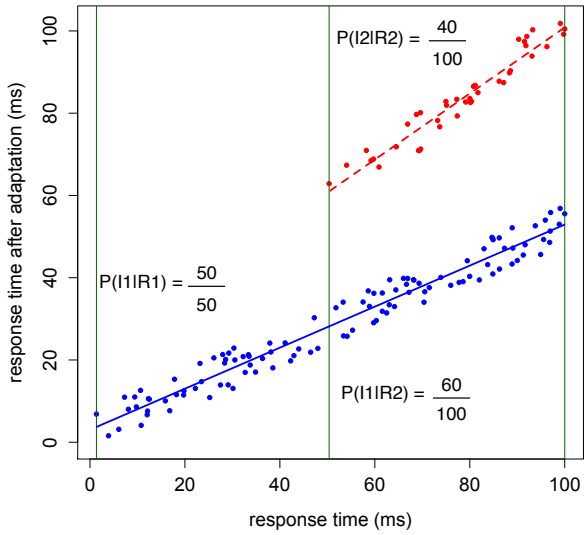

Figure 3.4: Probability computation specified for each impact function.

$$P(\text{impact}_i|\text{region}) = \frac{\#(\text{region}, \text{impact}_i)}{\#(\text{region})} \tag{3.2}$$

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 3.4, where the corresponding fraction is depicted next to each plane.

3.7 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 and Romano (2015), 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 these points is orthogonal to the contributions of this dissertation. 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.

Summary

In this chapter we have been through the design and implementation of RAMUN. First, we described the shortcomings of a tool like k-plane to learn impact models that we could write in the specification presented by Cámara, Lopes, Garlan, and Schmerl (2015). Next, we presented the extra steps RAMUN performs to learn a model, which could be written using this specification, with the help of k-plane.

In the next chapter we present the experimental evaluation made using this prototype.

4 Evaluation

We now present an experimental evaluation of RAMUN. For the evaluation we have experimented with a concrete instantiation of the abstract problem presented in Chapter 2. 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.

In the following sections we will describe the experiments made to test our solution's ability to fulfill the requirements we previously described. More concretely, the experiments will show:

a) the consequence of measurement's errors on the model's inference, and what that means for the parametrization of our solution;

b) this solution is able to detect different action's impacts for the same state, i.e., captures non-determinism;

c) how this solution compares to more typical approaches;

d) what is the effect of having an initial impact model, and how the initial model's correctness affects the learning phase's time;

e) and finally, we present an example where considering non-determinism is an advantage.

The alternative solution we will be considering in this chapter is *cubist* [2]. We opted to use this model learner because it is rule based, which is congruent with the rules we wish to output, considers continuous states, necessary to model real system's behaviors and it does not explicitly represent non-determinism, which will allow to present the advantages of doing so.

4.1 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 [3] to generate different workloads and drive httperf (Mosberger and Jin 1998) to issue the requests. To distribute the load among servers, we have used HAProxy 1.6 [4] running on a separate virtual machine.

[1] Rice University Bidding System: http://rubis.ow2.org
[2] J. R. Quinlan, "Rulequest Cubist" http://www.rulequest.com
[3] Autobench: http://www.xenoclast.org/autobench/
[4] Haproxy: the reliable, high performance tcp/http load balance: http://www.haproxy.org/

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 (2011), GoGrid's [5] 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.

4.2 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. 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 the adaptation action 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

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

Table 4.1: Number of impact functions found

Error Threshold\Added 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

the differences in outcomes are entirely due to the artifacts of each approach (and not to fluctuations in the data collected).

4.3 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 (Ramirez, Jensen, and Cheng 2012). 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 noisy datasets, where an additional error was added. We have created noisy 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 noisy 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 unmodified 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 4.1 we tested different error thresholds with datasets that had different levels of noisy. 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 4.1, when the added error starts increasing, the number of impact

Listing 4.1: Impact Model Learnt from elastic scaling of RUBiS

```
define f(s, req, rsp) = -34.994 * s + 0.025 * req + 0.360 * rsp + 74.506
define g(s, req, rsp) = -135.084 * s + 0.014 * req + 0.488 * rsp + 502.873
impactmodel enlistServer
1 <= s <= 3 & 250 <= req <= 800 & 3.655 <= rsp <= 595.775 ->
        {foreach s:S | s.isActive'=true &
                [1]     forall c:Client | c.rsp' = f(s, req, rsp)}
1 <= s <= 3 & 800 <= req <= 9950 & 4.255 <= rsp <= 595.775 ->
        {foreach s:S | s.isActive'=true &
                [0.623] forall c:Client | c.rsp' = f(s, req, rsp) +
                [0.377] forall c:Client | c.rsp' = g(s, req, rsp)}
1 <= s <= 3 & 9950 <= req <= 10000 & 3.655 <= rsp <= 595.775 ->
        {foreach s:S | s.isActive'=true &
                [1]     forall c:Client | c.rsp' = g(s, req, rsp)}
```

Listing 4.2: 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:
        [0.623] rsp' = -34.994 * s + 0.025 * req + 0.360 * rsp + 74.506
        [0.377] 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
```

functions found also increases. Furthermore, if the error threshold is small (0.2, 0.25) then even the unmodified 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 unmodified 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 noisy 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.

4.4 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.

Listing 4.1 is the representation of the learnt impact model from our use case using the specification proposed by Cámara, Lopes, Garlan, and Schmerl (2015). In order to simplify the model's presentation in Listing 4.2 we present an equivalent model in a simpler representation, which only presents information relevant for this dissertation. In particular, the simplified version represents validity conditions in the non indented lines as a combination of ranges per dimension, it also represents impact functions in each condition as a linear function of the input variables, finally, the probability of each impact function being observed is represented before each impact function in between square brackets. However„ the simplified version 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, instead of being above the impact model representation.

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.

4.5 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 system 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 4.1 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.

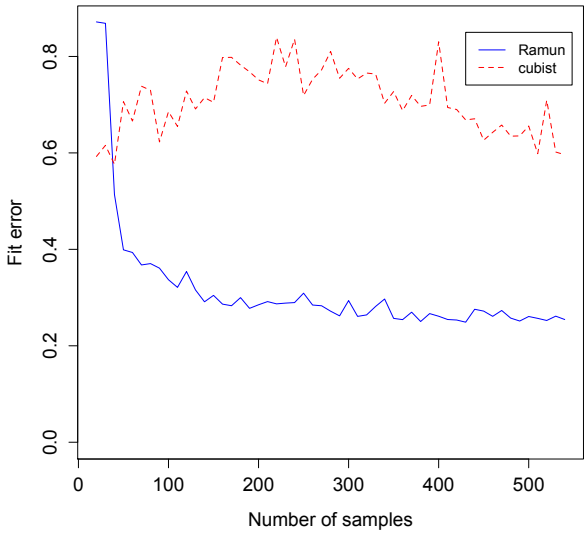

Figure 4.1: Fit comparison

4.6 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 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 50% of the dataset is split into training and test sets to validate the model through 10-fold cross-validation, as we did previously. Then we will feed to each version a growing dataset as input, from the training set, and test it with the test set.

As we can see in Figure 4.2, 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

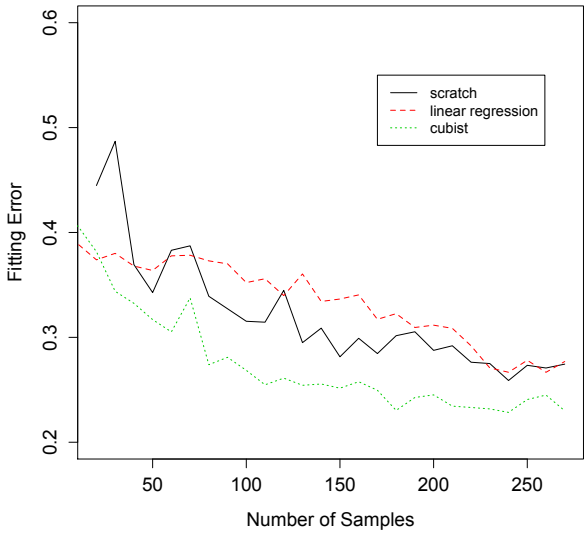

Figure 4.2: Fitting with Initial Models

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.

4.7 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 synthetic samples derived from an impact function that is parallel to the real impact but with an added 250ms.

In our experiment we will depart from the a dataset with the synthetic samples only and gradually add samples from the real dataset and measure the probability of the impact functions inferred based on synthetic samples and real samples (an impact function can also be inferred based on real and synthetic samples and in that case we will consider that it was based on the most common type of sample). Because we are only considering two impacts we will have at most two different impact functions.

We expect that by adding real samples to the dataset, the influence of the synthetic samples, which are not increased over time, will decrease and eventually have an insignificant impact on the inference process.

In Figure 4.3 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

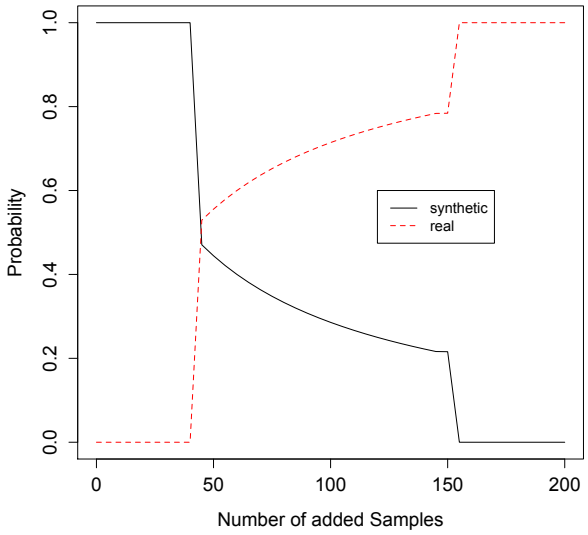

Figure 4.3: Probability of synthetic and real impact functions

distance between impact functions will have a great impact on these values. In particular, if the distance between the real and synthetic samples 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 correct anymore.

In Figure 4.4 we show the evolution of the probability of the impact function inferred based on synthetic samples when real samples are added. We did the same experiment with an initial impact function that is separated from the real impact by 250ms and 300ms, respectively the lower and higher error. As we expected the initial error influences how fast we can find new impact functions (inferred from real data), and how fast we are able to disregard impact functions inferred exclusively with synthetic data.

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.

4.8 Relevance of Capturing Non-Determinism

In order to illustrate the relevance of considering non-determinism when supporting adaptation systems, we will use as an example the Amazon website. At the time, Amazon found that an increase of 100ms of latency resulted in a loss of 1% in sales (Liddle 2008). Considering Amazon's dimension, it is safe to assume that the cost of such an increase in the experienced latency is much higher than the

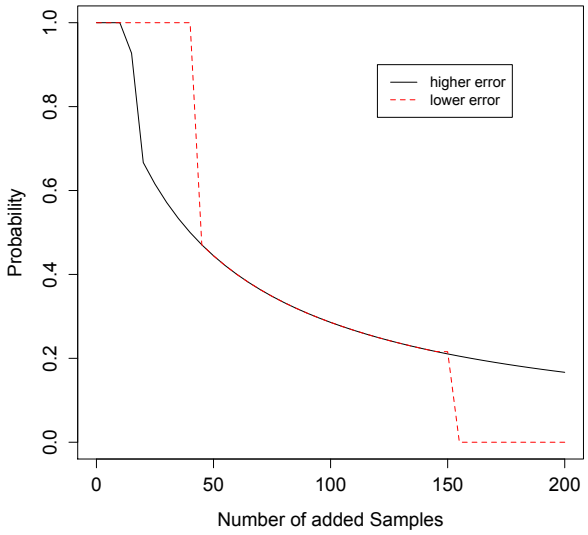

Figure 4.4: Probability of synthetic impact functions with different errors

operational cost of having, for example, more servers connected.

With this example and our case study in mind, we want present a simple example to check if considering non-determinism would be able to improve our case study's earnings assuming we are in the same conditions as Amazon. Therefore, we learnt an impact model, similar to the one presented in Listing 4.2 using Cubist, represented in Listing 4.3, which does not consider non-determinism, and compare it with the one we presented earlier in Section 4.4.

Now, considering a state where the number of active servers is 2, the current request rate is 1000 requests per second and the average response time is 100ms, we predict with both models the expected average response time if a new server was connected. Using the model learnt with Cubist, the expected average response time is 19.5616ms. If we use the model learnt with RAMUN the expected average response time is 65.518ms with 62.3% of probability and 295.505 with 37.7%. This information means that by considering non-determinism, we could predict that with a probability of 37.7% this adaptation could result in a loss of 1.955% in sales and with 62.3% of probability it could result in a boost of 0.345%. Therefore, the planner should consider connecting more than one server to reduce this chances of loss.

Listing 4.3: Impact Model learnt with Cubist

```
impact model(connect server):
rsp <= 89.45:
        [1]     rsp' = -45 * s + 0.034 * req + 0.21 * rsp + 100.1104
req <= 1650 & rsp > 89.45:
        [1]     rsp' = -211 * s + 0.0138 * req + 0.28 * rsp + 399.7616
s > 2 & rsp > 89.45:
        [1]     rsp' = 0.0136 * req + 0.25 * rsp + 136.8925
s <= 2 & req > 1650 & rsp > 89.45:
        [1]     rsp' = -30 * s + 0.0178 * req + 0.5 * rsp + 104.2625
```

On the other hand, if the planner only considers one outcome, like Cubist does, it will not be able to predict these scenarios, which ultimately could cost money to the service provider. There is a huge difference between RAMUN's model worst case and Cubist's only one. Considering Amazon's data, the difference between both could lead to a loss of 2.7% in sales. However, even our best case is worst than Cubist's prediction. So, considering the RAMUN's best and worst case for this scenario the loss could still be of approximately 2.3%.

As we pointed out in Chapter 2, this problem could be solved by using Cubist to learn based on an utility function, instead of impact. However, this approach is not robust to changes in the utility function. Which could be a problem to a business like Amazon that can change its requirements if they observe differences in this sales loss information. If the utility function changes, the model will have to be learnt all over again, which could be a problem to highly dynamic systems.

Summary

In this chapter we introduced the experimental evaluation made to RAMUN and its results. As our case study we used RUBiS, a well known auction website similar to eBay. Using this case study we corroborated RAMUN's validity to learn impact models. We also compared our proposed technique with a state of the art learner which outputs a model similar to RAMUN's, but that did not consider non-determinism explicitly. Finally, we presented some concerns one must have when parameterizing RAMUN and the consequences these choices may have on the learnt model.

The next chapter finishes this thesis by presenting the conclusions regarding the work developed and also introduces some directions in terms of future work.

5 Conclusions & Future Work

We have described the design, implementation, and experimental evaluation of RAMUN, a system that dynamically learns and refines an impact model, which supports the adaptation process by predicting the impact of adaptation actions.

The solution's key aspects are its ability to explicitly capture and learn adaptation action's non-deterministic impacts on the managed system, and the fact that the inferred impact models are easily interpretable by human operators, while keeping the model up to date. By capturing non-determinism the model's robustness, in the presence of exogenous factors that cannot be easily measured or captured in the model, is enhanced. Furthermore, it allows to use different adaptation policies, as the ones presented alongside our example. The model's readability allows the involvement of human administrators in the loop – a property that we argue is essential to build trust in the self-adaptive system. Finally, the fact that the system can update the model automatically 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 the output (impact of actions). We compared our solution with the tool that did not explicitly account for non-determinism and the result was that this feature allows to learn a model that fits better the dataset, which means it represents better the system's behavior. We also evaluated the consequence of some parametrization choices, such as the error threshold or the initial model used to initialize the dataset.

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 (2015).

References

(2005). An Architectural Blueprint for Autonomic Computing. Technical report, IBM.

Bradley, P. S. and O. L. Mangasarian (2000). k-Plane Clustering. *Journal of Global Optimization 16*(1), 23–32.

Brun, Y., G. Di Marzo Serugendo, C. Gacek, H. Giese, H. Kienle, M. Litoiu, H. Müller, M. Pezzè, and M. Shaw (2009). Engineering self-adaptive systems through feedback loops. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Volume 5525 LNCS, pp. 48–70.

Cámara, J., A. Lopes, D. Garlan, and B. Schmerl (2015). Adaptation impact and environment models for architecture-based self-adaptive systems. *Science of Computer Programming 127*(C), 50–75.

Cheng, B. H. C., 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 (2009). Software Engineering for Self-Adaptive Systems: A Research Roadmap. In *Softw. Eng. Self-Adaptive Syst.*, Volume 5525, pp. 1–26.

Cheng, S. W. and D. Garlan (2012). Stitch: A language for architecture-based self-adaptation. *Journal of Systems and Software 85*(12), 2860–2875.

Chiu, S. L. (1994). Fuzzy Model Identification Based on Cluster Estimation. *Journal of Intelligent and Fuzzy Systems 2*(3), 267–278.

Corapi, D., D. Sykes, K. Inoue, and A. Russo (2011). Probabilistic rule learning in nonmonotonic domains. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Volume 6814 LNAI, pp. 243–258. Springer.

Cortes, C. and V. Vapnik (1995). Support-vector networks. *Machine Learning 20*(3), 273–297.

De Lemos, R., H. Giese, H. A. Müller, M. Shaw, J. Andersson, M. Litoiu, B. Schmerl, G. Tamura, N. M. Villegas, T. Vogel, D. Weyns, L. Baresi, B. Becker, N. Bencomo, Y. Brun, B. Cukic, R. Desmarais, S. Dustdar, G. Engels, K. Geihs, K. M. Göschka, A. Gorla, V. Grassi, P. Inverardi, G. Karsai, J. Kramer, A. Lopes, J. Magee, S. Malek, S. Mankovskii, R. Mirandola, J. Mylopoulos, O. Nierstrasz, M. Pezzè, C. Prehofer, W. Schäfer, R. Schlichting, D. B. Smith, J. P. Sousa, L. Tahvildari, K. Wong, and J. Wuttke (2013). Software engineering for self-adaptive systems: A second research roadmap. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Volume 7475 LNCS, pp. 1–32.

Didona, D. and P. Romano (2015). On Bootstrapping Machine Learning Performance Predictors via Analytical Models. In *Icpads*, pp. 405–413.

Dietterich, T. G. (2009). *Machine learning in ecosystem informatics and sustainability*. McGraw Hill series in computer science. McGraw-Hill.

Elkhodary, A., N. Esfahani, and S. Malek (2010). Fusion. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering - FSE '10*, pp. 7.

Esfahani, N. and S. Malek (2013). Uncertainty in self-adaptive software systems. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Volume 7475 LNCS, pp. 214–238.

Garlan, D., S. W. Cheng, A. C. Huang, B. Schmerl, and P. Steenkiste (2004). Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer 37*(10), 46–54.

Gil, R. (2015). Automated Planning for Self-Adaptive Systems. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, Volume 2, pp. 839–842.

Guillaume, S. (2001). Designing fuzzy inference systems from data: An interpretability-oriented review. *IEEE Transactions on Fuzzy Systems 9*(3), 426–443.

Kephart, J. O. and D. M. Chess (2003). The vision of autonomic computing. *Computer 36*(1), 41–50.

Liddle, J. (2008). Amazon found every 100ms of latency cost them 1% in sales. *The GigaSpaces 27*.

Liu, H. (2011). A measurement study of server utilization in public clouds. *Proceedings - IEEE 9th International Conference on Dependable, Autonomic and Secure Computing, DASC 2011*, 435–442.

Mosberger, D. and T. Jin (1998). Httperf—a Tool for Measuring Web Server Performance. *ACM SIGMETRICS Performance Evaluation Review 26*(3), 31–37.

Quinlan, J. R. (1992). Learning with continuous classes. *Machine Learning 92*, 343–348.

Ramirez, A. J., A. C. Jensen, and B. H. C. Cheng (2012). A taxonomy of uncertainty for dynamically adaptive systems. In *ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, pp. 99–108.

Salehie, M. and L. Tahvildari (2009). Self-adaptive software. *ACM Transactions on Autonomous and Adaptive Systems 4*(2), 1–42.

Sykes, D., D. Corapi, J. Magee, J. Kramer, A. Russo, and K. Inoue (2013). Learning revised models for planning in adaptive systems. In *Proceedings - International Conference on Software Engineering*, San Francisco, CA, USA, pp. 63–71.

Takagi, T. and M. Sugeno (1985). Fuzzy Identification of Systems and Its Applications to Modeling and Control. *IEEE Transactions on Systems, Man and Cybernetics SMC-15*(1), 116–132.

Xu, J., M. Zhao, J. Fortes, R. Carpenter, and M. Yousif (2008). Autonomic resource management in virtualized data centers using fuzzy logic-based approaches. *Cluster Computing 11*(3), 213–227.