The Faculty of Science, Technology and Communication

Dissertation

Defence held on ??/??/2019 in Luxembourg

to obtain the degree of

Docteur de l'Université du Luxembourg et de l'Université de Rennes 1 en Informatique

by

Ludovic Mouline

Towards a Modeling Framework with Temporal and Uncertain Data for Adaptive Systems

Abstract

Vision: As state-of-the-art techniques fail to model efficiently the evolution and the uncertainty existing in dynamically adaptive systems, the adaptation process makes suboptimal decisions. To tackle this challenge, modern modelling frameworks should efficiently encapsulate time and uncertainty as first-class concepts.

Context Smart grid approach introduces information and communication technologies into traditional power grid to cope with new challenges of electricity distribution. Among them, one challenge is the resiliency of the grid: how to automatically recover from any incident such as overload? These systems therefore need a deep understanding of the ongoing situation which enables reasoning tasks for healing operations. **Abstraction** is a key technique that provided an illuminating description of systems, their behaviours, and/or their environments alleviating their complexity. **Adaptation** is a cornerstone feature that enables reconfiguration at runtime for optimizing software to the current and/or future situation.

Abstraction technique is pushed to its paramountcy by the model-driven engineering (MDE) methodology. However, information concerning the grid, such as loads, is not always known with absolute confidence. Through the thesis, this lack of confidence about data is referred to as **data uncertainty**. They are approximated from the measured consumption and the grid topology. This topology is inferred from fuse states, which are set by technicians after their services on the grid. As humans are not error-free, the topology is therefore not known with absolute confidence. This data uncertainty is propagated to the load through the computation made. If it is

neither present in the model nor not considered by the adaptation process, then the adaptation process may make suboptimal reconfiguration decision.

The literature refers to systems which provide adaptation capabilities as dynamically adaptive systems (DAS). One challenge in the grid is the phase difference between the monitoring frequency and the time for actions to have measurable effects. Action with no immediate measurable effects are named **delayed action**. On the one hand, an incident should be detected in the next minutes. On the other hand, a reconfiguration action can take up to several hours. For example, when a tree falls on a cable and cuts it during a storm, the grid manager should be noticed in real time. The reconfiguration of the grid, to reconnect as many people as possible before replacing the cable, is done by technicians who need to use their cars to go on the reconfiguration places. In a fully autonomous adaptive system, the reasoning process should consider the ongoing actions to avoid repeating decisions.

Problematic **Data uncertainty and delayed actions are not specific to smart grids.**

First, data are, almost by definition, uncertain and developers always work with estimates. Hardware sensors have by construction a precision that can vary according to the current environment in which they are deployed. A simple example is the temperature sensor that provides a temperature with precision to the nearest degree. Software sensors approximate also values from these physical sensors, which increases the uncertainty. For example, CPU usage is computed counting the cycle used by a program. As stated by Intel, this counter is not error-prone[1].

Second, it always exists a delay between the moment where a suboptimal state is detected by the adaptation process and the moment where the effects of decisions taken are measured. This delayed is due to the time needed by a computer to process data and, eventually, to send orders or data through networks. For example, migrating a virtual machine from a server to another one can take several minutes.

Through this thesis, I argue that this data uncertainty and this delay

[1]https://software.intel.com/en-us/itc-user-and-reference-guide-cpu-cycle-counter

cannot be ignored for all dynamic adaptive systems. To know if the data uncertainty should be considered, stakeholders should wonder **if this data uncertainty affects the result of their reasoning process, like adaptation**. Regarding delayed actions, they should verify **if the frequency of the monitoring stage is lower than the time of action effects to be measurable**. These characteristics are common to smart grids, cloud infrastructure or cyber-physical systems in general.

Challenge These problematics come with different challenges concerning the representation of the knowledge for DAS. The global challenge address by this thesis is: **how to represent the uncertain knowledge allowing to efficiently query it and to represent ongoing actions in order to improve adaptation processes?**

Vision **This thesis defends the need for a unified modelling framework which includes, despite all traditional elements, temporal and uncertainty as first-class concepts.** Therefore, a developer will be able to abstract information related to the adaptation process, the environment as well as the system itself.

Concerning the adaptation process, the framework should enable abstraction of the actions, their context, their impact, and the specification of this process (requirements and constraints). It should also enable the abstraction of the system environment and its behaviour. Finally, the framework should represent the structure, behaviour and specification of the system itself as well as the actuators and sensors. All these representations should integrate the data uncertainty existing.

Contributions Towards this vision, this document presents two approaches: a temporal context model and a language for uncertain data.

The temporal context model allows abstracting past, ongoing and future actions with their impacts and context. First, a developer can use this model to know what the ongoing actions, with their expect future impacts on the system, are. Second, she/he can navigate through past decisions to understand why they have been made when they have led to a sub-optimal state.

The language, named Ain'tea, integrates data uncertainty as a first-class citizen. It allows developers to attach data with a probability distribution which represents

their uncertainty. Plus, it mapped all arithmetic and boolean operators to uncertainty propagation operations. And so, developers will automatically propagate the uncertainty of data without additional effort, compared to an algorithm which manipulates certain data.

Validation Each contribution has been evaluated separately. The context model has been evaluated through the performance axis. The dissertation shows that it can be used to represent the Luxembourg smart grid. The model also provides an API which enables the execution of query for diagnosis purpose. In order to show the feasibility of the solution, it has also been applied to the use case provided by the industrial partner.

The language has been evaluated through two axes: its ability to detect errors at development time and its expressiveness. Ain'tea can detect errors in the combination of uncertain data earlier than state-of-the-art approaches. The language is also as expressive as current approaches found in the literature. Moreover, we use this language to implement the load approximation of a smart grid furnished by an industrial partner, Creos S.A.[2].

Keywords: dynamically adaptive systems, knowledge representation, model-driven engineering, uncertainty modelling, time modelling

[2]Creos S.A. is the power grid manager of Luxembourg. `https://www.creos-net.lu`

Table of Contents

1 Introduction

Contents

Model-driven engineering methodology and dynamically adaptive systems approach are combined to tackle challenges brought by systems nowadays. After introducing these two software engineering techniques, we describe five problems that we identified for such systems: data uncertainty, actions with long-term effects, emergent behaviours of such systems, different evolution paces of the subparts, and the temporal dimension in their structures and behaviours. We present the challenges that come with these problems. Before describing the two contributions of this thesis, we scope to the addressed sub-challenges tackled.

1.1 Context

Utilities are introducing more and more Information and Communication Technology (ICT)* in the grid to face the new challenges of electricity supply [**farhangi2010path**; **ipakchi2009grid**; **DBLP:journals/comsur/FangMXY12**]. The literature and the industry refer to these nowadays power grids as smart grid*.

In this document, we focus on the **self-healing*** capacity of such grids. A self-healing system* can automatically repair any incident, software or hardware, at runtime [**DBLP:journals/computer/KephartC03**]. For example, a smart grid can optimise the power flow to deal with failures of transformers[1] [**DBLP:journals/comsur/FangMXY**
In this way, the incident will impact as few users as possibles, ideally none.

This healing mechanism can be performed only if the smart grid* has a deep understanding of itself (its structure* and its behaviour*) and its environment* (where it is executed). This understanding can be extracted from an, or a set of, **abstraction**(s) of these elements. Abstractions provide an illuminating description of systems, their behaviours*, or their environments*. For example, Hartmann *et al.,* [**DBLP:conf/smartgridcomm/0001FKTPTR14**] provide a class diagram that describes the smart grid topology, when it uses power lines communications[2].

More generally, a self-healing system* is a **self-adaptive system***. Self-adaptive systems* optimize their behaviours* or configurations at runtime in response to a modification of their environments* or their behaviours* [**DBLP:conf/dagstuhl/ChengLGIMABBBC**
Kephart and Chess [**DBLP:journals/computer/KephartC03**] laid the groundwork for this approach, based on an IBM white paper [**computing2006architectural**]
Since then, practitioners have applied it to different domains [**DBLP:journals/corr/abs-1904-015**
such as cloud infrastructure [**DBLP:conf/icac/JavadiG17**; **OpenStack:Watcher:Wiki**; **DBLP:conf/icse/BarnaKFL17**] or Cyber-Physical System (CPS)* [**DBLP:conf/icac/Lalanda**
DBLP:conf/cbse/FouquetMFBPJ12; **DBLP:conf/smartgridsec/0001FKNT14**]

Model-Driven Engineering (MDE)* uses the abstraction mechanism to facilitate the development of nowadays software [**DBLP:journals/computer/Schmidt06**;

[1]Transformers change the voltage in the cables.
[2]Data are sent through cables that also distribute electricity.

DBLP:conf/ifm/Kent02; **DBLP:series/synthesis/2017Brambilla**]. This method-[redacted] ology can be applied to different stages of software development. In this thesis, we focus on one of its paradigms: **models@run.time*** [**DBLP:journals/computer/BlairBF09**[redacted] **DBLP:journals/computer/MorinBJFS09**]. The state of the system, its environment*, or its behaviour* is reflected in a model*, used for analysis. Developers can use this paradigm to implement adaptive systems* [**DBLP:journals/computer/MorinBJFS09**[redacted] **DBLP:conf/smartgridsec/0001FKNT14**].

In this thesis, we focus on the use of MDE* techniques, and more specifically, the models@run.time* paradigm, for the implementation of self-adaptive systems*. Adaptation processes use models* to have a deep understanding of the system, its structure and its behaviour, and its environment. Using the vocabulary of the research community, the model* represents the knowledge*. That is, **we studied the representation of the knowledge* of adaptive systems*, using the models@run.time* paradigm.**

1.2 Problem statement

During our study, we have identified several characteristics of adaptive systems* that bring challenges to the software engineering research community. First, information gathered is not always known with absolute confidence. Second, reconfigurations may not be immediate, and their effects are not instantaneously measured. Third, system behaviour may be emergent [**zio2011uncertainties**], *i.e.,* it cannot be entirely known at design time. Four, the different sub-parts of the system do not evolve at the same pace. Five, structure and behaviour of systems have a time dimension.

1.2.1 Data are uncertain

Most fuses are manually open and close by technicians rather than automatically modified. Then, technicians manually report the modifications done on the grid. Due to human mistakes, this results in errors. The grid topology is thus uncertain. This uncertainty is propagated to the load approximation, used to detect overloads in the grid. Wrong reconfigurations might be triggered, which could be even worse than if no change would have been applied.

More generally, **data are, almost by definition, uncertain and developers always work with estimates** [**DBLP:conf/asplos/BornholtMM14**; **metrology2008evaluat** **DBLP:journals/tkde/AggarwalY09**]. The uncertainty may be explained by how data are collected. We can distinguish three categories: sensors, humans, and results of computations. Sensors (software or hardware) always estimate the value and have a precision value due to the method of measurement [**metrology2008evaluation**; **DBLP:conf/asplos/BornholtMM14**]. Humans are error-prone. And computations can either give an approximation or be based on uncertain data. This uncertainty is then propagated through all steps until the final result.

For a specific domain, this uncertainty may impact the understanding of the real situation. For example, the uncertainty of the Central Processing Unit (CPU)* clock is too low to damage the percentage load of the processor. However, the uncertainty of the Global Positioning System (GPS)* may impact the understanding by, for instance, showing the user on the wrong road (compared to the real one). **If the data uncertainty can mislead the understanding of a system behaviour or state, then developers should implement an uncertainty-aware system.** For adaptive systems*, this lack of confidence may trigger suboptimal adaptations.

1.2.2 Actions have long-term effects

Reconfiguring a smart grid implied to change the power flow. It is done by connecting or disconnecting specific cables. That is, opening or closing fuses. As said before, technicians need to drive physically to fuse locations to modify their states. Besides, in the case of the Luxembourg smart grid, meters send energy measurement every 15 min, non-synchronously. Between the time a reconfiguration of the smart grid is decided, and the time the effects are measured, a delay of at least 15 min occurs. On the other hand, an incident should be detected in the next minutes. If the adaptation process does not consider this difference of paces, it can cause repeated decisions.

Actions are never immediate, take time to be executed, and have long-term effects. Through this thesis, we refer to such actions as long-term actions*. In computer science, the definition of "immediate" is specific to a domain. For example, for graphical user interfaces, a response time of less than 200 ms is considered as immediate. However, while working at the processor level, the execution time of one instruction is measured at the nanoseconds scale.

Not considering this delay may lead to sub-optimal decisions. For example, not considering the delay for the system to handle the migration of a virtual machine may lead to repeat decisions. We argue that **developers should take into account this delay if the frequency of the monitoring stage is lower than the time of action effects to be measurable**.

1.2.3 Systems may have emergent behaviours

Smart grid behaviour is affected by several factors that cannot be controlled by the grid manager. One example is the weather conditions. Smart grids rely on an energy production that is distributed over several actors. For instance, users, who were mainly consumers before, now produce energy by adding solar panels on the roof of their houses. The production of such energy depends on the weather, and even on the season[3]. Another example is the increasing adoption of electric vehicles, which de facto drastically increase the consumption of electricity during the night. Ignoring this characteristic of adaptive systems* may result in suboptimal situations that can be understood with difficulties.

System behaviour may be emergent. [**zio2011uncertainties**] Different factors can explain this phenomenon. As for smart grids*, systems may evolve in a stochastic and uncertain environment. Or, some system, like those name ultra-large systems, are so complex that a few engineers can tame it. But, groups of engineers will have an understanding of some part of the system.

Despite the complexity, engineers still need to understand how the system is behaving or behaved. It will help them to optimise the global behaviour or to understand and repair errors. **Engineers need tooling support to trace back previous behaviours and or replay them.**

[3]The angle of the sun has an impact on the amount of energy produced by solar panels. This angle varies according to the season.

1.2.4 Different part of a system evolve at different paces

Every meter sends consumption and production data every 15 min. However, this collection is not synchronous. That is, all meters do not send their data at the same timestamp. The global system, which receives all data, has not thus a global vision with the same freshness for all the part of the grid. Electricity data are very volatile: a peak or a drop may happen in less than a minute due to, for instance, the starting or the finishing of a washing machine. When analysing the data, the process should consider this difference of paces, and estimate the evolution of data. Otherwise, they will reason over outdated data, which do not reflect the real situation. It may lead to suboptimal decisions.

Different parts of the same system may evolve at different paces. Some systems are heterogeneous in terms of hardware and software. This diversity results in different evolution or reaction paces. For example, if some components are working on batteries, they will have a sleep cycle to save energy. Contrary, if some others are running connected directly to a power source, they can react faster.

Despite this difference of paces, a global vision of a system at a precise time point may be still required. The vision should deal with data that have different freshness. In the worst case, some data would be outdated and cannot be used. **Solutions to seamlessly predict or estimate what should be the current state of these outdated elements are thus required.**

1.2.5 Evolution of systems is linked with time

Power flow is impacted by consumption and production of users, and by the modifications of the topology. Knowing the last status of the grid is as important as knowing how it evolves. Based on the evolution, the grid operator can predict any future incidents, like overloads. It could also compare this evolution of behaviour with a normal one to detect, for example, any malicious behaviour.

Evolution of systems is inherently linked with a time dimension. Evolution and time are two related concepts. For some systems, not only the last states are important but also how they evolve. Then, analysis processes will investigate this evolution to know if it is a normal one or not. They can also use this evolution to predict how systems will evolve. Based on these predictions, they can proact on future incidents in the system.

Decisions are not made upon the last state of the system but how it evolves. The analysis process should thus navigate both in the structure of the system and its behaviour over time. **Engineers need efficient tooling to structure, represent, query, and store temporal data on a large scale.**

1.3 Challenge

In this section, we present five challenges for the research community in MDE* and adaptive systems*. The last one has been published in our vision paper regarding time awareness in MDE* in [**DBLP:conf/models/Benelallam0MFBB17**].

1.3.1 Data are uncertain

enges:u-data⟩? Data become a cornerstone piece to autonomously derive decisions from them, or at least to support decision-making processes. We argue that their uncertainty will impact all the development stages of software, from the design to the execution. Design techniques should provide mechanisms to help developers abstract and manipulating uncertain data. Control flows use data, for example, in the branching conditions. This branching should be redesigned to consider the uncertainty in the data.

The literature provides approaches to help engineers reason or manipulate data uncertainty, or at least probability distributions. For example, believe functions [**shafer1992dempster**] help to reduce this uncertainty by combining several sources of data. The probabilistic programming [**DBLP:conf/icse/GordonHNR14**] community provide frameworks and languages [**url:InferNET18**; **baudin2017openturns**] to propagate probabilities through computations.

However, from the best of our knowledge, no global study has been done to evaluate the impact of data uncertainty on the development of software. The following challenge still remains an open question for the software engineering community:

$\boxed{\text{How to engineer uncertainty-aware software (design, implement, test, and validate)?}}$

1.3.2 Actions have long-term effects

enges:long-term-act⟩? Decision-making processes follow the growing complexity of software. They are more and more able to make not only decisions on the current state of the system but also its past and future ones. And this decision may also have long-term effects.

Due to this complexity, developers and users may misunderstand the decisions taken by a system. Plus, designers may neglect or underestimate the impact of a decision. Moreover, as highlighted by Bencomo *et al.,* [**DBLP:conf/iceccs/BencomoWSW12**] systems should be self-explained. They should be able to explain the decisions made.

To achieve this vision and to help designers and users understanding the impact of a decision, we argue that the software engineering community should address the following question:

How to represent, query, store, and understand the impacts of long-term actions?

1.3.3 Systems may have emergent behaviours

allenges:ermger-bhv⟩? The growing complexity of systems also has another impact: they have emergent behaviour. This behaviour may be suboptimal and hard to understand by designers, who generally have a local vision of the system.

However, when this behaviour leads to failure, engineers still need to understand why and how to avoid a novel occurrence of the problem. Plus, as the behaviour might be suboptimal, they need to optimise it.

To reach this goal, engineers need tooling support to help them in their investigation process. In other words, the research community should answer the following global challenge:

How to understand, predict, and optimise emergent behaviours?

1.3.4 Different part of a system evolve at different paces

allenges:diff-paces⟩? Systems may be heterogeneous and diverse, in terms of hardware but also software. Due to this diversity, the different part of a system may evolve at different paces.

However, engineers may need to reason on a global view. They will thus need to deal with data that have different freshness but also with components that can have their behaviour changed at different spaces. While decisions are executed to optimise the global behaviour, the system may be in an inconsistent state or a less good state than the initial one.

When designing the adaptation process, engineers need thus to consider this difference in freshness. For example, they need to implement estimation functions to estimate the value of an outdated data. Plus, they need to consider the fact that the system can be in an inconsistent state while being updated. To sum up, the software engineering community should answer the following global challenge:

How to represent, query, and store inconsistent system states and behaviours?

1.3.5 Evolution of systems are linked with time

es:evol-syst⟩? System behaviours are temporal: the execution of the different actions are made over time. Plus, the structure of a system evolves over time. Not only may the last state be important but also how it evolves. Engineers need thus to analyse this temporal evolution.

Time in software engineering is not a new challenge. For example, Riviera *et al.,* [**DBLP:conf/models/RiveraRV08**] have already identified time as a challenge for the MDE* community. Different approaches have been defined [**DBLP:conf/sle/BousseCCGB15**; **DBLP:conf/sle/KansoT12**; **DBLP:conf/icse/KoegelH10**; **DBLP:conf/seke/0001FNMKT14**]

However, we notice that modelling, persistence, and processing of data evolution remain understudied. Thomas Hartmann started addressing these challenging in his PhD thesis [**DBLP:phd/basesearch/Hartmann16**]. The final global challenge, not fully addressed, is thus:

How to structure, represent query, and store efficiently temporal data on a large scale?

1.4 Scope of the thesis

Among all the challenges described in the previous section, this thesis focus on three of them: data uncertainty (Section 1.3.1), long-term actions (Section 1.3.2), and emergent behaviours (Section 1.3.3) More precisely, we address three sub-problems of these challenges.

Managing uncertainty requires significant expertise in probability and statistics theory. The literature provides different solutions to manage uncertainty [**zadeh1996fuzzy**; **metrology2008evaluation**; **shafer1992dempster**]. The application of these techniques requires a deep understanding of the underlying theories and is a time-consuming task [**DBLP:conf/quatic/VallecilloMO16**]. Moreover, it is hard to test and perhaps most importantly, very error-prone. In this thesis, we address thus the following problem:

Sub-challenge #1: How to ease the manipulation of data uncertainty?

Adaptation processes may rely on long-term action* like resource migration in cloud infrastructure. The lack of information about unfinished actions and their expected effects on the system, the reasoning component may take repeated of sub-optimal decisions. One step for enabling this reasoning mechanism is to have an abstraction layer which can represent these long-term actions* efficiently. In this thesis, we, therefore, cope with the following challenge:

Sub-challenge #2: How to enable reasoning over unfinished actions and their expected effects?

Due to the increasing complexity of systems, developers have difficulties in delivering error-free software [**DBLP:conf/icse/BarbosaLMJ17**; **DBLP:conf/icse/MongielloPS15**; **DBLP:conf/icse/HassanBB15**]. Moreover, complex systems or large-scale systems may have emergent behaviours. Systems very likely have an abnormal behaviour that was not foreseen at design time. Existing formal modelling and verification approaches may not be sufficient to verify and validate such processes [**DBLP:conf/icse/TaharaOH1**
In such situations, developers usually apply diagnosis routines to identify the causes of the failures. During our studies, we tackle the following challenge:

Sub-challenge #3: How to diagnose the self-adaptation process?

1.5 Contribution & Validation

In this thesis, we argue that modern modelling frameworks should consider uncertainty and time as first-class concepts. In this dissertation, I present two contributions that support this vision. First, we define a language with uncertainty at a first-class citizen: Ain'tea. We detail this contribution in Chapter 5. Second, we define a meta-model, and we formalise it, of the knowledge of adaptive systems*. We present this contribution in Chapter 6.

Ain'tea: Managing Data Uncertainty at the Language Level This contribution addresses the challenge of the manipulation of uncertain data (cf. Sub-Challenge #1). We propose Ain'tea, a language able to represent uncertain data as built-in language types along with their supported operations. It contains a sampling of distributions (Gaussian, Bernoulli, binomial, Dirac delta function, and Rayleigh) that covers the different data types (booleans, numbers, and references). We implement a prototype of the language, publicly available on GitHub[4]. We use a real-world case study based on smart grid*, built with our partner Creos S.A.. It shows first that our approach does not impact the conciseness of the language. Second, it highlights the feasibility and the advantages of uncertainty-aware type checking systems on the language level.

This contribution is under submission at the JOT Journal[5]:

- **insubmission:2019:comlan:datauncertainty**, **insubmission:2019:comlan:datauncertainty**

A temporal knowledge meta-model This contribution addresses the challenge of reasoning over unfinished actions and of the understanding of adaptive system* behaviour* (cf. Sub-Challenge #2 and #3). First, we formalise the common core concepts implied in adaptation processes, also referred to as knowledge*. The formalisation is based on temporal graphs and a set of relations that trace decisions impact to circumstances. Second, we propose a framework to structure and store the state and behaviour of a running adaptive system*, together with a high-level Application Programming Interface (API)* to efficiently perform diagnosis routines. Our framework relies on a temporal model-based solution that efficiently abstracts decisions, their corresponding circumstances, and their effects. We demonstrate the applicability of our approach by applying it to a smart grid* based example. We also show that our approach can be used to diagnose the behaviour of at most the last

[4]`https://github.com/lmouline/aintea/`
[5]`http://www.jot.fm/`

five days of a district in the Luxembourg smart grid* in ∼2.4 seconds.

Part of this contribution has been published at the IEEE International Conference on Autonomic Computing[6] (ICAC) and at the ACM/SIGAPP Symposium On Applied Computing[7] (SAC):

- **DBLP:conf/sac/MoulineB0FBMB18**, **DBLP:conf/sac/MoulineB0FBMB18**[redacted]
- **DBLP:conf/icac/MoulineBFBB18**, **DBLP:conf/icac/MoulineBFBB18**[redacted]

1.6 Structure of the document

We split the remaining part of this document into eight chapters. First, Chapter 2 describes the necessary background of the thesis. We present concepts related to MDE* and adaptive systems*. Based on this background, we show the gap of the current state of the art in Chapter 3. Then, in Chapter 4, we detail the vision defended in this thesis. We present the arguments in support of it and those that are in opposition. Chapter 5 and Chapter 6 described our two contributions, which go towards the defended vision. The former details our language, Ain'tea, that integrates uncertainty as a first-class citizen. The latter explains our temporal metamodel that can represent past and ongoing decisions with their circumstances and effects. Finally, we conclude in Chapter 7, and we present a set of perspectives of our work.

[6] `http://icac2018.informatik.uni-wuerzburg.de/`
[7] `http://www.sigapp.org/sac/sac2018/`

2

Background

t:background⟩?

Contents

This chapter describes the necessary background to understand this thesis. We first describe the concepts behind adaptive systems. Then we describe the model-driven engineering methodology. Finally, we give some details about the probability theory that we use.

2.1 Adaptive systems

sec:back:adapt-syst⟩?

2.2 Model-Driven Engineering*

Abstraction, also called modelling, is the heart of all scientific discipline, including computer science [**DBLP:journals/cacm/Kramer07**]. One will abstract a system, computer or not, a problem, or a solution to reason about it for a specific purpose. Abstraction reduces the scope to relevant parts, removing the superfluous that complexity the understanding. For example, a climatologist models all elements that impact the global climate (wind, ocean current, temperature, ...), ignoring local information, like the temperature under a forest, or global one, like the solar system. In contrast, astronomers model the solar system, ignoring all information regarding the Earth climate.

In computer science, different modelling formalisms have been proposed. We can cite the entity-relationship [**DBLP:journals/tods/Chen76**] that is used to describe data model for relational databases. In the web community, they use the ontology [**DBLP:journals/ijmms/Gruber95**] formalism to define the semantic web[1] [**berners2001semantic**]. The software engineering community uses the Unified Modelling Language (UML)* [**omg2017umlspec**] to formalism software system structure or behaviour.

Extending this need for abstraction to all software engineering activities, practitioners have proposed the MDE* methodology [**DBLP:journals/computer/Schmidt06**; **DBLP:conf/ifm/Kent02**]. This methodology advocates the use of models, or abstractions, as primary software artefacts [**DBLP:journals/software/WhittleHR14**]. The contributions proposed in this thesis are in line with this vision. In the following sections, we give an overview of the MDE* methodology and how we will use it.

2.2.1 Principles and vision

Global overview Software systems tend to be more and more complex. To tame this complexity [**DBLP:conf/icse/FranceR07**; **DBLP:journals/computer/Schmidt06**].

[1]Semantic web is defined an en extension of the Web to enable the processing by machines.

the MDE* methodology suggests to use models* for all the steps of software development and maintenance [**DBLP:journals/computer/Schmidt06**; **DBLP:series/synthesis/2017Bram** **DBLP:conf/icse/HutchinsonRW11**; **DBLP:conf/uml/BakerLW05**; **DBLP:conf/icse/Hutchins** **DBLP:journals/software/AtkinsonK03a**]: design, evolution, validation, etc. The core idea of this approach is two reduce the gap between the the problem and the solution space [**DBLP:journals/computer/Schmidt06**]. Two main mechanisms have been defined: Domain Specific Modelling Language (DSML)* and model transformation. The former is based on the separation of concern principle. Each concern[2] should be addressed with a specific language, which manipulates concepts, has a type system, and a semantics dedicated to this concern. These languages allow to create and manipulate models*, specific for a domain. The latter allows engineers to automatically generate software artefacts, such as documentation, source code, or test cases. Using these mechanisms, stakeholders can focus on their own problem keeping in ming the big picture. A well known examples is the Structured Query Language (SQL) [**SQL:Spec**] Using this language, engineers can query a relational database, the data model being the model*. They don't have to consider indexes (hidden behind the concept of, for example, primary keys) or all the mechanisms to persist and retrieve data from the disk.

Advantages and disadvantages Defenders of the MDE* approach mainly highlight the benefits of abstraction in software engineering[**DBLP:journals/computer/Schmidt06**; **DBLP:conf/ifm/Kent02**; **DBLP:conf/uml/BakerLW05**]. First, using a same model*, engineers can target different specific platforms. For example, the ThingML [**DBLP:conf/models/** language allow to specify the behaviour of system through state machines. A same ThingML can be deployed on different platform such a Arduino, Raspberry Pi. Second, thanks to the transformation engine, the productivity and efficiency of developers is improved. Third, models* allow engineers to implement verification and validation techniques, like model checking [**DBLP:books/daglib/0020348**], which improve the software quality. Finally, the models* enable the separation of application and infrastructure code and the reusability of models*.

[2]The definition of concern is intentionally left undefined as it is domain specific.

However, the literature has also identified some drawbacks of the MDE* approach [**DBLP:conf/ifm/Kent02**; **DBLP:conf/uml/BakerLW05**; **DBLP:conf/models/Wh** **DBLP:conf/icse/HutchinsonRW11**]. First, it requires a big initial effort when the DSML* needs to be defined. Second, current approaches do not allow the definition of very large models*. This drawback can be mitigated with new approaches such as NeoEMF [**DBLP:conf/ecmdafa/BenelallamGSTL14**; **DBLP:journals/scp/DanielSBTVG** which enable the storage of large models*, and MogwaÃŕ [**DBLP:conf/rcis/DanielSC16**][redacted] a query engine for large models*. Third, this approach suffers of a poor tooling support, as they should be reimplemented for each model*. As for the second drawback, recent contributions try to remove this limitation. For example, we can cite the work of Bousse *et al.,* [**DBLP:journals/jss/BousseLCWB18**] that define a generic omniscient debugger[3] for DSML*. Third, introducing MDE* in a software development team introduces organisational challenges. It changes the way developers interact and work together. Finally, abstraction is a two edges sword. Indeed, reasoning at an abstract level may be more complex as some prefer to work with concrete examples and based on simulation.

Fundamentals concepts MDE* is based on three fundamentals concepts: metamodel*, model*, and model transformation. In this thesis we do not use any transformation technique. In the next section, we thus detail the concept of metamodel* and model*.

2.2.2 Metamodel*, model*

Metamodel* Metamodels* the different concepts in a domain and their relationships. They represent the knowledge of a domain, for a specific purpose [**DBLP:conf/iceccs/Bezivi** In addition, they define the semantics rules and constraints to apply [**DBLP:journals/computer/S** They can be sees as models* of models*. In the MDE* community, they are generally defined using the class diagram of the UML* specification [**omg2017umlspec**]. They, therefore, contain classes, named metaclass, and properties (attributes or

[3]Debugger, generally, allow to execute step by step a programs, that is, forward. An omniscient debugger is also able to go backward: to navigate back in the previous states of a program [**DBLP:journals/corr/cs-SE-0310016**].

references). In the language engineering domain, metamodels* are used to defined the concepts that a language can manipulate. We detail this part in **??**. Object Management Group (OMG)*[4] define a standard metamodeling architecture: Meta Object Facility (MOF)* [**MOF:Spec**]. It is defined aside with Object Constraint Language (OCL)* [**OCL:Spec**], the standard constraint language to define constraints that cannot be specified by a class diagram, and XML Metadata Interchange (XMI)* [**XMI:Spec**], the standard data format to persist (meta)models*.

add section refer

Model* Models* capture some of system characteristics into an abstraction that can be understood, manipulated, or processed by engineers or other system. They are linked to their metamodels* through the conformance relationship. A model* is conformed to exactly one metamodel* if and only if it satisfies all the rules of the metamodel*. That is, each element of a model* should instantiate at least one element of the metamodel* and respect all the semantics rules and constraints, that can be defined using OCL*. Based on these models, stakeholders can apply verification and validation techniques, such as simulation and model checking, or model transformation. France *et al.,* have identified two class of models [**DBLP:conf/icse/FranceR07**]: development and runtime models. The former are abstraction above the code level. It regroups requirements, architectural, or deployment models. The latter abstract the runtime behaviour or status of systems. They are the basis of the models@run.time* paradigm, that we detail in the next section.

2.2.3 Models@run.time

At its origins, MDE* mainly addresses the challenges of designing, validating, and maintaining complex software systems. However, these systems were facing a new challenge: the uncertainty of the environment in which they are deployed. As we detail in section 2.1, systems have been improved to enable runtime adaptation. This process can only be done they the adaptation process has a deep understanding of the structure, the behaviour, and the environment of the system. This deep understanding can be provided by a proper abstraction of these runtime elements. Extending MDE* principles and visions, the models@run.time* paradigm spans the use of models* from

[4] `https://www.omg.org/`

design time to runtime. The model, as an abstraction of a real system, can be used during runtime to reason about the state of the actual system. A conceptual link between the model and the real system allows modifying the actual system through the model and vice versa. The models@run.time paradigm uses metamodels* to define the domain concepts of a real system together with its surrounding environment. Consequently, the runtime model depicts an abstract and yet rich representation of the system context that conforms to (is an instance of) its meta-model.

2.2.4 Tooling

Tooling is an important aspect of every approach to be adopted. Development platforms allow developers to create, manipulate, and persist (meta)models* through high or low level programming interfaces. For example, a graphical language can be used for this purpose. Additionally, these tools should embed transformation engines such as a code generator.

In the MDE* community, the standard tool is the Eclipse Modelling Framework (EMF)* [**steinberg2008emf**]. It is the de-facto baseline framework to build modelling tools within the Eclipse ecosystem. It embeds its own metamodelling language, ECore [**steinberg2008emf**; **ECore:website**]. ECore is thus the standard implementation of Essential MOF (EMOF)* [**MOF:Spec**], a subset of MOF* that corresponds to facilities found in object-oriented languages. As written on the EMF*-website[5], this modelling framework "provides tools and runtime support to produce a set of Java classes for the model, along with a set of adapter classes that enable viewing and command-based editing of the model, and a basic editor".

However, as highlighted by Fouquet *et al.,* [**DBLP:journals/corr/FrancoisNMDBPJ14**; **DBLP:conf/models/FouquetNMDBPJ12**], models generated by EMF have some limitations, which prevent their use for the models@run.time* paradigm. The models@run.time* can be used to implement an adaptive Internet of Things (IoT)* systems[6]. These systems contain small devices, like micro-controllers, that have limited

[5]`https://www.eclipse.org/modeling/emf/`

[6]Definition of IoT* by Gubbi *et al.,* [**DBLP:journals/fgcs/GubbiBMP13**]: "Interconnection of sensing and actuating devices providing the ability to share information across platforms through a unified framework, developing a common operating picture for enabling innovative applications."

memory, disk space, and process capacity. If one wants to deploy a model on it, thus it should have a low memory footprint, a low dependency size, a thread safety capacity, en efficient model (un)marshalling and cloning, a lazy loading mechanism, and a compatibility with a standard design tool, here EMF*. However, the approaches at this time failed to tame the first four requirements. They, therefore, define Kevoree Modelling Framework (KMF)* [**DBLP:journals/corr/FrancoisNMDBPJ14**; **DBLP:conf/models/FouquetNM**
a modelling framework specific for the models@run.time* paradigm.

Hartmann extended this work and created the GreyCat Modelling Environment (GCM)*[7]. Using this environment, a developer can create high-scalable models, designed for models@run.time*[**DBLP:conf/seke/0001FNMKT14**; **DBLP:conf/models/Moawad0FN**
with time as a first-class concept. All metaclasses have a default time attribute to represent the lifetime of the model element that instantiates it. The created models are object graphs stored on a temporal graph database, called GreyCat[8] [**DBLP:journals/is/HartmannFMRT**
DBLP:phd/basesearch/Hartmann16]. In addition to time, metamodels* defined by GCM* can have attribute with a value computed from a machine learning algorithm [**DBLP:journals/sosym/0001MFT19**]. Based on the metamodel* definition, a Java and Javascript API* are generated, to manipulate the model, *i.e.,* the temporal object graph.

2.2.5 Modelling in the context of this thesis

The models@run.time* paradigm is a well-know approach to handle the challenges faced in adaptive system* development. Plus, the GCM* has been designed to design a data model, with time as first-class concept. In this thesis, we will use this tool to define a metamodel* to abstract the knowledge of adaptive systems* (cf. Chapter 6).

2.3 Software Language Engineering*

As stated by Favre *et al.,* [**DBLP:conf/sle/FavreGLP10**], software languages are softwares. As traditional software, they need to be designed, tested, deployed, and maintained. These activities are grouped under the term Software Language

[7]`https://github.com/datathings/greycat/tree/master/modeling`
[8]`https://greycat.ai/`

Engineering (SLE)* [**kleppe2008software**]. Before explaining the role of software languages in this thesis, we will first define them in this section.

2.3.1 Software Languages

Most of, not to say all of, developers have used, at least once, a programming language to develop a software. For example, one may use Javascript to implement client-side behaviour of a web site and another one C to implement a driver. We can distinguish another kind of languages, named modelling languages. Those models allow developers to implement a model. For example, we can argue that many developers have already used the HTML language to implement a Document Object Model (DOM)*[9] With the emergence of executable models[10], the difference between models and programs are more and more blurry. So it is for the difference between programming and modelling language. Hence, Annake Kleppe uses the term software language* to combine both kind of languages.

Another way to classify software languages* is by their scope [**DBLP:journals/sigplan/Deurse** General Purpose Language (GPL)* are languages that can be used for any domain whereas Domain Specific Language (DSL)*[11] that are restricted to a specific domain. Using a GPL*, a developer can use it to implement a full software are benefits from great tooling support. However, she may manipulates concepts that are different from those of the problem space. For example, implementing an automatic coffee machine in Java, she will have to manipulate the concept of class, object, functions. In contrary, DSL* offer a language close to their problem domain [**DBLP:journals/smr/DeursenK98**] but might suffer of a poor tooling support [**voelter2014generic**]. As for MDE*, there is some research efforts to remove this disadvantage [**DBLP:journals/jss/BousseLCWB18**]. Using DSLs*, developers can have simpler code, easier to understand and maintain [**DBLP:journals/sigplan/DeursenKV00**; **DBLP:journals/smr/DeursenK98**]

Software languages* are composed of two parts [**DBLP:journals/computer/HarelR04**]

[9]"The DOM* is a platform -and language- neutral interface that will allow programs and scripts to dynamically access and update the content, structure and style of [web] documents." [**DOM:Spec**]

[10]Executable models are models that have a semantics attached to their concepts.

[11]In this thesis, we do not make the difference between DSML* and DSL*

a syntax and a semantics. The syntax defines the element allowed in the language and the semantics their meaning. We can distinguish two kind of syntax: the abstract one and the concrete one. The abstract one define the different concepts manipulated with the language and their relationships. It can be expressed by a metamodel*/ The concrete abstract defines how these concepts are represented. It exists two categories of concrete syntax: graphical and textual. As for the syntax, there is two kinds of semantics: the static and the dynamic. Static semantics defines the constraints of the abstract syntax that cannot be directly expressed in the formalism chosen. For example, the static semantics will define uniqueness constraint for some elements or will forbid any cycle in dependencies (if any). The type system usually goes in part of the static semantics. The dynamic semantics defines the behaviour of the language.

When designing a language, engineers can start with the abstract or with the concrete abstract. In the first case, we say that they define the language in a model*-first way. Others will prefer to start by the concrete syntax, and thus they are using a grammar-first approach. Some tools, like XText [**DBLP:conf/oopsla/EysholdtB10**] [redacted] can generate automatically the model from the grammar.

2.3.2 SLE* in this thesis

In our vision, we argue that uncertainty should be considered as a first-class citizen for modelling frameworks. This modelling framework should allow the definition of metamodels* with uncertainty management capacities. As described in the previous sections, these metamodels* can correspond to the abstract syntax of a language. As a contribution, we, therefore, propose a language with uncertainty as a first-class citizen: Ain'tea(*cf.* Chapter 5).

2.4 Probability theory

Data uncertainty, and more generally uncertainty, is linked with confidence. Indeed, the confidence level that one can give to a data depends on its uncertainty. The more uncertain value is, the less trust it can be placed in it. However, it is rather difficult to put exact numbers on this confidence level. A strategy used by experts to formalise this confidence level, and thus the uncertainty, is to use probability distributions.

In this section, we thus introduce the necessary concepts of these distributions. We first describes the concept of random variables in the probability theory. Then we describe what a probability distribution is and the properties that will be used in this thesis. Finally, we will describe the distributions used in this thesis.

2.4.1 Random variables

Definition The three base elements of the probability theory are: an outcome, an event, and a sample space. An outcome is the occurrence of a random phenomenon. An event is a set of outcomes and a sample space S is the set of all possible outcomes. Let us take an example: the rolling of two dices. i denotes the result of one die and j the result of the other. An example of an outcome is the result of a roll, like the pair (2, 6). An event could contain all outcomes where both dices have an even result: $E = \{(i, j) \mid i, j \in \{2, 4, 6\}\}$. The sample space of our example is this equals to: $S_1 = \{(i, j) \mid i, j \in \{1, 2, \ldots, 6\}\}$. We thus have:

A random variable is a function from the sample space S to $\mathbb{R}$. In our example, we can define two random variables X and Y that represent, respectively, the minimum of the two dices and the sum of the two. We have thus: $X = min(i, j)$ and $Y = i + j$.

Discrete and continuous Random variable can be either discrete or continuous. For discrete random variables, we can list all the events of the sample space, whereas we cannot for continuous ones. For example, it is possible to list all result of the rolling of n dices, $n \in \mathbb{N}$. But, we cannot list all possible temperatures of a room (if we consider that we have an infinite precision).

Independence and disjoint Independence and joint are defined at the event level. Two event are independents if the probability of occurrence of one does not impact the probability of occurrence of the others. Two events are disjoints if and only if they do not share any outcome. Mathematically speaking, events A and B are independent if and only if $P(A \cap B) = P(A) * P(B)$, and they are disjoint if and only if $P(A \cap B) = P(A) * P(B)$.

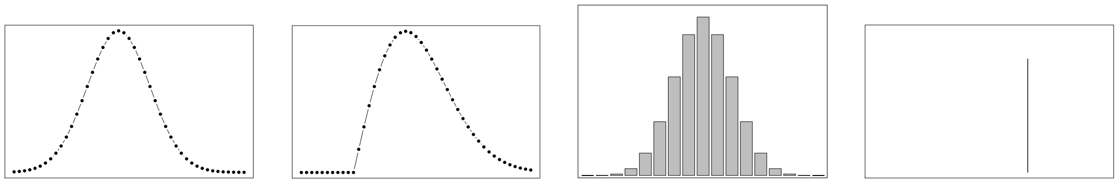

(a) Gaussian (b) Rayleigh (c) binomial (d) Dirac delta function

Figure 2.1: Probability distributions used in this thesis

oba:examples⟩?

2.4.2 Distribution

A probability distribution of a random variable X is a function that gives the probability that X takes on the values x, for all possible values of the sample space. Here, we can distinguish two cases: discrete and continuous distributions. A distribution is said discrete when it is based on a discrete random variable, and continuous when the random variable is continuous.

The difference that will interest us in this thesis between the two is how they compute the probability, and thus the confidence level. For discrete distributions, the probability that $X = x$ corresponds to the result of the function. By definition, the probability that the random variable is inferior to a vale x is thus equals to the sum of probability for $X < x$.

Contrary, for continuous distribution, the probability is mapped to the area under the function that which defines the probability distribution, *i.e.,* the integral of the function. For example, the confidence that a given uncertain value is greater than zero is the surface under the function $f(x)$ for any $f(x) \mid f(x) > 0$. As the area under a precise point is null, the probability that a continuous random variable equals x is zero.

2.4.3 Distribution used in this thesis

Probability distributions have proven their ability to represent uncertain data. Among the existing distributions, in this thesis we use on five of them: Gaussian (or Normal) distribution, Rayleigh distribution, binomial distribution, Bernoulli distribution and the Dirac delta function. The Gaussian and the Rayleigh distributions will represent continuous distributions. Discrete distributions are represented by the Bernoulli and the binomial one. Finally, the Dirac delta function can represent the confidence level of exactly one value.

Bernoulli distribution Bernoulli distribution represents the distribution of a binary phenomenon. One well-known example is the flip of a fair coin. Bernoulli(p) denotes a random variable that equals 1 with a probability of p and 0 with a probability of $(1 - p)$. Following the coin example, 1 can represent head and 0 tail.

Binomial distribution The binomial distribution represents the probability of success of a binary phenomenon over a set of trials. Figure 2.1c illustrates an example of a Binomial distribution. It is defined by two parameters: n and p. n is the number of trials done and p the probability of success. By definition, it is defined over a discrete domain, more specifically on the set of natural number $\mathbb{N}$. In this case, the confidence level corresponds to the values of the function.

It has been proven that this distribution is similar to a Gaussian distribution in certain cases [**box2005**]. If the domain definition can be changed from discrete to continuous, a binomial distribution can be approximated by a Gaussian distribution.

Gaussian (or Normal) distribution The Gaussian distribution, commonly referred to as normal distribution, is the most general probability distribution. For example, the International Bureau of Weights and Measures encourages the use of this distribution to quantify the uncertainty of measured values [**metrology2008evaluation**]. It is defined by two parameters: a mean and a variance. The distribution is defined on a continuous domain: $]-\infty; +\infty[$. An example of this distribution is illustrated in Figure 2.1a.

Rayleigh distribution Another distribution defined on a continuous domain ($[0; +\infty[$) is the Rayleigh distribution. It is often used for GPS positions [**bornholt2013abstraction**
This distribution is defined using a unique parameter: a variance. We depict an example of this distribution in Figure 2.1b.

Dirac delta function The Dirac delta function is defined as a probability function with $f(x) = +\infty$ for $x = 0$ and $f(x) = 0$ for all the other points. To represent other values than zero, the following variable substitution can be used $x = x - a$, where. We call a the shifting value. By definition, the integral of this function on the whole domain definition is equal to 1. As it is considered as a continuous distribution, confidence is mapped to the integral of the function. By applying a coefficient, we can modify the value of this integral, *e.g.,* to have 0.8 as the integral. We define this probability distribution using two parameters a coefficient and a shifting value. An example of this probability distribution is shown in Figure 2.1d. Conventionally, the Dirac function is represented as a vertical line that stops at the coefficient. The figure shows a Dirac function with a coefficient of 0.8 and a shifting value of 3. As it is defined on a single value, this distribution can be used for any numbers. This distribution can be used to represent uncertainty that is due to human errors.

3

State of the art

?⟨chapt:sota⟩?

Contents

This chapter reviews work related to the one presented in this dissertation.

3.1 Introduction

Delayed actions General research question:

RQ1: Do current state of the art solutions allow modelling and reasoning over delayed actions?

Sub research questions:

- **RQ1.1**: How current approaches model the evolution of the context and/or the evolution of the behaviours of systems over time?
- **RQ1.2**: Do these solutions model actions, their circumstances and their effects?
- **RQ1.3**: What are the solutions that enable the reasoning over the evolving context and/or behaviours of systems?

Uncertainty Snowballing approach [**DBLP:conf/ease/Wohlin14**]

Inclusion criteria

- **IC1**: The paper has been published before the May 31 2019
- **IC2**: The paper is available online and written in English
- **IC3**: The paper describes a modelling approach that abstract the context or behaviour of a system or an approach that enables to reason or navigate through a temporal model.

Exclusion criteria

- **EC1**: The paper has at most 4 pages (short paper).
- **EC2**: The paper presents a work in progress (workshop papers), a poster, a vision, a position, doctoral studies or the paper is a Bachelor, Master or PhD dissertation.
- **EC3**: The paper describes a secondary study (*e.g.,* literature reviews, lessons learned).
- **EC4**: The document has not been published in a venue with a peer-review process. For example, technical and research report or white papers.
- **EC5**: The document is an introduction to the proceedings of a venue or a special issue or it is a guest paper.

However, the references of papers rejected are considered for the snowballing iteration.

4

A unified modelling framework

chapt:vision⟩?

As argued in the introduction section, adaptation combined with abstraction comes with a novel challenge: the representation of uncertain data and delayed action. In this chapter, we present a vision concerning a modelling framework that should consider uncertainty and time as first-class concepts. We first argue towards this vision. Before detailing the synthesis of this vision, we show arguments against this vision. To do so, we answered with pros and cons to three questions: why do we need a modelling framework? why time and uncertainty should be put as first-class concepts? why should time and uncertainty combined in the same structure?

5

AinâĂŹtea: Managing Data Uncertainty at the Language Level

chapt:aintea⟩?

After identifying and discussing the key concepts associated with data uncertainty, this chapter presents Ain'tea, a language that integrates them directly into the grammar, type system and semantics part. To validate and exemplify our approach, we apply it to a smart-grid scenario and compare it to framework-based approaches. We show that developers benefit from the language semantics and type system which guide them to manipulate uncertain data without deep probability theory knowledge.

6

A temporal knowledge meta-model to represent, reason and diagnose decisions, their circumstances and their impacts

?⟨chapt:tkm⟩?

Contents

In this chapter, we first propose a knowledge formalism to define the concept of a decision. Second, we describe a novel temporal knowledge model to represent, store and query decisions as well as their relationship with the knowledge (context, requirements, and actions). We validate our approach through a use case based on the smart grid at Luxembourg. We also demonstrate its scalability both in terms of execution time and consumed memory.

6.1 Introduction

Adaptive systems have proven their suitability to handle the increasing complexity of systems and their ever-changing environment. To do so, they make adaptation decisions, in the form of actions, based on high-level policies. For instance, the OpenStack Watcher project [**OpenStack:Watcher:Wiki**] implements the MAPE-k loop to assist cloud administrators in their activities to tune and rebalance their cloud resources according to some optimization goals (e.g., CPU and network bandwidth). For readability purpose, we refer to adaptation decision as decision in the remaining part of this document.

Despite the reactivity of adaptation processes, impacts of their decisions can be measurable long after they have been taken. We identified two problematics caused by this difference of paces:

- How to diagnose the self-adaptation process?
- How to enable reasoning over unfinished actions and their expected effects?

To address them, we propose a temporal knowledge model which can trace decisions over time, along with their circumstances and effects. By storing them, the adaptation process could consider ongoing actions with their expected effects, also called impacts. Plus, in case of faulty decisions, developers may trace back their effects to their circumstances. Our current approach is limited to the representation of measurable effects of any decision, and therefore action.

The meta-model allow structuring and storing the state and behaviour of a running adaptive system, together with a high-level API to efficiently perform diagnosis routines. Our framework relies on a temporal model-based solution that efficiently abstracts decisions and their corresponding circumstances. Specifically, based on existing approaches for modelling and monitoring adaptation processes, we identify a set of properties that characterise context, requirements, and actions in self-adaptive systems. Then, we formalise the common core concepts implied in adaptation processes, also referred to as knowledge, by means of temporal graphs and a set of relations that trace decisions impact to circumstances. Finally, thanks to exposing common interfaces in adaptive processes, existing approaches in requirements and goal modelling engineering can be easily integrated into our framework.

The rest of this chapter is structured as follows. In the remaining part of this section, we motivate our approach, we summarise core concepts manipulated in adaptation processes, and we present a use case scenario based on the Luxembourg Smart Grid (*cf.* Chapter). Then, we provide a formal definition of these concepts in Section 6.2. Later, we describe the proposed data model in Section 6.3. In Section 6.4, we demonstrate the applicability of our approach by applying it to the smart grid example. We conclude this chapter in Section 6.5.

add ref

Update with last version

6.1.1 Motivation

m:intro:motiv⟩

Delayed action

elayed_action⟩

In this section, we motivate the need to reason over delayed actions. To do so, we first give four examples of these actions. Then we detail why the effects of actions should be considered. Finally, we summarise and motivate the need for incorporating actions and their effects on the knowledge.

Delayed action examples Until here, we have claimed that adaptation processes should handle delayed actions. In order to show their existence, we give four different examples: two based on our use case, one on cloud infrastructure and a last one on smart homes. From our understanding, three phenomena can explain this delay: the time to execute an action(s) (Example 1), the time for the system to handle the new configuration (Example 3) and the inertia of the measured element (Example 2 and 4).

Example 1: Modification of fuse states in smart grids Even if the Luxembourg power grid is moving to an autonomous one, not all the elements can be remotely controlled. One example is fuses that still need to be open or closed by a human. Open and close actions in the Luxembourg smart grid both imply technicians who are contacted, drive to fuse places and manually change their states. If several fuses need to be changed due to one decision, only one technician will drive to them, sequentially, and executes the modifications. For example, in our case, our industrial partner asks us to consider that each fuse modification takes 15 min whereas any

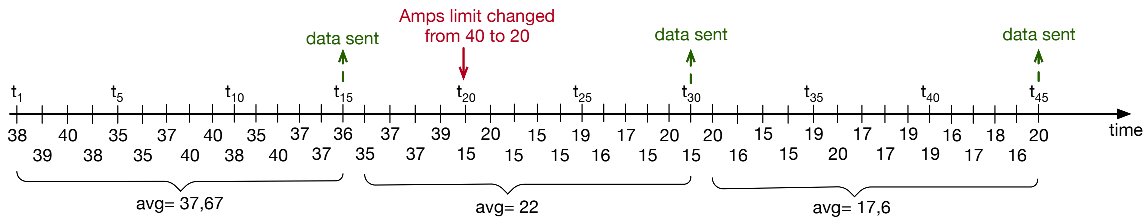


Figure 6.1: Example of consumption measurement before and after a limitation of amps has been executed at t_{20}.

ng-action-amps-limit⟩

incident should be detected in the minute. Let's imagine that an incident is detected at 4 p.m. and can be solved by modifying three fuses. The incidents will be seen as resolved by the adaptation process at 4 p.m. + 15 min * 3 = 4:45 p.m. In this case, the delay of the action is due to the execution time that is not immediate.

Example 2: Reduction of amps limit in smart grids[1] In its smart grid project, Creos S.A.envisages controlling remotely amps limits of customers. Customers will have two limits: a fixed one, set at the beginning, and a flexible one, remotely managed. The action to remotely change amps limits will be performed through specific plugs, such as one for electric vehicles. Even if the action is near instant, due to how power consumption is collected, its impacts would not be visible immediately. Indeed, data received by Creos S.A.corresponds to the total energy consumed since the installation. From this information, only the average of consumed data for the last period can be computed.

In Figure 6.1, we depict a scenario that shows the delay between the action is executed and the impacts are measured. Each time point represents one minute, with the consumption at this moment.

Let's imagine a customer who has his or her limit set to 40 amps[2] and consumes near this limit. We consider that data are sent every 15 min. After receiving data sent t_{15} and processing them, the adaptation process detects an overload and decides

[1]This example is based on randomly generated data. As this action is not yet available on the Luxembourg smart grid, we miss real data. However, it reflects an hypothesis shared with our partner.

[2]The user cannot consume more than 40 amps at a precise time t_i.

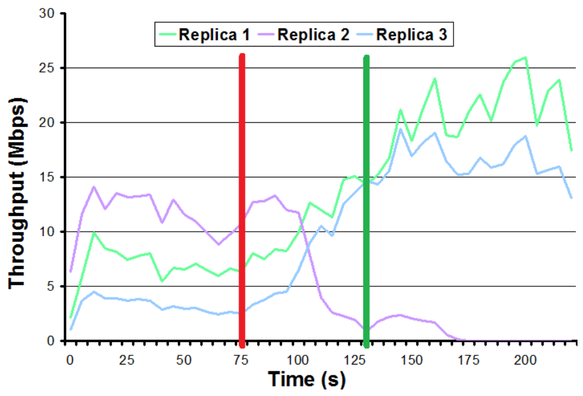


Figure 6.2: Figure extracted from [**DBLP:conf/nsdi/WangBR11**]. The red bar depicted the moment when Replica 2 stop receiving new connections. The green one represents the moment where all the rules in the load balancer stop considering R2. Despite these two actions, the throughput of the machine does not drop to 0 due to existing and active connections.

load-balencer⟩

to reduce the limits to 20 amps for the customer. However, considering the delay for data to be collected and the one to send data[3], the action is received and executed at t_{20}. At t_{30}, new consumption data is sent, here equals 22 amps. Here, there are two situations. First, this reduction was enough to fix the overload. Even in this idealistic scenario, the adaptation process must wait at worst 15 min (t_{30} - t_{15}) to see the resolution (without considering the communication time). Second, this reduction was not enough - as the adaptation process considered that the consumption data will be at worst 20 amps and here it is 22. Before seeing the incident as solved and knowing that the decision fixed the incident, the adaptation process should wait for new data, sent at t_{45}, *i.e.*, around 30 min ($t_{45} - t_{15}$) after the detection.

In this case, the delay of this action can be explained by the inertia in the average of the consumption.

Example 3: Switching off a machine from a load balancer An example based on cloud infrastructure of delayed actions is to remove a machine from a load balancer, for example during a scale down operation. Scale down operations

[3]Reminder: the smart grid is not built upon a fast network such a fiber network.

allows cloud managers to reduce allocated resources for a specific task. It is used either to reduce the cost of the infrastructure or to reallocate them to other tasks. In [**DBLP:conf/nsdi/WangBR11**], Wang *et al.,* present a load-balancing algorithm. In their evaluation, they present the figure depicted in Figure 6.2 that shows the evolution of the throughput after the server Replica 2 (R2) is removing from the load balancer. The red bar shows the moment where R2 stop receiving new connection and the green the moment where it is removed from the load balancer algorithm. However, despite these actions have been taken, R2 should finish the ongoing tasks that it is executing. This explains why the throughout is progressively decreasing to 0 and there is a delay of around 100s between the red bars and the moment where R2 stop being active.

This example shows a delayed action due to the time required by the system to handle the new configuration.

Example 4: Modifying home temperature through a smart home system Smart home systems have been implemented in order to manage remotely a house or to perform automatically routines. For example, it allows users to close or open blinds from their smartphones. Based on instruction temperatures, smart home systems manage the heating or cooling system to reach them at the desired time. However, heating or cooling a house is not immediate, it can take several hours before the targeted temperature is reached. Plus, if the temperature sensor and the heating or cooling system are not placed nearby, the new temperature can take time before being measured. This can be explained due to the temperature inertia plus the delay for the temperature to be propagated.

Through these four examples, we show that delayed actions can be found in different kinds of systems, from CPS to cloud infrastructure. However, not only knowing that an action is running is important but also knowing its expecting effect. We detail this point in the following section.

The need to consider effects In the previous section, we show the existence of delayed actions. One may argue that action statuses are already integrated into the knowledge. For example, the OpenStack Watcher framework stores them in a database[4], accessible through an API. However, for the best of our knowledge Watcher does not store the expecting effects of each action. While the adaptation process knows what action is running, it does not know what it should expect from them.

Considering our example based on the modification of fuses, if the system knows that the technician is modifying fuse states, it does not know what would be the effects. In this case, when the adaptation process analyses the system context it may wonder: what will be the next grid configuration? How the load will be balanced? Will the future configuration fix all the current incidents? If the effects are not considered by the adaptation process, then it may take suboptimal decisions.

Let's exemplify this claim through a scenario based on the fuse example (*cf.* Example 1). As explained before, the overload detected at 4 p.m. takes around 45 min to be fixed. The system marks this incident as "being resolved". In addition to this information, the knowledge contains another one saying that it is being solved by modifying three fuses. However, during the resolution stage, a cable is also being overloaded. The adaptation process has two solutions. It can either wait for the end of the resolution of the first incident to see if both overloaded elements will be fixed or it takes other actions without considering the ongoing actions and their impacts. Applying the first strategy may make the resolution of the second incident late, whereas the second one may generate a suboptimal sequence of actions. For example, the second modifications may undo what has been done before or both actions may be conflicting.

Conclusion Actions, like fuse modification in a smart grid or removing a server from a load balancer, generated during by adaptation processes could take time upon completion. Moreover, the expected effects resulting from such action is reflected in the context representation only after a certain delay. One used workaround is the selection, often empirically, of an optimistic time interval between two iterations of the MAPE-K loop such that this interval is bigger than the longest action execution time. However, the time to execute an action is highly influenced by system overload or failures, making such empirical tuning barely reliable. We argue that by enriching

[4]`https://docs.openstack.org/watcher/latest/glossary.html#watcher-database-definition`

context representation with support for past and future planned actions and their expected effects over time, we can highly enhance reasoning processes and avoid empirical tuning.

The research question that motivates our work is thus: how to enable reasoning over unfinished actions and their expected effects?

Fined and rich context information directly influences the accuracy of the actions taken. Various techniques to represent context information have been proposed; among which we find the models@run.time [**DBLP:journals/computer/MorinBJFS09**; **DBLP:journals/computer/BlairBF09**]. The models@run.time paradigm inherits model-driven engineering concepts to extend the use of models not only at design time but also at runtime. This model-based representation has proven its ability to structure complex systems and synthesize its internal state as well as its surrounding environment.

Diagnosis support

Faced with growingly complex and large-scale software systems (e.g. smart grid systems), we can all agree that the presence of residual defects becomes unavoidable [**DBLP:conf/icse/BarbosaLMJ17**; **DBLP:conf/icse/MongielloPS15**; **DBLP:conf/ics**
Even with a meticulous verification or validation process, it is very likely to run into an unexpected behaviour that was not foreseen at design time. Alone, existing formal modelling and verification approaches may not be sufficient to anticipate these failures [**DBLP:conf/icse/TaharaOH17**]. As such, complementary techniques need to be proposed to locate the anomalous behaviour and its origin in order to handle it in a safe way.

As there might be many probable causes behind an abnormal behaviour, developers usually perform a set of diagnosis routines to narrow down the scope or origin of the failure. One way to do so is by investigating the satisfaction of its requirements and the decisions that led to this system state, as well as their timing [**DBLP:conf/iceccs/BencomoWSW12**]. In this perspective, developers may set up a set of systematic questions that would help them understand why and how the system is behaving in such a way. These questions may comprise:

- what goal(s) the system was trying to reach by executing a tactic a?
- what were the circumstances used by a decision d and its expected impact on the context?
- what decision(s) influenced the system's context at a time t?

Bencomo *et al.,* [**DBLP:conf/iceccs/BencomoWSW12**] argue that comprehensive explanation about the system behaviour contributes drastically to the quality of the diagnosis, and eases the task of troubleshooting the system behaviour. To enable this, we believe that adaptive software systems should be equipped with traceability management facilities to link the decisions made to their **(i) circumstances, that is to say, the history of the system states and the targeted requirements, and (ii) the performed actions with their impact(s) on the system**. In particular, an **adaptive system should keep a trace of the relevant historical events**. Additionally, it should be able to **trace the goals intended to be achieved by the system to the adaptations and the decisions that have been made, and vice versa**. Finally, in order to enable developers to interact with the system in a clear and understandable way, appropriate abstraction to **enable the navigation of the traces and their history should also be provided**. Unfortunately, suitable solutions to support these features are under-investigated.

Existing approaches [**hassel13**; **DBLP:conf/models/HeinrichSJRMHRP14**; **DBLP:conf/icac/EhlersHWH11**; **DBLP:conf/icse/MendoncaAR14**; **DBLP:conf/icse/Casanov** **DBLP:conf/icse/IftikharW14a**] are accompanied by built-in monitoring rules and do not allow to interact with the underlying system in a simple way. Moreover, they do not keep track of historical changes as well as causal relationships linking requirements to their corresponding adaptations. Only flat execution logs are stored.

6.1.2 Use case scenario

:tkm:intro:uc⟩ In order to provide a readable and understandable example of the formalism, we give a simplified version of the use case presented in Section **??**.

Excerpt of a smart grid Figure 6.3 shows a simplified version of a smart grid with one substation, one cable, three smart meters and one dead-end cabinet. Both the substation and the cabinet have one fuse each. The meters regularly send con-

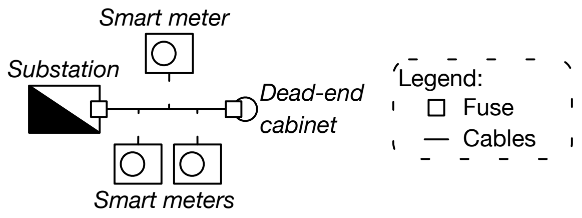


Figure 6.3: Simplified version of a smart grid

⟨fig:tkm:excerptSG⟩

sumption data at the same timestamp. For this example, we consider one requirement: minimizing the number of overloads. To achieve so, among the different actions, two actions are taken into account in this example: decreasing or increasing the amps limits of smart meters.

Adaptation scenario The system starts at t_0 with the actions, the requirements and all element of the context that remain fixed: the grid installation. Meters send their values at t_1, t_2 and t_3. Based on these data, the load on cables and substation is computed. On t_1, an overload is detected on the cable, which breaks the requirement. At the same time point, the system decides to reduce the load of all smart meters. The impact of these actions will be measured at t_2 and t_3, *i.e.,* the consumption will slowly reduce until the cable is no longer overloaded from t_3.

Diagnosis scenario As all adaptive systems, smart grids are prone to failures [**DBLP:conf/smartgridsec/0001FKNT14**]. Using our approach, an engineer could diagnose the system, and determine the adaptation process responsible for this failure. For instance, considering some reports about regular power cuts during the last couple of days, in a particular area, a stakeholder may want to interrogate the system and determine what past decision(s) have led to this suboptimal state. More concretely, he will ask: did the system make any decisions that could have impacted the customer consumption? If so, what goal(s) the system was trying to reach and what were the values used at the time the decision(s) was(were) made?

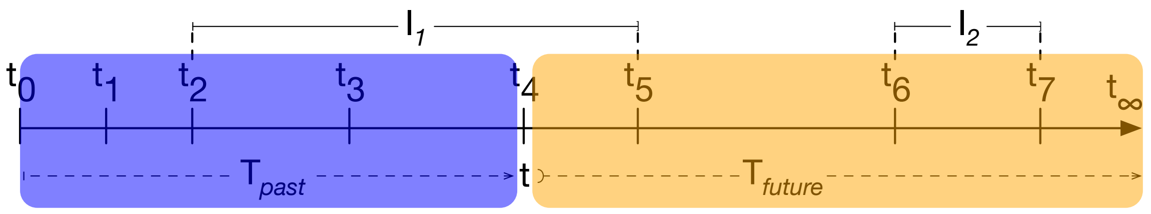


Figure 6.4: Time definition used for the knowledge formalism

ormalismeTime⟩

6.2 Knowledge formalization

m:k-formalism⟩

As discussed previously, we consider knowledge* to be the association of context* information, requirements*, and action* information, all in one global and unified model. While context* information captures the state of the system environment and its surroundings, the system requirements* define the constraints that the system should satisfy along the way. Actions*, on the other hand, are meant to reach the goals of the system.

In this section, we provide a formalization of the knowledge* used by adaptation processes based on a temporal graph. Indeed, due to the complexity and interconnectivity of system entities, graph data representation is an appropriate way to represent the knowledge*. Augmented with a temporal dimension, temporal graphs are then able to symbolize the evolution of system entities and states over time. We benefit from the well-defined graph manipulation operations, namely temporal graph pattern matching and temporal graph relations to represent the traceability links between the decisions* made and their circumstances*.

Before describing this formalism, we describe the semantics used for the temporal axis. Then, we exemplify the knowledge formalism using the Luxembourg smart grid use case, detailed in Section 6.1.2.

6.2.1 Formalization of the temporal axis

ism:timeAxis⟩?

The formalism described below has been made with two goals in mind. First, the definition of the time space should allow the distinction between past and future. Doing this distinction enable the differentiation between measured data and predicted (or planned data). Second, it should permit the definition of the life cycle of an element of the knowledge*, which can be seen as a succession of states with a validity period that should not overlap each other.

Time space T is considered as an ordered discrete set of time points non-uniformly distributed. As depicted in Figure 6.4, this set can be divided into 3 different subsets $T = T_{past} \cup \{t\} \cup T_{future}$, where:

- T_{past} is the subdomain $\{t_0;t_1;...;t_{current-1}\}$ representing graph data history starting from t_0, the oldest point, until the current time, t, excluded.
- $\{t\}$ is a singleton representing the current time point
- T_{future} is subdomain $\{t_{current+1};...;t_{\infty}\}$ representing future time points

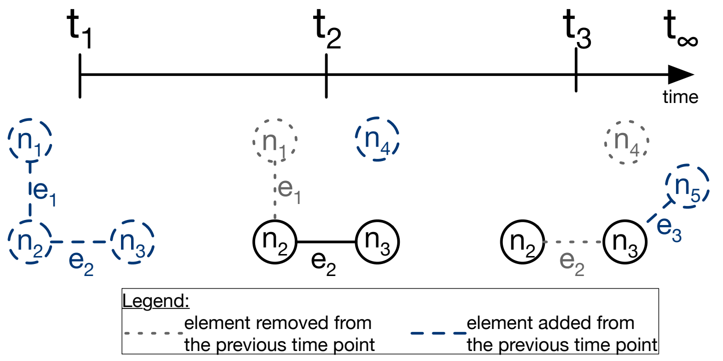
The three domains depend completely on the current time {t} as these subsets slide as time passes. At any point in time, these domains never overlap: $T_{past} \cap \{t\} = \emptyset$, $T_{future} \cap \{t\} = \emptyset$, and $T_{past} \cap T_{future} = \emptyset$. The definition of these three subsets reaches the first goal.

In addition, there is a right-opened time interval $I \in T \times T$ as $[t_s, t_e)$ where $t_e - t_s > 0$. In English words, it means that the interval should represent at least one time point and should follow the time order. For any $i \in I$, $start(i)$ denotes its lower bound and $end(i)$ its upper bound. As detailed in Section 6.2.2, these intervals are used to define the validity period for each node of the graph (our second goal).

Figure 6.4 displays an example of a time space $T_1 = \{t_0, t_1, t_2, t_3, t_4, t_5, t_6, t_7\}$. Here, the current time is $t = t_4$. According to the definition of the past subset (T_{past}) and the future one (T_{future}), there is: $T_{past1} = \{t_0, t_1, t_2, t_3\}$ and $T_{future1} = \{t_5, t_6, t_7\}$. Two intervals have been defined on T_1, namely I_1 and I_2. The first one starts at t_2 and ends at t_5 and the last one is defined from t_6 to t_7. As shown with I_1, an interval could be defined on different subsets, here it is on all of them (T_{past}, t, and T_{future}).

6.2.2 Formalism

-formalism:formalism⟩ **Graph definition** First, let K be an adaptive process over a system knowledge* represented by a graph such as $K = (N, E)$, comprising a set of nodes N and a set of edges E. Nodes represent any element of the knowledge (context, actions, *etc.*) and edges represent their relationships. Nodes have a set of attribute values:



t1
t2
t3
t∞
time
n1
e1
n2
e2
n3
n1
e1
n4
n2
n3
e2
n4
n5
e3
n2
e2
n3
Legend:
element removed from the previous time point
element added from the previous time point

Figure 6.5: Evolution of a temporal graph over time

km:validityEx⟩

$\forall n \in N, n = (id, P)$, where P is the set of key-value attributes. An attribute value has a type (numerical, boolean, ...). Every relationship $e \in E$ can be considered as a couple of nodes with a label $(n_s, n_t, label) \in N \times N$, where n_s is the source node and n_t is the target node.

Adding the temporal dimension In order to augment the graph with a temporal dimension, the relation V^T is added. So now the knowledge K is defined as a temporal graph such as $K = (N, E, V^T)$.

A node is considered valid either until it is removed or until one of its attributes value changes. In the latter case, a new node with the updated value is created. Whilst, an edge is considered valid until either its source node and target node are valid, or until the edge itself is removed. Otherwise, nodes and edges are considered invalid. The temporal validity relation is defined as $V^T : N \cup E \rightarrow I$. It takes as a parameter a node or an edge ($k \in N \cup E$) and returns a time interval ($i \in I$, *cf.* Section **??**) during which the graph element is valid.

Figure 6.5 shows an example of a temporal graph K_1 with five nodes (n_1, n_2,n_3, n_4, and n_5) and three edges (e_1, e_2, and e_3) over a lifecycle from t_1 to t_3. In this way, K_1 equals $(\{n_1, n_2, n_3, n_4, n_5\}, \{e_1, e_2, e_3\}, V_1^T)$. Let's assume that the graph is created at t_1. As n_1 is modified at t_2, its validity period starts at t_1 and ends at t_2: $V_1^T(n_1) = [t_1, t_2)$. n_2 and n_3 are not modified; their validity period thus starts at t_1 and ends at t_∞: $V_1^T(n_2) = V_1^T(n_3) = [t_1, t_\infty)$. Regarding the edges, the first one, e_1, is between n_1 and n_2 and the second one, e_2 from n_2 to n_3. Both are created at t_1. As n_1 is being modified at t_2, its validity period goes from t_1 to t_2: $V_1^T(e_1) = [t_1, t_2)$. e_2 is deleted at t_3. Its validity period is thus equal to: $V_1^T(e_2) = [t_1, t_3)$.

Lifecycle of a knowledge element One node represents the state of exactly one knowledge element during a period named the validity period. The lifecycle of a knowledge element is thus modeled by a unique set of nodes. By definition, the validity periods of different nodes cannot overlap. A same time period cannot be represented by two different nodes, which could create inconsistency in the temporal graph.

To keep track of this knowledge element history, the Z^T relation is added to the graph formalism: $K = (N, E, V^T, Z^T)$. It serves to trace the updates of a given knowledge element at any point in time. This relation can also be seen as a temporal identity function which takes as parameters a given node $n \in N$ and a specific time point $t \in T$, and returns the corresponding node at that point. Formally, $Z^T : N \times T \to N$.

In order to consider this new relation in the example presented in Figure 6.5, the definition of K_1 is modified to $K_1 = (\{n_1, n_2, n_3, n_4, n_5\}, \{e_1, e_2, e_3\}, V_1^T, Z_1^T)$ In Figure 6.5, let's imagine that n_1, n_4, and n_5 represent the same knowledge element k_e. The lifecycle of k_e is thus:

- n_1 for period $[t_1, t_2)$,
- n_4 for period $[t_2, t_3)$,
- n_5 for period $[t_3, t_\infty)$.

Let t_1' be a timepoint between t_1 and t_2. When one wants to resolve the node representing the knowledge element at t_1', she or he gets n_1 node, no matter of the node input (n_1, n_4, or n_5): $Z_1^T(n_4, t_1) = n_1$. On the other hand, applying the same relation with another node (n_2 or n_3) returns another node. For example, if n_2 and n_3 do not belong to the same knowledge element, then it will return the node given as input, for example $Z_1^T(n_2, t_1) = n_2$.

Knowledge elements stored in nodes Nodes are used to store the different knowledge elements: context, requirements and actions. The set of nodes N is thus split in three subsets: $N = C \cup R \cup A$ where C is the set of nodes which store context information, R a set of nodes for requirement information and A a set of nodes for action information.

Actions define processes that indirectly impact the context: they will change the behaviour of the system, which will be reflected in the context information. Requirements are also processes that are continuously run over the system in order to check the specifications. Here, the purpose of the A and R subset is not to store these processes but to list them. It can be thought as a catalogue of actions and requirements, with their history.

Using a high-level overview, these processes can be depicted as: taking the knowledge as input, perform tasks, and modify this knowledge as output. As detailed in the next two paragraphs, action executions and requirement analysis can be formalized by relations.

Temporal queries for requirements At the current state, the formalism of the knowledge K does not contain any information regarding the requirement analysis. To overcome this, system requirements analysis R_A are added such as $K = (N, E, V^T, Z^T, R_A)$. R_A is a set of couples composed of patterns $P_{[t_j,t_k]}(K)$ and requirements R over these patterns: $R_P = P \cup R$.

$P_{[t_j,t_k]}$ denotes a temporal graph pattern, where t_j and t_k are the lower and upper bound of the time interval respectively. $P_{[t_j,t_k]}$ is the result of a function which takes the knowledge and an interval as input: $P_{[t_j,t_k]} : K \times I$. The time interval can be either fixed (absolute), *i.e.,* both bounds are precisely defined, or sliding (relative), *i.e.,* the upper bound is computed from the lower bound. For example, $P_{[t_0,t_4]}$ is considered as fixed and $P_{[t_0,t_0+4]}$ is considered as relative. Each element of the pattern should be valid for at least one timepoint: $\forall\, p \in P_{[t_j,t_k)}, V^T(e) \cap [t_j, t_k) \neq \emptyset$. Patterns can be seen as temporal subgraphs of K, with a time limiting constraint coming in the form of a time interval.

Temporal relations for actions Like for R_A, the knowledge K needs to be augmented with action executions A_E: $K = (N, E, V^T, Z^T, R_A, A_E)$. Actions executions A_E can be regarded as a couple (A, A_F), where A is the action that is executed and

A_F a set of relations or isomorphisms mapping a source temporal graph pattern $P_{[t_j,t_k]}$ to a target one $P_{[t_l,t_m]}$, $A_F : K \times I \rightarrow K \times I$.

The left-hand side of the A_F relation depicts the temporal graph elements over which an action is applied. Every relation may have a set of application conditions. They describe the circumstances under which an action should take place. These application conditions are either positive, should hold, or negative, should not hold. Application conditions come in the form of temporal graph invariants. The side effects of these actions are represented by the right-hand side.

Finally, we associate to A_E a temporal function E_{A_E} to determine the time interval at which an action has been executed. Formally, $E_{A_E} : A_E \rightarrow I$.

Temporal relations for decisions Finally, the knowledge formalism needs to include the last, but not the least, element: decisions made by the adaptation, $K = (N, E, V^T, Z^T, R_A, A_E, D)$ While the source of relations in D represents the state before the execution of an action, the target shows its impact on the context*. Its intent is **to trace back impacts of action executions to the decisions they originated from**.

A decision present in D is defined as a set of actions executed, *i.e.,* a subset of A_E, combined with a set of requirement analysis, *i.e.,* a subset of R_A. Formally, $D = \{\ A_D \cup R_D \mid A_D \subseteq A_E, R_A \subseteq R_P\}$. We assume that each action should result from only one decision: $\forall a \in A, \forall d1, d2 \in D \mid a \in d1 \wedge a \in d2 \rightarrow d1 = d2$.

The temporal function E_{A_E} is extended to decisions in order to represent the execution time: $E_{A_E} : (A \cup D) \rightarrow I$. For decision, the lower bound of the interval corresponds to the lowest bound of the action execution intervals. Following the same principle, the upper bound of the interval corresponds to the uppermost bound of the action execution intervals. Formally, $\forall d \in D \rightarrow E_{A_E}(d) = [l, u)$, where $l = \min\limits_{a \in A_d}\{E_{A_E}(a)[start]\}$ and $u = \max\limits_{a \in A_d}\{E_{A_E}(a)[end]\}$.

Sum up Knowledge of an adaptive system can be formalism with a temporal graph such as $K = (N, E, V^T, Z^T, R_A, A_E, D)$, wherein:

- N is a set of nodes to represent the different information (context, actions and requirements)

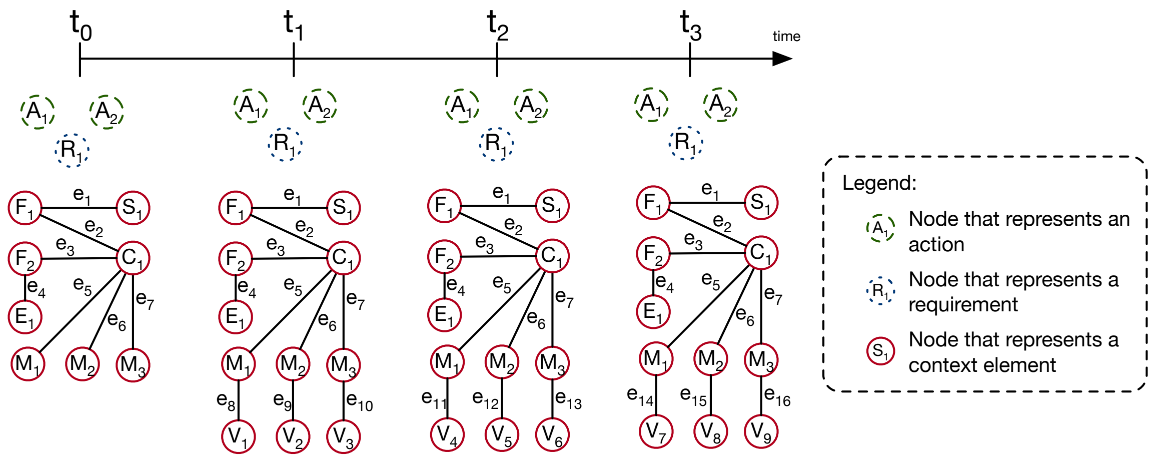

t0
t1
t2
t3
time
A1
A2
R1
F1
S1
F2
C1
E1
M1
M2
M3
V1
V2
V3
V4
V5
V6
V7
V8
V9
e1
e2
e3
e4
e5
e6
e7
e8
e9
e10
e11
e12
e13
e14
e15
e16
Legend:
Node that represents an action
Node that represents a requirement
Node that represents a context element

Figure 6.6: Application of the formalism with a temporal graph that represents the knowledge of the smart grid described in Section 6.1.2

m:application⟩

- E is a set of edges which connects the different nodes,
- V^T is a temporal relation which defines the temporal validity of each element,
- Z^T is a relation to track the history of each knowledge elements,
- R_A is a relation that defines the different requirements processes,
- A_E is a relation that defines the different action processes,
- D is a set of action executions, which result from the same decision, and requirement analysis.

Decisions D can allow adaptation processes to reason over ongoing and future executions of decisions. Moreover, it allows tracing the state of the knowledge before and after the decision has been or is executed, thanks to its A_D component. Plus, it represents which action has been used for this. Thanks to the R_A relation, one can access the requirements at the root of the decision and the state of the knowledge used by this requirement.

In the next section, we exemplify this formalism over our case study.

6.2.3 Application on the use case

In this section we apply the formalism described on the use case presented in Section 6.1.2.

Let K_{SG} be the temporal graph that represents the knowledge of this adaptive system: $K_{SG} = (N_{SG}, E_{SG}, V^T_{SG}, Z^T_{SG}, R_{P_{SG}}, A_{P_{SG}}, D_{SG})$. Figure 6.6 shows the nodes and edges of this knowledge.

Description of $\boldsymbol{N_{SG}}$ N_{SG} is divided into three subsets: C_{SG}, R_{SG} and A_{SG}. R_{SG} contains one node, R_1 in Figure 6.6, which represents the requirement of this example (minimizing the number of overloads): $R_{SG} = \{R_1\}$. Two nodes, A_1 and A_2, belong to A_{SG}: $A_{SG} = \{A_1, A_2\}$. They represent the two actions of this example, respectively decreasing and increasing amps limits. Regarding the context C_{SG}, there are three nodes to represent the three smart meters (M_1, M_2, and M_3), one for the substation (S_1), two for the fuses (F_1 and F_2), one for the dead-end cabinet (E_1), one for the cable (C_1) and one node per consumption value received (V_i): $C_{SG} = \{M_1, M_2, M_3, S_1, F_1, F_2, E_1, C_1\} \cup \{V_i | i \in [1..9]\}$.

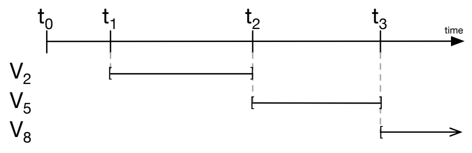
According to the scenario, except for nodes to store consumption values, the other nodes are created at t_0 and are never modified. Therefore, their validity period starts at t_0 and never ends: $\forall n \in A_{SG} \cup R_{SG} \cup \{M_1, M_2, M_3, S_1, F_1, F_2, E_1, C_1\}, V^T_{SG}(n) = [t_0, t_\infty)$. Considering the consumption values, all the nodes represent the history of the values for the three smart meters. In other words, there are three knowledge elements: the consumption measured for each meter. Let C_i notes the consumption measured by the smart meter M_i. As shown in Figure **??**, there is:

- C_1 of M_1 is represented by $\{V_1, V_4, V_7\}$,
- C_2 of M_2 is represented by $\{V_2, V_5, V_8\}$,
- C_3 of M_3 is represented by $\{V_3, V_5, V_9\}$.

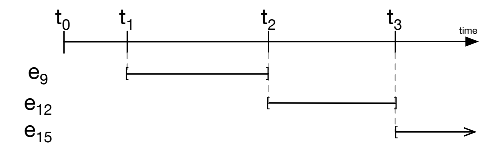
Taking C_2 as an example, V_2 is the initial consumption value, replaced by V_5 at t_2, itself replaced by V_8 at t_3. Applying the V^T_{SG} on these different values, results are thus:

- $V^T_{SG}(V_2) = [t_1, t_2)$,
- $V^T_{SG}(V_5) = [t_2, t_3)$,
- $V^T_{SG}(V_8) = [t_3, t_\infty)$.

These validity periods are shown in Figure 6.7a. As meters send the new consumption values at the same time, this example can also be applied to C_1 and C_3.



(a) Consumption values C_2



(b) Edges linking the meter node M_2 to its consumption values C_2

Figure 6.7: Validity periods of consumption values and their edges to the smart meter M_2

From these validity periods, the Z^T_{SG} can be used to navigate to the different values over time. Let's continue with the same example, C_2. In order to get the evolution of the consumption value C_2, given the initial one, one will use the Z^T_{SG} relation:

- $Z^T_{SG}(V_2, t_{s1}) = \emptyset$, where $t_0 \leqslant t_{s1} < t_1$
- $Z^T_{SG}(V_2, t_{s2}) = V_2$, where $t_1 \leqslant t_{s2} < t_2$
- $Z^T_{SG}(V_2, t_{s3}) = V_5$, where $t_2 \leqslant t_{s3} < t_\infty$.
- $Z^T_{SG}(V_2, t_{s4}) = V_8$, where $t_2 \leqslant t_{s4} < t_\infty$.

Description of $\boldsymbol{E_{SG}}$ In this example, edges are used to store the relationships between the different context elements. For example, the edge between the substation S_1 and the fuse F_1 allow representing the fact that the fuse is physically inside the substation. Another example, edges between the cable C_1 and the meters M_1, M_2 and M_3 represent the fact that these meters are connected to the smart grid through this cable.

One may consider that relations (validity, Z^T, decisions, action executions and requirements analysis) will be stored as edges. But this decision is let to the implementation part of this formalism.

In our model, only consumption values (V_i nodes) are modified over time. Plus, since the scenario does not imply any edge modifications, only those between meters and values are modified. The edge set contains thus sixteen edges: $E_{SG} = \{e_i \mid i \in [1..16]\}$.

By definition, the unmodified edges have a validity period starting from t_0 and never ends: $\forall i \in [1..7], V_{SG}^{T}(e_i) = [t_0, t\infty)$. The history of the three knowledge elements that represent consumption values do not only impact the nodes which represent the values but also the edges between those nodes and the meters ones:

- C_1 impacts edges between M_1 and V_1, V_4, and V_7, *i.e.,* $\{e_8, e_{11}, e_{14}\}$,
- C_2 impacts edges between M_2 and V_2, V_5, and V_8, *i.e.,* $\{e_9, e_{12}, e_{15}\}$,
- C_3 impacts edges between M_3 and V_3, V_6, and V_9, *i.e.,* $\{e_{10}, e_{13}, e_{16}\}$.

Continuing with C_2 as an example, the initial edge value is e_9 from t_1, which is replaced by e_{12} from t_2, itself replaced by e_{15} from t_2. The validity relation, applied to these edges, thus returns:

- $V_{SG}^{T}(e_9) = [t_1, t_2) = V_{SG}^{T}(V_2)$,
- $V_{SG}^{T}(e_{12}) = [t_2, t_3) = V_{SG}^{T}(V_5)$,
- $V_{SG}^{T}(e_{15}) = [t_3, t_\infty) = V_{SG}^{T}(V_8)$,

These validity periods are depicted in Figure 6.7b. As they are driven by those of consumption values (V_2, V_5, and V_8), they are equal.

As for nodes, the Z_{SG}^{T} relation can navigate over time through these values. For example, to get the history of the edges between the consumption value C_2 and the meter represented by M_2, one can apply the Z_{SG}^{T} relation as follows:

- $Z_{SG}^{T}(e_9, t_{s1}) = \emptyset$, where $t_0 \leqslant t_{s1} < t_1$
- $Z_{SG}^{T}(e_9, t_{s2}) = e_9$, where $t_1 \leqslant t_{s2} < t_2$,
- $Z_{SG}^{T}(e_9, t_{s3}) = e_12$, where $t_2 \leqslant t_{s3} < t_3$,
- $Z_{SG}^{T}(e_9, t_{s4}) = e_15$, where $t_3 \leqslant t_{s4} < t_\infty$.

Description of $\boldsymbol{D_{SG}}$, $\boldsymbol{A_{E_{SG}}}$, and $\boldsymbol{R_{A_{SG}}}$ As described in the scenario (cf. Section 6.1.2), the requirement analysis detects that t_1 the requirement is broken. The adaptation process will thus apply the "decreasing amps limits" action on the three meters. Following Example 2 detailed in Section 6.1.1, we consider that the action will impact the consumption values on the next two measurements: t_2 and t_3.

In the knowledge, we thus have one decision: $D_{SG} = D_1$. This decision has been taken after one requirement analysis, $R_{A_{SG1}}$, that detects no respect of the requirement R_1. To determine if there is an overload, this analysis needs to know the topology and the consumption values. The pattern is thus defined by all nodes related to the grid network and consumption values at $t1$: $P_{1[t_1,t_1+1]} = \{S_1, F_1, F_2, C_1, E_1, M_1, M_2, M_3, V_1, V_2, V_3\}$. So we have: $R_{A_{SG1}} = \{R_1, P_{1[t_1,t_1+1]}\}$.

The knowledge also includes the three action executions: $A_{E_{SG1}}$, $A_{E_{SG2}}$, and $A_{E_{SG3}}$. These actions have been executed on, respectively, M_1, M_2, and M_3. Following the definition, they all contain the action A_1 and similar relation which linked the circumstances to the impacts. The circumstances are the state of the knowledge at t_0, which contain all information of the grid network and the consumption values. We denote them $P_{2[t_1,t_1+1]}$, $P_{3[t_1,t_1+1]}$, and $P_{4[t_1,t_1+1]}$, all equal $P_{1[t_1,t_1+1]}$. The impact contains all consumption values received at t_2 and t_3. Each action impacts the consumption value of the meter that it modifies. For example, $A_{E_{SG2}}$ only impacts values of meter M_2. For this action, the output pattern is thus : $P_{5[t_2,t_3]} = \{V_5, V_8\}$. In summary, $A_{E_{SG1}}$, $A_{E_{SG2}}$, and $A_{E_{SG3}}$ are defined as follows:

- for the action executed on M_1: $A_{E_{SG1}} = (A_1, A_{F1})$, with $A_{F1} : P_{2[t_1,t_1+1]} \rightarrow \{V_4, V_7\}$,
- for the action executed on M_2: $A_{E_{SG2}} = (A_1, A_{F2})$, with $A_{F2} : P_{3[t_1,t_1+1]} \rightarrow \{V_5, V_8\}$,
- for the action executed on M_3: $A_{E_{SG3}} = (A_1, A_{F3})$, with $A_{F3} : P_{4[t_1,t_1+1]} \rightarrow \{V_6, V_9\}$,

The decision described in the scenario is thus equal to: $D_1 = \{R_{A_{SG1}}, A_{E_{SG1}}, A_{E_{SG2}}, A_{E_{SG3}}\}$. At t_2, this decision will still be valid. The adaptation process can thus include it in the adaptation process to reason over the ongoing actions. If at t_3 the cable remains overloaded, then one may use this element to check if the system tried to fix it, how and based on which information.

6.3 modelling the knowledge

⟨sec:tkm:mm⟩ In order to simplify the diagnosis of adaptive systems, this thesis proposes a novel metamodel* that combines, what we call, design elements and runtime elements. Design elements abstract the different elements involved in knowledge* information to assist the specification of the adaptation process. Runtime elements instead, represent the data collected by the adaptation process during its execution. In order to maintain the consistency between previous design elements and newly created ones, instances of design elements (*e.g.,* actions) can be either added or removed. Modifying these elements would consist in removing existing elements and creating new ones. Combining design elements and runtime elements in the same model helps not only to acquire the evolution of system but also the evolution of its structure and specification (e.g. evolution of the requirements of the system). Design time elements are depicted in gray in the Figures 6.8– 6.11. Note that, this thesis does not address how runtime information is collected.

For the sake of modularity, the metamodel* has been split into four packages: Knowledge, Context, Requirement and Action. All the classes of these packages have a common parent class that adds the temporality dimention: *TimedElement* class. Before describing the Knowledge (core) package, we detail this element. Then, we introduce in more details the other three packages used by the Knowledge package: Context, Requirement, and Action. In below sections, we use "*Package::Class*" notation to refer to the provenance of a class. If the package is omitted, then the provenance package is this one described by the figure or text.

6.3.1 Parent element: *TimedElement* class

we assume that all the classes in the different packages extend a *TimedElement* class. This class contains three methods: *startTime*, *endTime*, and *modificationsTime*. The first two methods allow accessing the validity interval bounds defined by the previously discussed V^T relation. The last method resolves all the timestamps at which an element has been modified: its history. This method is the implementation of the relation Z^T described in our formalism (cf. Section 6.2.2).

6.3.2 Knowledge metamodel

sec:tkm:mm:knoeldge⟩? In order to enable interactive diagnosis of adaptive systems, traceability links between the decisions made and their circumstances should be organized in a well-structured representation. In what follows, we introduce how the knowledge* metamodel* helps to describe decisions*, which are linked to their goals and their context (input and impact). Figure 6.8 depicts this metamodel*.

Knowledge package is composed of a *context**, a set of *requirements**, a set of

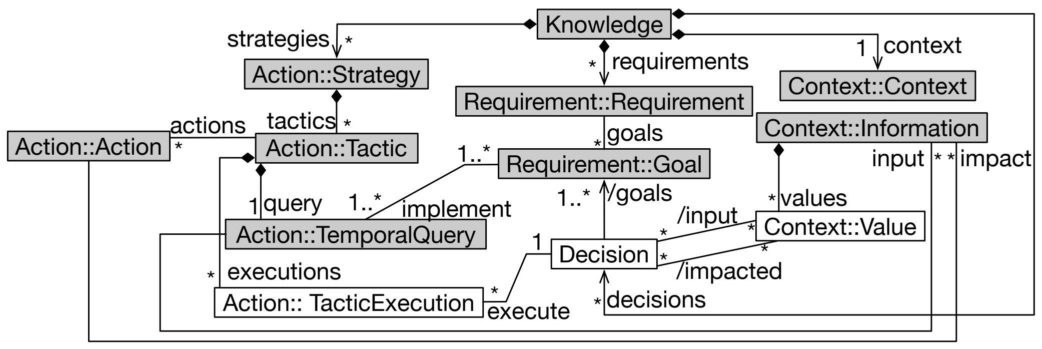


Figure 6.8: Excerpt of the knowledge metamodel

:knowledge-mm⟩

strategies, and a set of *decisions**. A decision* can be seen as the output of the Analyze and Plan steps in the Monitor, Analyse, Plan, and Execute over knowledge (MAPE-k)* loop.

Decisions comprise target *goals* and trigger the execution of one *tactic* or more. A decision has an *input* context and an *impacted* context. The context impacted by a decision (*Decision.impacted*) is a derived relationship computed by aggregating the impacts of all actions belonging to a decision (see Fig. 6.11). Likewise, the *input* relationship is derived and can be computed similarly. In the smart grid example, a decision can be formulated (in plain English) as follows: since the district D is almost overloaded (*input context*), we reduce the amps limit of greedy consumers using the "*reduce amps limit*" *action* in order to reduce the load on the cable of the district (*impact*) and satisfy the "*no overload*" policy (*requirement*).

As all the elements inherit from the *TimedElement*, we can capture the time at which a given decision and its subsequent actions were executed, and when their impact materialized, *i.e.,* measured. Thanks to this metamodel representation, a developer can apprehend the possible causes behind malicious behaviours by navigating from the context values to the decisions that have impacted its value (*Property.expected.impact*) and the goals it was trying to reach (*Decision.goals*). In Section 6.1.2, we present an example of interactive diagnosis queries applied to the smart grid use case.

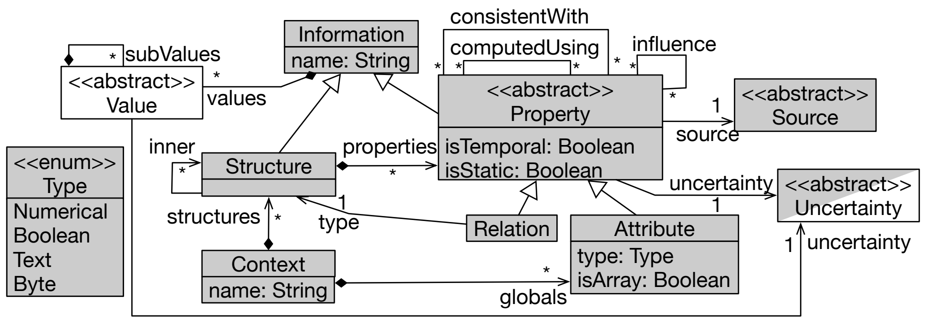


Figure 6.9: Excerpt of the context metamodel

⟨fig:context-model⟩

6.3.3 Context metamodel

Context models structure context information acquired at runtime. For example, in a smart-grid system, the context model would contain information about smart-grid users (address, names, etc.) resource consumption, etc.

An excerpt of the context model is depicted in Figure 6.9. we propose to represent the context as a set of structures (*Context.structures*) and global attributes (*Context.globals*). A structure can be viewed as a C-structure with a set of properties (*Property*): attributes (*Attribute*) or relationships (*Relation*). A structure may contain other nested structures (*Structure.inner*). Structures and properties have values. They correspond to the nodes described in the formalization section (*cf.* Section 6.2.2). The connection feature described in Section **??** is represented thanks to three recursive relationships on the Property class: *consistentWith*, *computedUsing* and *influence*. Additionally, each property has a source (*Source*) and an uncertainty (*Uncertainty*). It is up to the stakeholder to extend data with the appropriate source: measured, computed, provided by a user, or by another system (*e.g.,* weather information coming from a public API). Similarly, the uncertainty class can be extended to represent the different kinds of uncertainties. Finally, a property can be either historic or static.

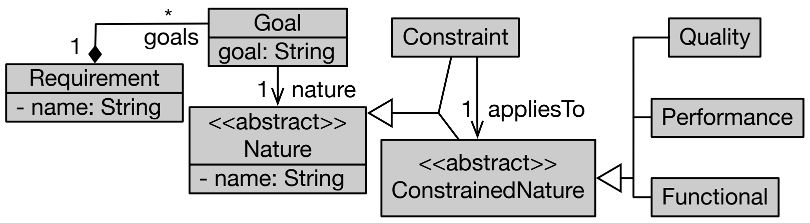


Figure 6.10: Requirement metamodel

irement-model⟩

6.3.4 Requirement metamodel

As different solutions to model system requirements exist (*e.g.,* KAOS [**DBLP:journals/scp/Dardenne** i* [**yu2011modelling**] or Tropos [**DBLP:journals/aamas/BrescianiPGGM04**]), [redacted] in this metamodel, we abstract their shared concepts. The requirement model, depicted in Figure 6.10, represents the *requirement* as a set of *goals*. Each goal has a *nature* and a textual specification. The nature of the goals adheres to the four categories of requirements presented in Section **??**. One may use one of the existing requirements modelling languages (*e.g.,* RELAX) to define the semantics of the requirements. Since the requirement model is composed solely of design elements, we may rely on static analysis techniques to infer the requirement model from existing specifications. The work of Egyed [**DBLP:conf/icse/Egyed01**] is one solution among others. This work is out of the scope of the paper and envisaged for future work.

In the guidance example, the requirement model may contain a **balanced resource distribution** requirement. It can be split into different goals: (i) *minimizing overloads*, (ii) *minimizing production lack*, (iii) *minimizing production loss.*

6.3.5 Action metamodel

ec:action-mm⟩?

Similar to the requirements metamodel, the actions metamodel also abstracts main concepts shared among existing solutions to describe adaptation processes and how they are linked to the context. Figure 6.11 depicts an excerpt of the action metamodel. we define a strategy as a set of tactics (*Strategy*). A tactic contains a set of actions (*Action*). A tactic is executed under a precondition represented as a temporal query

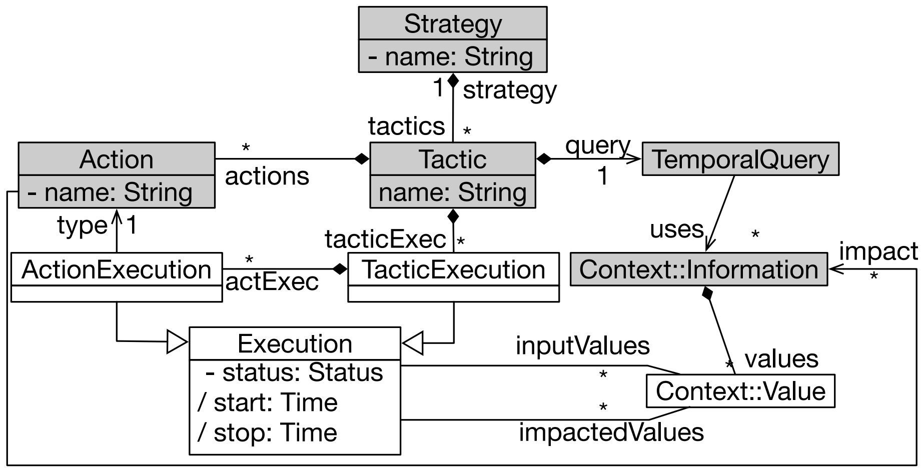


Figure 6.11: Excerpt of the action metamodel

⟨fig:action-mm⟩

(*TemporalQuery*) and uses different data from the context as input. In future work, we will investigate the use of preconditions to schedule the executions order of the actions, similarly to existing formalisms such as Stitch [**DBLP:journals/jss/ChengG12**]. The query can be as complex as needed and can navigate through the whole knowledge model. Actions have impacts on certain properties, represented by the *impacted* reference.

The different executions are represented thanks to the *Execution* class. Each execution has a status to track its progress and links to the impacted context values(*Execution.impactedValues*). Similarly, input values are represented thanks to the *Execution.inputValues* relationship. An execution has *start* and *end* time. Not to confuse with the *startTime* and *endTime* of the validity relation V^T. Whilst the former corresponds to the time range in which a value is valid, the *start* and *stop* time in the class execution correspond to the time range in which an action or a tactic was being executed. The start and stop attributes correspond to the relationL E_{A_E} (see Section 6.2.2). These values can be derived based on the validity relation. They correspond to the time range in which the status of the execution is "*RUNNING*". Formally, for every execution node e, $E_{A_E}(e) = (V(e) \mid e.status = \text{“RUNNING”})$.

Similarly to requirement models, it is possible to automatically infer design elements of action models by statically analyzing actions specification. Since acquiring information about tactics and actions executions happens at runtime, one way to achieve this is by intercepting calls to actions executions and updating the appropriate action model elements accordingly. This is out of the scope of this paper and planned for future work.

6.4 Validation

km:validation⟩ To validate and evaluate our approach, we implemented a prototype publicly available online[5]. This implementation leverages the GreyCat framework[6], more precisely the modelling plugin, which allows designing a meta-model using a textual syntax. Based on this specification, GreyCat generates a Java and a JavaScript API to create and manipulate models that conform to the predefined metamodel. The GreyCat framework handles time as a built-in concept. Additionally, it has native support of a lazy loading mechanism and an advanced garbage collection. This is achieved by dynamically loading and unloading model elements from the main memory when necessary.

The validation of our approach has been driven by the two research questions formulated in the introduction section:

- How to diagnose the self-adaptation process?
- How to enable reasoning over unfinished actions and their expected effects?

To address the first one, we describe how one can use our approach to represent the knowledge of an adaptation process for a smart grid system. Then, we present a code to extract the circumstances and the goals of a decision. For the second one, we present a scenario where a developer can use our approach to reason over unfinished actions and their expected effects. The presented code shows how information can be extracted from our model to enable any reasoning algorithm. Finally, we present a performance evaluation to show the scalability of our approach.

[5]https://github.com/lmouline/LDAS
[6]https://github.com/datathings/greycat

6.4.1 Diagnostic: implementation of the use case

In what follows, we explain how a stakeholder, Morgan, can apply our approach to a smart grid system in order to, first, abstract adaptive system concepts, then, structure runtime data, and finally, query the model for diagnosis purpose. The corresponding object model is depicted in Figure 6.12. Due to space limitation, we only present an excerpt of the knowledge model. An elaborate version is accessible in the tool repository.

Abstracting the adaptive system At design time (t_d), either manually or using an automatic process, Morgan abstracts the different tactics and actions available in the adaptation process. Among the different tactics that Morgan would like to model is "*reduce amps limit*". It is composed of three actions: sending a request to the smart meter (*askReduce*), checking if the new limit corresponds to the desired one (*checkNewLimit*), and notifying the user by e-mail (*notifyUser*). Morgan assumes that the *askReduce* action impacts consumption data (*csmpt*). This tactic is triggered upon a query (*tempQ*) that uses meter (*mt*), consumption (*csmpt*) and customer (*cust*) data. The query implements the "*no overload*" goal: the system shall never have a cable overload. Figure 6.12 depicts a flattened version of the temporal model representing these elements. The tag at upper-left corner of every object illustrates the creation timestamp. All the elements created at this stage are tagged with t_d.

Adding runtime information The adaptation process checks if the current system state fulfills the requirements by analyzing the context. To perform this, it executes the different temporal queries, including *tempQ*. For some reasons, the tempQ reveals that the current context does not respect the "*no overload*" goal. To adapt the smart grid system, the adaptation process decides to start the execution of the previously described tactic (*exec1*) at t_s. As a result, a decision element is added to the model along with a relationship to the unsatisfied goal. In addition, this decision entails the planning of a tactic execution, manifested in the creation of the element *exec1*and its subsequent actions (*notifyU*, *checkLmt*, and *askRed*). At t_s, all the actions execution have an IDLE status and an expected start time. All the elements created at this stage are tagged with the t_s timestamp in Figure 6.12.

At t_{s+1}, the planned tactic starts being executed by running the action *askReduce*.

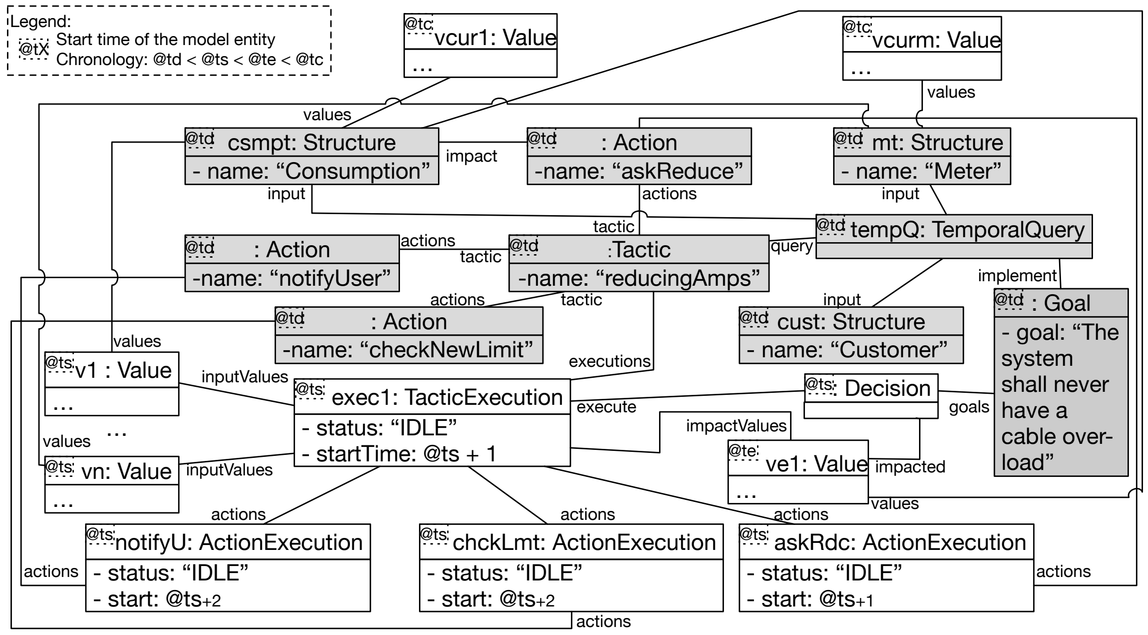


Figure 6.12: Excerpt of the knowledge object model related to our smart grid example

km:valid:diag⟩

The status of this action turns from *IDLE* to *RUNNING*. Later, at t_{s+2}, the execution of *askReduce* finishes with a *SUCCEED* status and triggers the execution of the actions *notifyUser* and *checkNewLimit* in parallel. The status of *askReduce* changes to *SUCCEED* while the status of *notifyUser* and *checkNewLimit* turns to *RUNNING*. The first action successfully ends at t_{s+3} while the second ends at t_{s+4}. As all actions terminates with a *SUCCEED* status at t_{s+4}, accordingly, the final status of the tactic is set *SUCCEED* and the *stop* attribute value is set to t_e.

Interactive diagnosis query After receiving incident reports concerning regular power cuts, and based on the aforementioned knowledge model, Morgan would be able to query the system's states and investigate why such incidents have occurred. As described in Section 6.1.2, she/he will interactively diagnose the system by interrogating the context, the decisions made, and their circumstances.

The first function, depicted in Listing 6.1, allows to navigate from the currently measured values (*vcur1*) to the decision(s) made. The for-loop and the if-condition are responsible for resolving the measured data for the past two days. Past elements are accessed using the *resolve* function that implements the Z^T relation (*cf.* Section 6.2.2). After extracting the decisions leading to power cuts, Morgan carries on with the diagnosis by accessing the circumstances of this decision. The code to perform this task is depicted in Listing 6.1, the second function (getCircumstances). Note that the relationship *Decision.input* is the aggregation of *Decision.excecute.inputValues*.

ode:actions-to-goals⟩

```
// extracting the decisions
Decision[] impactedBy(Value v) {
  Decision[] respD
  for( Time t: v.modificationTimes() ):
    if (t >= v.startTime() - 2 day)
      Value resV = resolve(v,t)
    respD.addAll(from(resV).navigate(Value.impacted))
  return respD
}
// extracting the circumstances of the made decisions
Tuple<Value[], Goal[]> getCircumstance(Decision d) {
  Value[] resValues = from(d).navigate(Decision.input)
  Goal[] resGoals = from(d).navigate(Decision.goals)
  return Tuple<>(resValues, resGoals)
}
```

Listing 6.1: Get the goals used by the adaptation process from executed actions

6.4.2 Reasoning over unfinished actions and their expected effects

By associating the action model to the knowledge model, we aim at enhancing adaptation processes with new abilities to reason. In this section, we present an example of a reasoning algorithm which considers the impacts of running actions. This example is based on our use case (*cf.* Section **??**).

Let's imagine that the adaptation process detects overloaded cables in the smart grid. To fix this situation, it takes several countermeasures, among which there are fuse state modifications. As detailed in Section 6.1.1, this action is considered as delayed action. Later, another incident is detected, for example, a substation is being overloaded. Before taking any actions, the adaptation process can, thanks to our solution, verify if the running actions will be sufficient to solve this new incident. If not, it can either take additional actions or replan the running one. The algorithm to reschedule current actions or to compute additional actions is out of the scope of this thesis. Here, we present the code to extract the required information from our model.

Checking if the running actions will be sufficient to solve all current issues can also be thought as: will the issue remain with the new context, *i.e.,* after each action have been executed. In our case, it is like verifying if the second overload will still remain with the new topology, which is coming. The adaptation process, therefore, needs to extract the context in the future. To do so, the adaptation process should know the latest timepoint at which the impact will be measured. Listing 6.2 shows the code to get this timepoint. Running, idle and finished actions are accessed thanks to the first two nested loops with the if-condition. We consider that failed and canceled actions have no effects. As finished actions may still have effects, we also consider them. Then we navigate through all impacted values to get their start time, *i.e.,* the beginning of their validity period (V^T relation, *cf.* Section 6.2.2). Doing so, we are sure to get the latest timepoint at which an impact will be measurable.

latest-impact⟩
```
Time latestImpact(Knowledge k) {
  Time latestTime = CURRENT_TIME

  for(Decision d: from(k).navigate(decisions))
    for(TacticExecution te: from(d).navigate(execute))
      if(te.status == "RUNNING" || te.status == "IDLE" || te.status == "SUCCEED")
        for(Value v: from(te).navigate(impactedValues))
          if(v.startTime() > latestTime)
            latestTime = v.startTime()

  return latestTime
}
```

Listing 6.2: Get latest timepoint at which the impact will be measured

Using this timepoint, then the adaptation process can then compute how the grid should be after the actions have been executed. If the system has no prediction mechanism, then the adaptation process can verify how the power will be balanced over the new topology. Otherwise, it can use this prediction feature to compute the expected loads with the coming topology. Using this information, it can verify if all current incidents will be solved by the ongoing actions or not. If not, it may take additional actions or reschedule them.

Listing 6.3 depicts the code to extract all running actions. The nested loops allow accessing all executions made by decision. Then, we filter only those with the “RUNNING” status. The resulting collection should then be given to the scheduling algorithm, which will decide if rescheduling is possible and how.

km:valid:extract-act⟩

```
TacticExecution[] runningActions(Knowledge k) {
  TacticExecution[] resA
  for(Decision d: k.decisions) {
    for(TacticExecution te: d.execute) {
      if(te.status == Status.RUNNING) {
        resA.add(te)
      }
    }
  }
  return resA
}
```

Listing 6.3: Extract ongoing actions and their effects

Using our model, developers have two solutions to model a rescheduling operation. Either they modify the actions, which may delete the history of the previous decision, or they mark all running and idle actions as “CANCELED” and create a new decision, with new actions, which update the circumstances and re-use the same requirements.

6.4.3 Performance evaluation

GreyCat stores temporal graph elements in several key/value maps. Thus, the complexity of accessing a graph element is linear and depends on the size of the graph. Note that in our experimentation we evaluate only the execution performance of diagnosis algorithms. For more information on I/O performance in GreyCat, please refer to the original work by Hartmann *et al.,* [**DBLP:conf/seke/0001FJRT17**; **DBLP:phd/basesearch/Hartmann16**].

⟨code:traversal-used⟩

```
MATCH (input)-[*4]->(output)
WHERE input.id IN [randomly generated set]
RETURN output
LIMIT O
```

Listing 6.4: Traversal used during the experimentations

We consider a diagnosis algorithm to be a graph navigation from a set of nodes (input) to another set of nodes (output). Unlike typical graph algorithms, diagnosis algorithms are simple graph traversals and do not involve complex computations at the node level. Hence, we believe that three parameters can impact their performance (memory and/or CPU): the global size of the graph, the size of the input, and the number of traversed elements. In our evaluation, we altered these parameters and report on the behaviour of the main memory and the execution time. The code of our evaluation is publicly available online[7]. All experiments reporting on memory consumption were executed 20 times after one warm-up round. Whilst, execution time experiments were run 100 times after 20 warm-up rounds. The presented results correspond to the mean of all the iterations. We randomly generate graph with sizes (N) ranging from 1 000 to 2 000 000. At every execution iteration, we follow these steps: (1) in a graph with size N, we randomly select a set of I input nodes, (2) then traverse M nodes in the graph, (3) and we collect the first O nodes that are at four hops from the input element. Listing 6.4 describes the behaviour of the traversal using Cypher, a well-known graph traversal language.

We executed our experimentation on a MacBook Pro with an Intel Core i7 processor (2.6 GHz, 4 cores, 16GB main memory (RAM), macOS High Sierra version 10.13.2). We used the Oracle JDK version 1.8.0_65.

How performance is influenced by the graph size *N*? This experimentation aims at showing the impact of the graph size (N) on memory and execution time while performing common diagnosis routines. We fix the size of I to 10. To assure that the behaviour of our traversals is the same, we use a seed value to select the starting input elements. We stop the algorithm when we reach 10 elements. Results are depicted in Figure 6.13.

As we can notice, the graph size does not have a significant impact on the execution time of diagnosis algorithms. For graphs with up to 2,000,000 elements, execution time remains between 2 ms and four 4 ms. We can also notice that the memory consumption insignificantly increases. Thanks to the implementation of a lazy loading

[7]https://bitbucket.org/ludovicpapers/icac18-eval

and a garbage collection strategy by GreyCat, the graph size does not influence memory or execution time performance. The increase in memory consumption can be due to the internal indexes or stores that grow with the graph size.

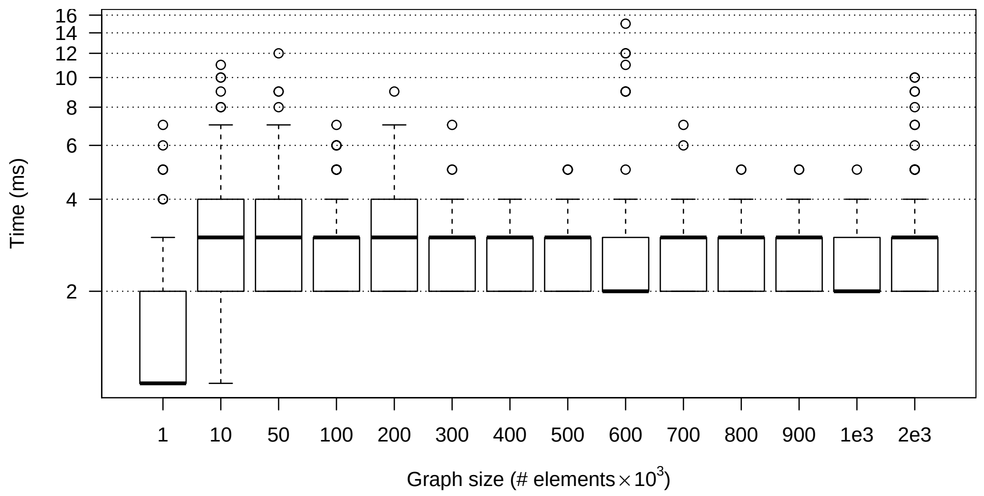
How performance is influenced by the input size (I)? The second experiment aims to show the impact of the input size (I) on the execution of diagnosis algorithms. We fix the size of N to 500 000 and we variate I from 1 000 nodes to 100 000, *i.e.,* from 0.2% to 20% of the graph size. The results are depicted in Figure 6.14 (straight lines).

Unlike to the previous experiment, we notice that the input size (I) impacts the performance, both in terms of memory consumption and execution time. This is because our framework keeps in memory all the traversed elements, namely the input elements. The increase in memory consumption follows a linear trend with regards to N. As it can be noticed, it reaches 2GB for I=100 000. The execution time also shows a similar curve, while the query response time takes around than around 60ms to run for I=1 000, it takes a bit more than 4 seconds to finish for I=100 000. Nonetheless, these results remain very acceptable for diagnosis purposes.

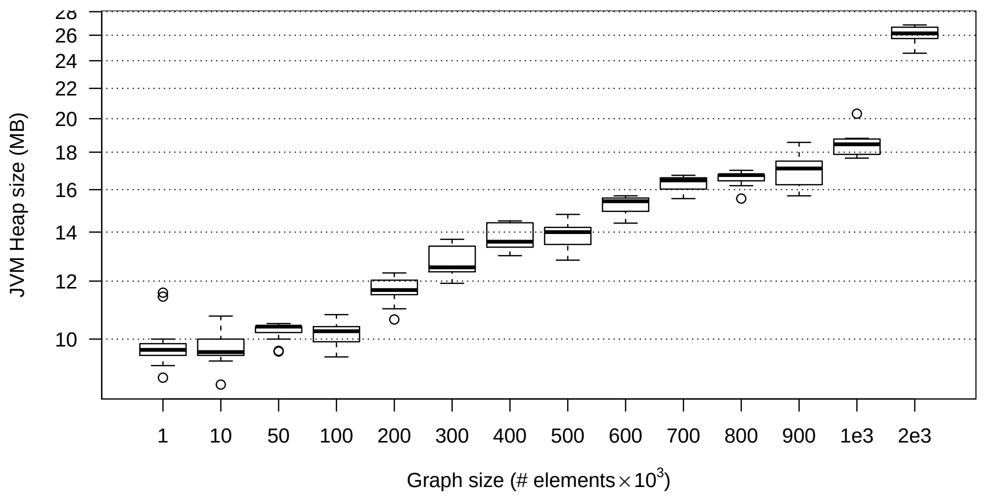
How performance is influenced by the number of traversed elements (M)? For the last experiment, we aim to highlight the impact of the number of traversed elements (M). For this, we fix I and O to 1, and randomly generate a graph with sizes ranging from 1 000 to 100 000. Our algorithm navigates the whole model (M=N). We depict the results in Figure 6.14 (dashed curve). As we can notice, the memory consumption increases in a quasi-linear way. The memory footprint to traverse M = 100 000 elements is around 0.9GB. The progress of the execution time curve behaves similarly, in a quasi-linear way. Finally, the execution time of a full traversal over the biggest graph takes less than 2.5 seconds.

6.4.4 Discussion

By linking context, actions, and requirements using decisions, data extraction for explanation or fault localization can be achieved by performing common temporal graph traversal operations. In the detailed example, we show how a stakeholder



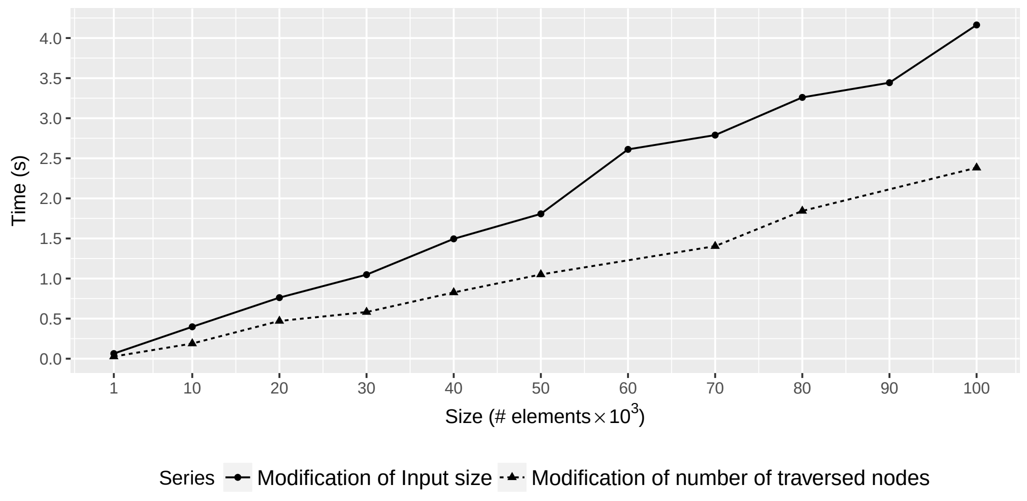
(a) Execution time evolution



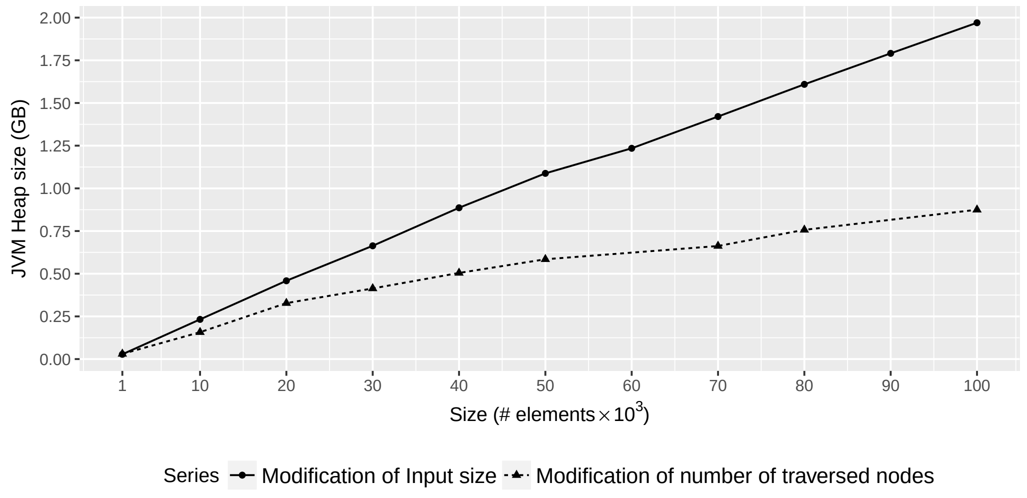
(b) Memory evolution

Figure 6.13: Experimentation results when the knowledge based size increases

⟨fig:exp1⟩



(a) Evolution of the execution time



(b) Evolution of the memory consumption

Figure 6.14: Results of experiments when the number of traversed or input elements increases
⟨fig:exp-res⟩

could use our approach to define the different elements required by such systems, to structure runtime data, finally, to diagnose the behaviour of adaptation processes.

Our implementation allows to dynamically load and release nodes during the execution of a graph traversal. Using this feature, only the needed elements are kept in the main memory. Hence, we can perform interactive diagnosis routines on large graphs with an acceptable memory footprint. However, the performance of our solution, in terms of memory and execution time, is restricted by the number of traversed elements and the number of input elements. Indeed, as shown in our experimentation, both the execution time and the memory consumption grow linearly.

In the Luxembourg smart grid, a district contains approximatively 3 data concentrators and 227 meters[8]. Counting the global datacenter, the network is thus composed of 231 elements. Each meter sends the consumption value every 15 min, being 908 every hours. Plus, there is from 0 to 273 topology modifications in the network. In total, the system generates from 908 to 1,181 new values every hour. If we consider that we have one model element per smart grid entity and one model element per new value, 100,000 model elements correspond thus from $((100,000 - 231) * 1H)/1,181 = 84,5H$ ($\sim$ 3,5 days) to $((100,000 - 231) * 1H)/908 = 109,9H$ ($\sim$ 4,6 days) of data. In other word, our approach can efficiently interrogate up to $\sim$5 days history data in 2.4s of one district.

6.5 Conclusion

km:conclusion⟩ Adaptive systems are prone to faults given their evolving complexity. To enable interactive diagnosis over these systems, we proposed a temporal data model to abstract and store knowledge elements. We also provided a high-level API to specify and perform diagnosis algorithms. Thanks to this structure, a stakeholder can abstract and store decisions made by the adaptation process and link them to their circumstances –targeted requirements and used context– as well as their impacts. In our evaluation, we showed that our solution can efficiently handle up to 100 000

[8]Previously, our studies uses the data described in [**DBLP:conf/smartgridcomm/0001FKTPTR14**] [redacted] which corresponded to the all Luxembourg at this date. Since 2014, the smart grid has been more and more deployed. Numbers present in this paper now correspond more to one district.

elements, in a single machine. This size is comparable to 5 days history of one district in the Luxembourg smart grid.

Throughout this work, we assumed that designers are able to link actions with their expected impacts at design time. However, this is not always true. Some impacts cannot be known in advance. In this perspective, in addition to the future plans already mentioned throughout the paper, we will investigate techniques to identify unknown impacts on the context model, for instance, by studying the use of machine learning techniques. In order to improve the accuracy and correctness of diagnosis routines, another aspect to be considered for future work is handling uncertainty in self-adaptive systems. Understanding the effect of uncertainty on the development of self-adaptive systems and their diagnosis is still an open question. We plan to explore this research direction by answering the following questions: How to represent and express uncertainty in self-adaptive systems at design time? How to efficiently interrogate data with uncertainty in self-adaptive systems, for instance, for troubleshooting purpose?

7

Conclusion

t:conclusion⟩?

This chapter concludes the thesis and presents some perspectives.

List of publications and tools

Papers included in the dissertation

- 2017
  - **DBLP:conf/models/Benelallam0MFBB17**
- 2018
  - **DBLP:conf/icac/MoulineBFBB18**
  - **DBLP:conf/sac/MoulineB0FBMB18**
- in the process of submission
  - **insubmission:2019:comlan:datauncertainty**

Papers not included in the dissertation

- 2017
  - **DBLP:conf/programming/Mouline0FTBB17**
- 2018
  - **DBLP:conf/mobiquitous/GuineaBMT18**

Tools included in the dissertation

- Ain'tea: a language which integrated data uncertainty as a first-class citizen
  - `https://github.com/lmouline/aintea`
- LDAS: a meta-model of knowledge for adaptive systems
  - `https://github.com/lmouline/LDAS`

Abbreviations

API Application Programming Interface. 11, 19

CPS Cyber-Physical System. 2
CPU Central Processing Unit. 4

DOM Document Object Model. 20
DSL Domain Specific Language. 20, *Glossary:* DSL
DSML Domain Specific Modelling Language. 15, 16, 20

E-MOF Essential MOF (EMOF). 18
EMF Eclipse Modelling Framework. 18, 19

GCM GreyCat Modelling Environment. 19
GPL General Purpose Language. 20
GPS Global Positioning System. 4

ICT Information and communication technology. 2
IoT Internet of Things. 18

KMF Kevoree Modelling Framework. 19

MAPE-k Monitor, Analyse, Plan, and Execute over knowledge. 57, *Glossary:* MAPE-k
MDE Model-Driven Engineering. 2, 3, 7, 9, 12, 14–18, 20
MOF Meta Object Facility. 17, 18

OCL Object Constraint Language. 17

OMG Object Management Group. 17

SLE Software Language Engineering. 19, 21, *Glossary:* SLE

UML Unified Modelling Language. 14, 16

XMI XML Metadata Interchange. 17

Glossary

action In this document, we use the definition provided by IEEE Standards [**iso2017systems**]
"Process that, given the context* and requirements* as input, adjusts the system behaviour*". 45

adaptive system In this document, we modified the definition of self-adaptive systems provided by Cheng *et al.,* in [**DBLP:conf/dagstuhl/ChengLGIMABBBCSDFGGGKKKLMM**
Adaptive systems are able to have their behaviour* adjusted in response to the perception of the environment* and the system themselves. If a system perform this adjustment on itself, the literature refers to it as self-adaptive system. 3–5, 7, 11, 12, 19

behaviour See system behaviour*. 2, 3, 11

circumstance State of the knowledge* when a decision* has been taken. 45

context In this document, we use the definition provided by Anind K. Dey [**DBLP:journals/puc/Dey01**]
"Context is any information that can be used to characterise the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and [the system], including the user and [the system] themselves". 45, 50, 56

decision A set of actions* taken after comparing the state of the knowledge* with the requirement*. 45, 56, 57

DSL In this document, we use the definition provided by Deursen *et al.,* [**DBLP:journals/sigplan/Deurse**
"is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually

restricted to, a particular problem domain.”. 20, *Abbreviation:* DSL

environment See system environment*. 2, 3

knowledge The knowledge of an adaptive system gathers information about the context*, actions* and requirements*. 3, 11, 45, 46, 55, 56

long-term action An action* that is not immediate, or that takes time to be executed, or that has long-term effects.. 4, 10

MAPE-k A theoretical model of the adaptation process proposed by Kephart and Chess [**DBLP:journals/computer/KephartC03**]. It divides the process in four stages: monitoring, analysing, planning and executing. These four stages share a knowledge*. 57, *Abbreviation:* MAPE-k
metamodel In this document, we use the definition provided by Douglas C. Schmidt [**DBLP:journals/computer/Schmidt06**]: “[Metamodels] define the relationships among concepts in a domain and precisely specify the key semantics and constraints associated with these domain concepts”. 16–19, 21, 55, 56
model In this document, we use the definition provided by Brambilla *et al.,* [**DBLP:series/synthes**
“[A model is] a simplified or partial representation of reality, defined in order to accomplish a task or to reach an agreement on a topic.” The model should conform to a metamodel*: each element of the model instantiates one from the metamodel* and satisfies all semantics rules [**DBLP:conf/iceccs/BezivinJT05**]. 3, 15–18, 21
models@run.time In this document, we use the definition provided by Blair *et al.,* [**DBLP:journals/computer/BlairBF09**]: “A model@run.time is a causally connected self-representation of the associated system that emphasises the structure, behaviour, or goals of the system from a problem space perspective”. 3, 17–19

requirement In this document, we use the definition provided by IEEE Standards [**iso2017systems**]: “(1) Statement that translates or expresses a need and its associated constraints and conditions, (2) Condition or capability that must be met or possessed by a system [...] to satisfy an agreement, standard, specification, or other formally imposed documents”. 45, 56

self-adaptive system See adaptive system*. 2, 3

self-healing Refers to the capacity of detecting, diagnosing, and repairing any error in the system. See self-healing system*. 2

self-healing system In this document, we use the definition provided by Kephart and Chess [**DBLP:journals/computer/KephartC03**]: "[A self-healing] system automatically detects, diagnoses, and repairs localised software and hardware problems.". 2

SLE In this document, we use the definition provided by Anneke Kleppe [**kleppe2008software**] "the application of systematic, disciplined, and measurable approaches to the development, use, deployment, and maintenance of software languages". 20, *Abbreviation:* SLE

smart grid In this document, we use the definition provided by the National Institute of Standards and Technology (NIST)* [**NIST:SmartGrid:Def:What**]: "a modernized grid that enables bidirectional flows of energy and uses two-way communication and control capabilities that will lead to an array of new functionalities and applications.". 2, 5, 11, 12

software language In this document, we use the definition provided by Anneke Kleppe [**kleppe2008software**]: "any language that is created to describe and create software systems". 20

structure See system structure*. 2