UNIVERSITÉ DU LUXEMBOURG

UNIVERSITÉ DE RENNES 1

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

Dissertation defence committee *(Waiting for approval)*

Dr. Jacques Klein, chairman
Senior Research Scientist, University of Luxembourg, Luxembourg, Luxembourg

Prof. Dr. Manuel Wimmer, vice-chairman & reviewer
Professor, JKU Linz, Linz, Austria

Prof. Dr. Yves Le Traon, co-supervisor
Professor, University of Luxembourg, Luxembourg, Luxembourg

Prof. Dr. Olivier Barais, co-supervisor
Professor, University of Rennes 1, Rennes, France

Dr. Johann Bourcier, advisor
Lecturer, University of Rennes 1, Rennes, France

Dr. Franck Fleurey, member & reviewer
Research Associate, SINTEF, Oslo, Norway

A-Prof. Dr. Ada Diaconescu, expert
Research Associate, Telecom ParisTech, Paris, France

Dr. François Fouquet, expert & advisor
CTO, DataThings, Luxembourg, Luxembourg

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 a description of systems, their behaviours, and/or their environments alleviating their complexity. **Adaptation** is a cornerstone feature that enables reconfiguration at runtime for optimising software to the current and/or future situation.

The model-driven engineering (MDE) methodology promotes the use of abstraction in software engineering. 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 **long-term 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 long-term actions are not specific to smart grids.**

First, data are, almost by definition, uncertain and developers work with estimates in most cases. 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, we argue that this data uncertainty and this de-

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

lay 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 long-term 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 contributions: 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. First, 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.

Second, 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

Part I

Context and challenges in modelling adaptive systems

1

Introduction

⟨chapt:intro⟩

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

Self-adaptive systems optimize their behaviours or configurations at runtime in response to a modification of their environments or their behaviours [CdLG+09]. Kephart and Chess [KC03] laid the groundwork for this approach, based on an IBM white paper [Com+06]. Since then, practitioners have applied it to different domains [GVD19] such as cloud infrastructure [JG17; Tea15; BKF+17] or Cyber-Physical System (CPS) [LGC17; FMF+12; HFK+14a]. One example of such a system is a smart grid, which employs the adaptation capacity to heal itself autonomously.

A smart grid is a power grid in which utilities introduces Information and Communication Technology (ICT) to face the new challenges of electricity supply [Far10; IA09; FMX+12]. One of the required feature is the self-healing capacity. A self-healing system can automatically repair any incident, software or hardware, at runtime [KC03]. For example, a smart grid can optimise the power flow to deal with failures of transformers[1] [FMX+12].

The adaptation process can be performed only if the system has a deep understanding of the situation and the problem. In this case, the situation comprises the structure, the behaviour and the environment (where it is executed) of the system. This understanding can be extracted from an, or a set of, **abstraction**(s) of these elements. Abstractions provide a description of systems, their behaviours, or their environments. For example, Hartmann *et al.,* [HFK+14b] provide a class diagram that describes the smart grid topology, when it uses power lines communications[2].

Model-Driven Engineering (MDE) defenders argue for using the abstraction mechanism to facilitate the development of current software [Sch06; Ken02; BCW17]. This methodology can be applied to different stages of software development. In this thesis, we focus on one of its paradigms: **models@run.time** [BBF09; MBJ+09]. As we depict in Figure 1.1, using this paradigm, the adaptation process relies on a model for analysing the situation and triggering the adaptation. In this document, we say that the model represents the knowledge of the adaptation process. Developers

[1] Transformers change the voltage in the cables.

[2] Data are sent through cables that also distribute electricity.

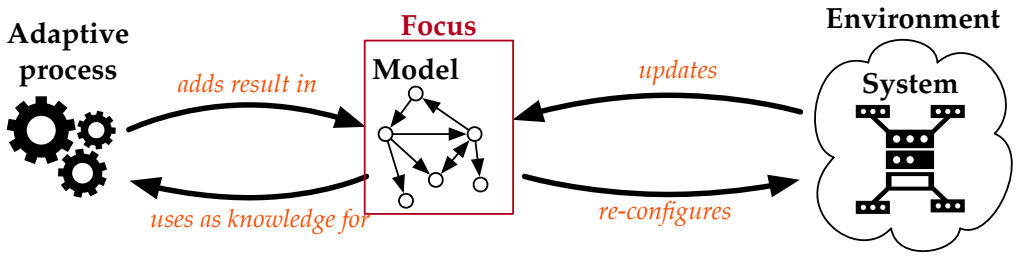


Figure 1.1: Overview of the models@run.time and focus of the thesis

:context:m@rt⟩

can use this paradigm to implement adaptive systems [MBJ+09; HFK+14a]. This dissertation contributes to this modelling layer.

1.2 Challenges

During our study, we have identified five 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 [ZA11], *i.e.,* it cannot be entirely known at design time. Four, the different sub-parts of the system do not evolve at the same rate. Five, structure and behaviour of systems have a time dimension. The last one has been published in our vision paper regarding time awareness in MDE [BHM+17]. We detailed them in this section, .

1.2.1 Engineering uncertainty-aware software

hallenges:duc⟩

Most fuses are manually opened and closed 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 work with estimates in most cases** [BMM14; Met08; AY09]. The uncertainty may

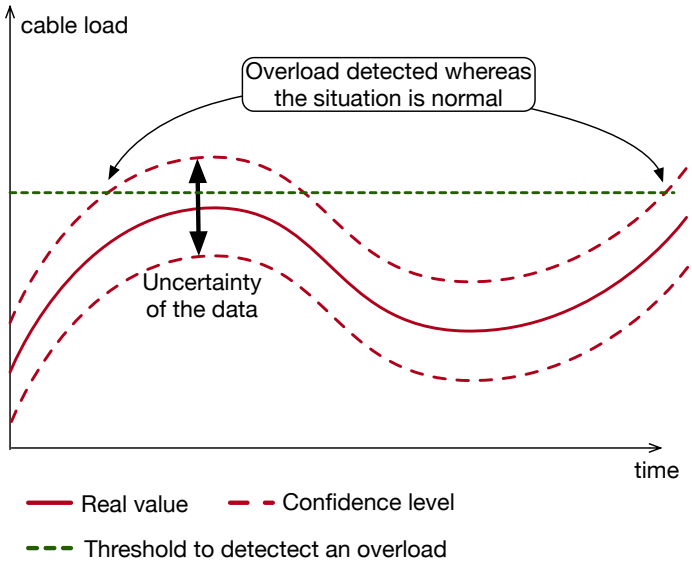


Figure 1.2: Illustration of the problem due to data uncertainty

⟨fig:intro:chal:duc⟩

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 [Met08; BMM14]. Humans are error-prone. 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 as depicted in Figure 1.2. 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 cable load in a smart grid may trigger false detection of an overload, as depicted in Figure 1.2. **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.

Therefore, we argue that data uncertainty impacts all the development stages of software, from the design to the execution. Among the different stages, in this thesis

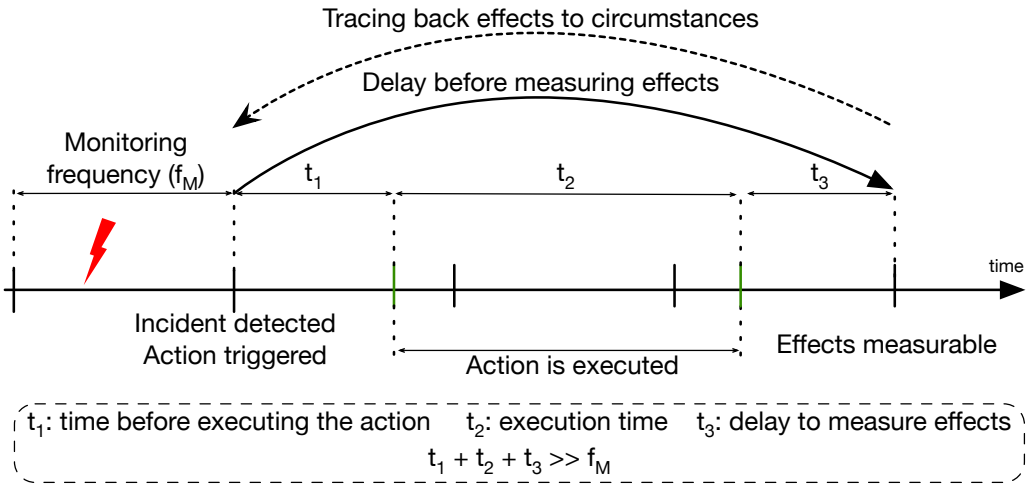


Figure 1.3: Illustration of a long-term action

l:longTermAct⟩

we focus on the design one. We firmly think that design techniques should provide mechanisms to help developers abstract and manipulating uncertain data.

The literature provides approaches to help engineers reason or manipulate data uncertainty, or at least probability distributions. For example, believe functions [Sha76] help to reduce this uncertainty by combining several sources of data. The probabilistic programming [GHN+14] community provide frameworks and languages [MWG+18; BDI+17] to propagate probabilities through computations.

However, from the best of our knowledge, no 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:

How to engineer uncertainty-aware software (design, implement, test, and validate)?

1.2.2 Reasoning over long-term actions

s:longTermAct⟩

Reconfiguring a smart grid implies to change the power flow by opening or closing fuses. As said before, technicians need to drive physically to fuse locations to modify their states. In the case of the Luxembourg smart grid, meters send energy measurement every 15 min, non-synchronously. Therefore, 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 rates, it can cause repeated decisions.

More generally, a difference may exist between the monitoring frequency of the time for action effects to be measured. One cause of this is what we call long-term actions in this document, illustrated in Figure 1.3. A long-term action is defined as an action that takes time to be executed (delay to be executed and execution time) or that have long-term effects. A second cause is an impossibility to reduce the monitoring frequency since systems must be reactive in some cases. This difference in rates may damage the decision process.

Therefore, we argue that **decision-making processes should consider this delay if the frequency of the monitoring stage is lower than the time of action effects to be measurable**. From the best of our knowledge, none of the approaches allows developers implementing such tools. One open challenge for the research community is thus:

How to model, store, and query long-term actions with their effects?

1.2.3 Diagnosing the adaptation process

challenges:diagnosis⟩ Smart grid behaviour is affected by several factors that cannot be controlled by the grid manager. One example is weather conditions. Smart grids rely on an energy production distributed over several actors. For instance, users, who were mainly consumers before, can 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]. Despite this stochasticity of the behaviour, engineers need to implement an adaptation process, that can lead to suboptimal grid configuration.

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 [BdMM+17; MPS15; HBB15]. Even with a meticulous verification or validation process, it is very likely to run into an unexpected behaviour that was not foreseen at

[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.

design time. Alone, existing formal modelling and verification approaches may not be sufficient to anticipate these failures [TOH17]. 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.

Bencomo *et al.,* [BWS+12] 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, as shown in Figure 1.3, 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**. In other words, the research community should answer the following global challenge:

How to trace back adaptation decision effects to their circumstances?

1.2.4 Modelling inconsistent states of systems

Every meter sends consumption and production data every 15 min. However, this collection is not synchronous. That is, not all meters 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 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. Reconfiguration of the grid may thus be suboptimal due to outdated information.

Different parts of a system may evolve at different rates. Some systems are heterogeneous in terms of hardware and software. This diversity results in different evolution or reaction rates. 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 rates, 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. For example, the adaptation process may require a global view of the system. In the worst case, some data would be outdated and cannot be used.

When designing the adaptation process, engineers need thus solutions to deal with an inconsistent system state. One solution can, for example, seamlessly estimate what should be the current value of outdated data. One global challenge for the software engineering community is therefore:

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

1.2.5 Modelling temporal and interconnected data

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 and interconnected data on a large scale.**

Time in software engineering is not a new challenge. For example, Riviera *et al.,* [RRV08] have already identified time as a challenge for the MDE community. Different approaches have been defined [BCC+15; KT12; KH10; HFN+14b].

However, we notice that research efforts made by the MDE community did not focus on the modelling, persistence, and processing of evolving data. Thomas Hartmann started addressing these challenging in his PhD thesis [Har16]. The final global challenge, not fully addressed, is thus:

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

1.3 Scope of the thesis

Among all the challenges described in the previous section, this thesis focus on three of them: data uncertainty (Section 1.2.1), long-term actions (Section 1.2.2), and error-prone adaptation process (Section 1.2.3). 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 [Zad; Met08; Sha76]. The application of these techniques requires a deep understanding of the underlying theories and is a time-consuming task [VMO16]. 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 for software engineer?

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 or 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 [BdMM+17; MPS15; HBB15]. Moreover, complex systems

```
Gaussian<double> average(Gaussian<double>[] arrays) {
    Gaussian<int> sum = arrays[0];
    for(int i
        sum =
    }
    return sum / arrays.length;
}
```

Gaussian distribution can only be applied on (float, double). Actual: int

Figure 1.4: Overview of the language proposed, Ain'tea

intro:contrib:aintea⟩

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 [TOH17]. 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 model the decisions of an adaptation process to diagnose it?

1.4 Contribution & validation

⟨sec:intro:contrib⟩ 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 metamodel, 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. An overview of the language is depicted in Figure 1.4. 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,

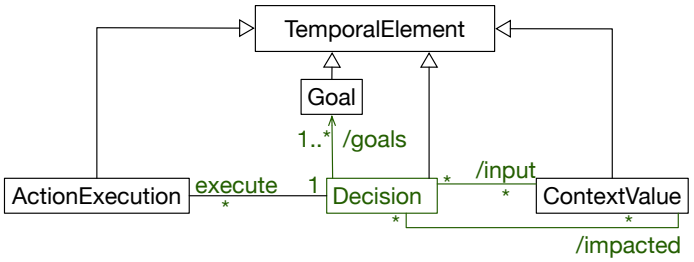


Figure 1.5: Overview of the temporal knowledge model

o:contrib:tkm⟩

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]:

- "Ain'tea: Managing Data Uncertainty at the Language Level", Mouline, Benelallam, Hartmann, Bourcier, Barais, and Cordy

A temporal knowledge metamodel This contribution addresses the challenge of reasoning over unfinished actions, and 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 decision impacts 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 give an overview of the metamodel in Figure 1.5. 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 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

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

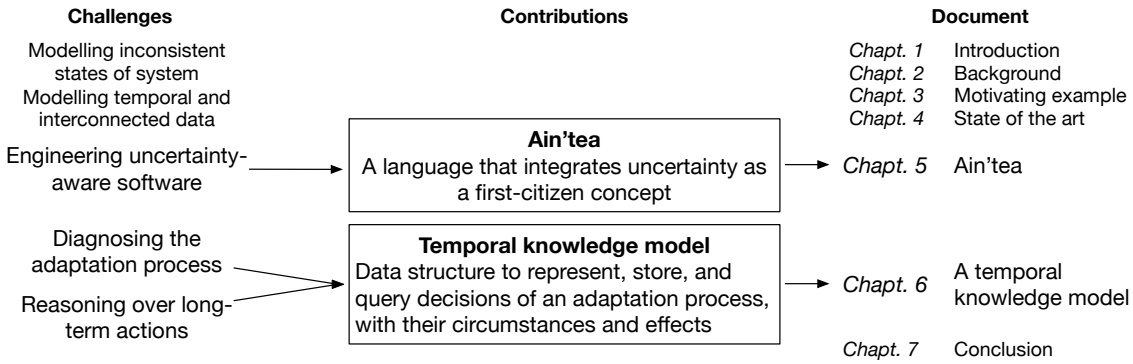


Figure 1.6: Structure of the document

⟨fig:intro:structDoc⟩

on Autonomic Computing[6] (ICAC) and at the ACM/SIGAPP Symposium On Applied Computing[7] (SAC):

- "Enabling temporal-aware contexts for adaptative distributed systems", Mouline, Benelallam, Hartmann, Fouquet, Bourcier, Morin, and Barais
- "A Temporal Model for Interactive Diagnosis of Adaptive Systems", Mouline, Benelallam, Fouquet, Bourcier, and Barais

1.5 Structure of the document

We split the remaining part of this document into six chapters, as shown in Figure 1.6. First, Chapter 2 describes the necessary background of the thesis. Then, Chapter 3 describes a motivating example, based on a smart grid system. 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 4. Chapter 5 and Chapter 6 described our two contributions. 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 actions with their circumstances and effects. Finally, we conclude in Chapter 7, and we present a set of future works.

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

2

Background

pt:background⟩

Contents

In this chapter, we describe the principles of the different software engineering approaches related to this thesis: adaptive systems, Model-Driven Engineering (MDE), Software Language Engineering (SLE), and probability theory. For each of them, we also detail the concepts used in our work, and we explicitly link them to our vision and our contributions.

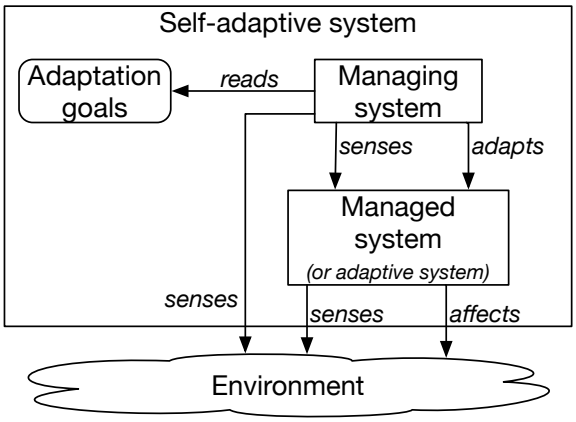


Figure 2.1: Conceptual vision of adaptive system (based on [Wey19])

:adptSyst:principles⟩

2.1 Adaptive systems

This section introduces the background to understand adaptive systems. First, we describe the principles and vision of adaptive systems. Before we characterise the information used by an adaptation process, we detail a model-based technique to implement them: models@run.time. Finally, we highlight the key concepts used in this thesis and link them to the contributions.

2.1.1 Principles and vision

Adaptive systems take their origins in the vision paper of Kephart and Chess [KC03], which is based on the autonomic computing vision pushed by IBM engineers [Com+06]. These systems are recognised by their ability to have their behaviour or structure adapted automatically in response to changes in their environment or of the systems themselves. This adaptation helps them to achieve their goals based on high-level objectives [CdLG+09]. If a system performs itself this adaptation mechanism with minimal interference, the literature refers to it as a self-adaptive system [BSG+09].

Danny Weyns identified two principles for adaptive systems [Wey19]: the internal and the external principles. The former one is based on the "discipline split" defined by Andersson *et al.,* [AdLM+09]: each adaptive system can be split into two concerns. First, the domain concern categorises the part of the system that deals with

the purpose for which the system has been built. Second, the adaptation concern handles the adaptation mechanism and interacts with the first one without interfering with it [KM90]. The external principle says that adaptive systems should handle changes and uncertainties in their environment, the managed systems, and the goal autonomously.

In addition to these principles, the literature has defined four adaptation goals, usually called the self-* features [Com+06]: self-healing, self-optimising, self-configuring, and self-protecting. First, the healing capacity, defined when the failures in the system can be automatically discovered, diagnosed, and repaired. Second, the adaptation mechanism can be used to optimise the system by tuning resources and balancing workloads. Third, the system can be autonomously configured for dynamically adapting to the changes in the environment. Four, threats can be anticipated, detected, and identified by the adaptation process to protect the managed system. Besides, we can add the self-organisation feature [Dem98]: adaptive systems can "acquire and maintain themselves, without external control." [WH04]. It is mainly discussed for distributed systems, where local rules are applied to adjust their interactions and act co-operatively for adaptation. However, this mechanism can lead to emergent behaviour [WH04].

As depicted in Figure 2.1, each adaptive system is composed of four elements: the environment, the managed system, adaptation goals, and the managing system [Wey19]. The environment includes all external entities, virtual or physical, with which the adaptive system interacts on each it effects [Jac97]. Only the elements that are monitored are part of the system. One may distinguish the environment to the adaptive system as, contrary to the element of the adaptive system, it cannot be directly impacted by the engineer. The managed system evolves in the environment and covers all the part of the system that implements the domain concern. In the literature, researchers use different names to refer to it: managed element [KC03], system layer [GCH+04], core function [ST09], base-level subsystem [WMA12], or controllable plant [FHM14]. To enable the adaptation process, the managed system should contain sensors, for monitoring, and actuators, for modifications. This adaptation process needs adaptation goals to perform. They are related to the

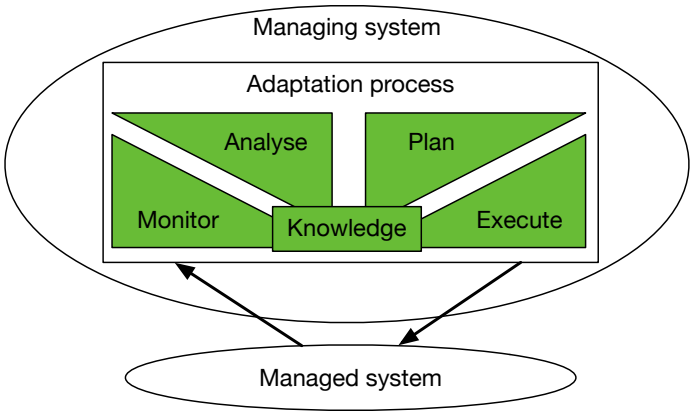


Figure 2.2: MAPE-k loop (based on [KC03])

yst:principles:mapek⟩

managed system and mainly concern its software quality metrics [WA13]. At the roots of the self-* features, Kephart and Chess have defined four families of goals: configuration, optimisation, healing, and protection [KC03]. Engineers can redefine these goals over time, which should consider the uncertainty of the environment or the system. To express such goals, different approaches have been defined, such as probabilistic temporal logic [CGK+] or fuzzy goals [BPS10]. Finally, the managing system will use these goals to drive the adaptation of the managed system in response to changes in the environment. It thus continuously monitor the environment and the managing system. Researchers use different names to refer to this element: autonomic manager [KC03], architecture layer [GCH+04], adaptation engine [ST09], reflective subsystem [WMA12], controller [FHM14].

In the literature, we can find different approaches to engineer adaptive systems [GCH+04]. Among them, the most used one took its inspiration from control theory [BSG+09]: the feedback control loop. The common implementation is the MAPE-k loop [KC03; Com+06], shown in Figure 2.2. This loop is split into four phases: monitoring, analyse, planning, and executing. During the monitoring phase, all information of the managed system and the environment are put into the knowledge. Based on the updated knowledge, the analyse phase detects any need for adaptation using the adaptation goals. If any, the planning phase computes the set of actions to adjust the managing system structure or behaviour. Finally, the executing phase

completes the plan. From the MDE community, one approach to implement such a feedback look is to use the models@run.time paradigm, explained in the next section.

2.1.2 Models@run.time

ound:sas:m@rt⟩ The adaptation process needs to have a deep understanding of the system and its environment. Following the MDE methodology[1], research efforts have led to the models@run.time paradigm [MBJ+09; BBF09]. It stresses the use of a model layer, causally connected to the system, and used by the adaptation process. The causal connection encompasses two features of the model. First, the model reflects the up-to-date state of the system (structure and behaviour) and its environment. Second, any modification of the model triggers a modification of the system. In this way, the model can be used as an interface between the system and the adaptation process, as shown in Figure 1.1. The layer can contain several models to represent different aspect of the system at runtime [MBJ+09]: structure, behaviour, environment, *etc.*Moreover, it can contains representation that guide the adaptation process [CGF+09; HHP+08] such as configurations point.

Following the MAPE-k, the model layer should structure the knowledge. In this thesis, we define a metamodel of this knowledge to enable developers diagnosing, understanding, and reasoning over long-term action (*cf.* Chapter 6). In the next section, we thus characterise the information that composes the knowledge.

2.1.3 Characterisation of information of the knowledge

General concepts of adaptation processes

Similar to the definition provided by Kephart [KC03], IBM defines adaptive systems as "a computing environment with the ability to manage itself and **dynamically adapt** to change in accordance with **business policies and objectives**. [These systems] can perform such activities based on **situations they observe or sense in the IT environment** [...]" [Com+06].

Based on this definition, we can identify three principal concepts involved in

[1]MDE is a methodology that promotes the usage of models for software engineering activities (*cf.* Section 2.2)

Element	Description
Context	Context data are characterised by: their volatility, their temporality, their uncertainty, their source, or their connection.
Requirement	Three kinds of requirements: performance, specific quality, and constraint.
Action	Four pieces of information characterise an action: requirement(s) to achieve, initial context, effects, and execution information (start time, end time, status).

Table 2.1: Characterisation of information of the knowledge

round:adptSyst:charc⟩

adaptation processes. The first concept is **actions**. They are executed to perform a dynamic adaptation through actuators. The second concept is **business policies and objectives**, which is also referred to as the **system requirements** in the domain of (self-)adaptive systems. The last concept is the observed or sensed **situation**, also known as the **context**. The following subsections provide more details about these concepts. Table 2.1 gives an overview of these different elements.

Context

In this thesis, we use the widely accepted definition of context provided by Dey [Dey01]: "Context is **any information that can be used to characterize** 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". In this section, we list the characteristics of this information based on several works found in the literature [HIR02; HFN+14b; BBH+10; PZC+14]. We use them to drive our design choices of our *Knowledge* metamodel (*cf.* Section 6.3.2).

Volatility Data can be either **static** or **dynamic**. Static data, also called frozen, are data that will not be modified, over time, after their creation [HIR02; MSS13; BBH+10]. For example, the location of a machine, the first name or birth date of a user can be identified as static data. Dynamic data, also referred to as volatile data, are data that will be modified over time.

Temporality In dynamic data, we may be interested not only in storing the latest value, but also the previous ones [HFN+14b; HIR02]. We refer to these data as

historical data. Temporal data is not only about past values, but also future ones. Two kinds of future values can be identified, **predicted** and **planned**. Thanks to machine learning or statistical methods, dynamic data values can be **predicted**. **Planned** data are set by a system or a human to specify planned modification on the data.

Uncertainty One of the recurrent problems facing context-aware applications is the data uncertainty [dLGM+10; HIR02; MSS13; BBH+10]. Uncertain data are not likely to represent reality. They contain noise that makes it deviate from its real value. This noise is mainly due to the inaccuracy and imprecision of sensors. Another source of uncertainty is the behaviour of the environment, which can be unpredictable. All the computations that use uncertain data are also uncertain by propagation.

Source In this dissertation, we distinguish three main categories of data source: sensed, computed, profiled, and given. Sensed data are thous measured by a sensor (hardware of software). Computed data result from a process that combine different data. And profiled data are those that are learned. Finally, given data are static data manually set in a software.

Connection Context data entities are usually linked using three kinds of connections: conceptual, computational, and consistency [HIR02; BBH+10]. The conceptual connection relates to (direct) relationships between entities in the real world (e.g. smart meter and concentrator). The computational connection is set up when the state of an entity can be linked to another one by a computation process (derived, predicted). Finally, the consistency connection relates to entities that should have consistent values. For instance, temperature sensors belonging to the same geographical area.

Requirement

knowledge:req⟩ Adaptation processes aim at modifying the system state to reach an optimal one. All along this process, the system should respect the **system requirements** established ahead. Through this thesis, we use the definition provided by IEEE [III17]: "(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".

Although in the literature, requirements are categorised as functional or non-functional, in this document we use a more elaborate taxonomy introduced by Glinz [Gli07]. It classifies requirements in four categories: functional, performance, specific quality, and constraint. All these categories share a common feature: they are all temporal. During the life-cycle of an adaptive system, a stakeholder can update, add or remove some requirements [CA07; PSR10].

Action

In the IEEE Standards [III17], an action is defined as: "**process of transformation** that **operates upon data** or other types of inputs to create data, produce outputs, or **change the state** or condition of the subject software".

In the context of adaptive systems, we can define an action as a process that, given the context and requirements as input, adjusts the system behaviour. This modification will then create new data that correspond to an output context. In the remainder of this document, we refer to the output context as impacted context, or effect(s). Whereas requirements are used to add preconditions to the actions, context information is used to drive the modifications. Actions executions have a start time and a finish time. They can either succeed, fail, or be cancelled by an internal or external actor.

2.1.4 Key concepts for this thesis

Adaptive systems have been proposed to tackle the growing complexity of systems (structure and behaviour) and their environment. One common model for implementing them is the well-known MAPE-k loop, a feedback control loop based on shared knowledge. Applying the models@run.time paradigm, this knowledge can be structured by a model, which is causally connected to the system. This model should represent the uncertain and time dimension of the knowledge. In this thesis, we consider that the knowledge comprises information related to the context, the

actions, and the requirements. In this thesis, we propose a metamodel[2] to represent the decisions made by the adaptation process over time (*cf.* Chapter 6). Additionnaly, we define a language, Ain'tea, to propagate uncertainty in the computation made during the adaptation process (*cf.* Chapter 5).

2.2 Model-Driven Engineering

⟨sec:back:mde⟩

Abstraction, also called modelling, is the heart of all scientific discipline, including computer science [Kra07]. 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 extra that complexify 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 ones, 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 [Che76] that is used to describe the data model for relational databases. In the web community, they use the ontology [Gru95] formalism to define the semantic web[3] [BHL+01]. The software engineering community applies the Unified Modelling Language (UML) [OMG17] to formalism software system structure or behaviour.

Extending this need for abstraction to all software engineering activities, practitioners have proposed the MDE methodology [Sch06; Ken02]. This methodology advocates the use of models, or abstractions, as primary software artefacts [WHR]. 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 [FR07; Sch06], the MDE methodology suggests to use models for all

[2] A metamodel is a model that defines another model (*cf.* Section 2.2).

[3] The Semantic Web is defined as en extension of the Web to enable the processing by machines.

the steps of software development and maintenance [Sch06; BCW17; HRW11; BLW05; HWR+11; AK03]: design, evolution, validation, etc. The core idea of this approach is two reduce the gap between the problem and the solution space [Sch06]. 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[4] 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 enables engineers to generate automatically software artefacts, such as documentation, source code, or test cases. Using these mechanisms, stakeholders can focus on their problem keeping in mind the big picture. A well-known example is the Structured Query Language (SQL) [fSta16] 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[Sch06; Ken02; BLW05]. First, using the same model, engineers can target different specific platforms. For example, the ThingML [HFM+16] language allow specifying the behaviour of the system through state machines. The same ThingML code can be deployed on different platform such as 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 [BK08], which will enhance the software quality. Finally, the models enable the separation of application and infrastructure code and the reusability of models.

However, the literature has also identified some drawbacks of the MDE approach [Ken02; BLW05; WHR+13; HRW11]. First, it requires a significant 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

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

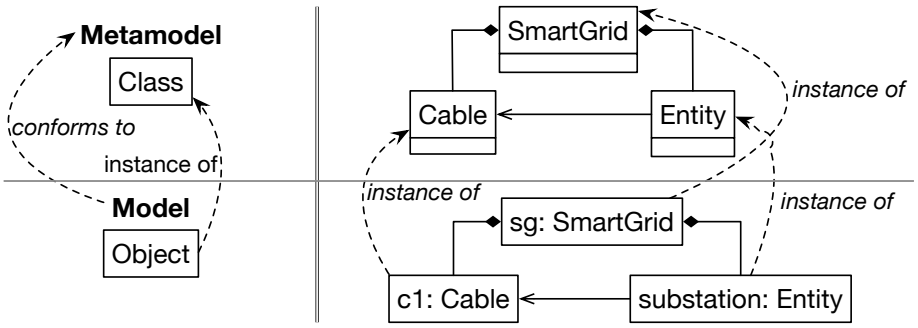


Figure 2.3: Difference and relation between metamodel and model

de:meta-model⟩

new methods such as NeoEMF [BGS+14; DSB+17], which enable the storage of large models, and MogwaÃŕ [DSC16], a query engine for large models. Third, this approach suffers from 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.,* [BLC+18] that define a generic omniscient debugger[5] for DSML. Third, introducing MDE in a software development team presents 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 [BJT05]. Also, they define the semantics rules and constraints to apply [Sch06]. They can be seen as

[5]Debuggers, generally, allow engineers to execute step-by-step programs, that is, forward. An omniscient debugger is also able to go backwards: to navigate back in the previous states of a program [Lew03].

models of models. In the MDE community, they are generally defined using the class diagram of the UML specification [OMG17], as shown in Figure 2.3. They, therefore, contain classes, named metaclass, and properties (attributes or references). In the language engineering domain, metamodels are used to define the concepts that a language can manipulate (*cf.* Section 2.3). Object Management Group (OMG)[6] define a standard metamodeling architecture: Meta Object Facility (MOF) [Gro16a]. It is set aside with Object Constraint Language (OCL) [Gro14], the standard constraint language to define constraints that cannot be specified by a class diagram, and XML Metadata Interchange (XMI) [Gro15], the standard data format to persist (meta)models.

Model Models capture some of the system characteristics into an abstraction that can be understood, manipulated, or processed by engineers or another system. They are linked to their metamodels through the conformance relationship as depicted in Figure 2.3. 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 classes of models [FR07]: development and runtime models. The former are abstraction above the code level. It regroups requirements, architectural, or deployment models. The latter abstract runtime behaviour or status of systems. They are the basis of the models@run.time paradigm, explained in Section 2.1.2. 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 metamodel.

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

2.2.3 Tooling

Tooling is an essential 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) [SBM+08]. It is the defacto baseline framework to build modelling tools within the Eclipse ecosystem. It embeds its metamodelling language, ECore [SBM+08; Fou10]. ECore is thus the standard implementation of Essential MOF (EMOF) [Gro16a], a subset of MOF that corresponds to facilities found in object-oriented languages. As written on the EMF-website[7], 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.,* [FNM+14; FNM+12], 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[8]. These systems contain small devices, like micro-controllers, that have limited 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, an efficient model (un)marshalling and cloning, a lazy loading mechanism, and 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) [FNM+14; FNM+12], a modelling framework specific to the models@run.time paradigm.

Hartmann extended this work and created the GreyCat Modelling Environment

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

[8] Definition of IoT by Gubbi *et al.,* [GBM+13]: "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."

(GCM)[9]. Using this environment, a developer can create high-scalable models, designed for models@run.time[HFN+14b; MHF+15], 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[10] [HFM+19; Har16]. In addition to time, metamodels defined by GCM can have an attribute with a value computed from a machine learning algorithm [HMF+19]. Based on the metamodel definition, a Java and a Javascript API are generated, to manipulate the model, *i.e.,* the temporal object graph.

2.2.4 Concepts used in this thesis

The models@run.time paradigm is a well-known approach to handle the challenges faced in adaptive system development. Plus, the GCM has been designed to design a data model, with time as the 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

⟨sec:back:sle⟩ As stated by Favre *et al.,* [FGL+10], software languages are software. Like traditional software, they need to be designed, tested, deployed, and maintained. These activities are grouped under the term Software Language Engineering (SLE) [Kle08]. 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 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 language, named modelling languages. Those models allow developers to implement a model. For example, we can argue that many

[9] https://github.com/datathings/greycat/tree/master/modeling
[10] https://greycat.ai/

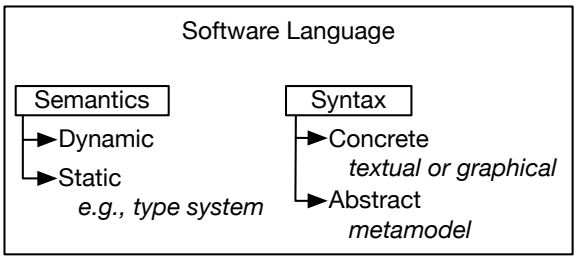


Figure 2.4: Composition of a software language

ound:sle:lang⟩

developers have already used the HTML language to implement a Document Object Model (DOM)[11] With the emergence of executable models[12], 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 kinds of languages.

Another way to classify software languages is by their scope [vDKV00]. General Purpose Language (GPL) are languages that can be used for any domain whereas Domain Specific Language (DSL)[13] that are restricted to a specific domain. Using a GPL, a developer can use it to implement a full software benefit from great tooling support. However, she may manipulate 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, DSLs are close to their problem domain [vDK98] but might suffer from poor tooling support [Voe14]. As for MDE, there are some research efforts to remove this disadvantage [BLC+18]. Using DSLs, developers can have simpler code, easier to understand and maintain [vDKV00; vDK98].

As depicted in Figure 2.4, Software languages are composed of two parts [HR04]: syntax and semantics. The syntax defines the element allowed in the language and the semantics their meaning. We can distinguish two kinds of syntax: the abstract one and the concrete one. The abstract one defines the different concepts manipulated

[11] "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." [W3C05]

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

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

with the language and their relationships. A metamodel can express it. The concrete abstract describes how these concepts are represented. It exists two categories of concrete syntax: graphical and textual. As for the syntax, there are two kinds of semantics: the static and the dynamic. The 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 constraints 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 [EB10], can generate the model automatically 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 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 describe the concept of random variables in probability theory. Then we describe what a probability distribution is and the properties that will be used in

this thesis. Finally, we will explain 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 dice. 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 dice 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:

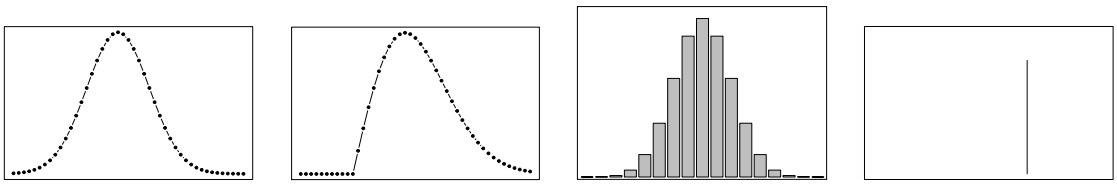
A random variable is a function from the sample space S to $\mathbb{R}$. In our example, we can define two random variables X and Y that represent, respectively, the minimum of the two dice and the sum of the two. We have thus: $X = min(i, j)$ and $Y = i + j$.

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 results of the rolling of n dice, $n \in \mathbb{N}$. 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 events are independent if the probability of occurrence of one does not impact the probability of occurrence of the others. Two events are disjoint 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 $P(A \cap B) = P(A) * P(B)$.

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



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

Figure 2.5: Probability distributions used in this thesis

:back:proba:examples⟩

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. We give an example of these distributions in Figure 2.5. The Gaussian and the Rayleigh distributions will represent continuous distributions. The Bernoulli and the binomial one represent discrete distributions. 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 the head and 0 tail.

Binomial distribution The binomial distribution represents the probability of success of a binary phenomenon over a set of trials. Figure 2.5c 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 [BHH05]. 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 [Met08]. 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 2.5a.

Rayleigh distribution Another distribution defined on a continuous domain ($[0;+\infty[$) is the Rayleigh distribution. It is often used for GPS positions [Bor13]. This distribution is defined using a unique parameter: a variance. We depict an example of this distribution in Figure 2.5b.

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 2.5d. 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

Motivating example: smart grid

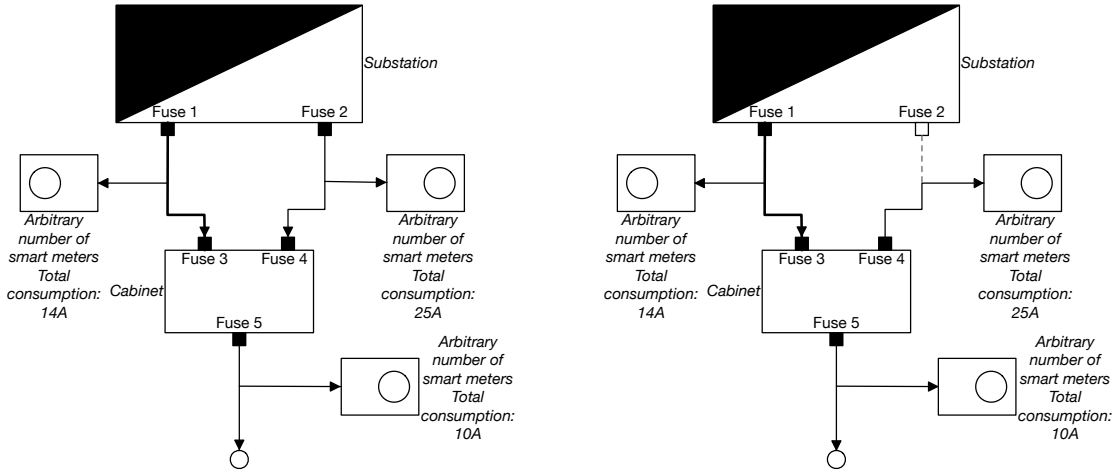
chapt:example⟩

Contents

In this chapter, we present our motivating example, based on the Luxembourg Smart Grid. This example has been built in collaboration with our partner, Creos S.A., the Luxembourgish grid manager. We first several examples of long-term action and then we motivate the need for a language with uncertainty as a first-class citizen with code samples.

In this chapter, we exemplify the impacts of uncertain data and long-term actions on a smart grid system. We built these example with Creos Luxembourg, our partner[1]. We first give an overview of a smart grid. Then, we focus on cable load estimation and the impacts of not taking uncertainty into consideration. Then, we concentrate on the reconfiguration feature of smart grids and the effect of long-term action. We also extend this example with other examples from other domain such as cloud infrastructure.

3.1 Smart grid overview



(a) Electricity flow 1 (b) Electricity flow 2

Figure 3.1: Two different electric flows for a same grid topology

ple:sg-overview:topo⟩

The National Institute of Standards and Technology (NIST) defines a smart grid as "a planned nationwide network that uses information technology to deliver electricity efficiently, reliably, and securely"[2]. Conceptually, a smart grid is composed of different entities, like smart meters, cabinets (connection points of cables) and power substations. These entities are connected, forming a network, and able to exchange information using different technologies [HFK+14b; HMF+16].

[1] Creos Luxembourg is the main grid operator in the country

[2] `https://www.nist.gov/engineering-laboratory/smart-grid/smart-grid-beginners-guide`

The network is the connection linking the smart grid entities by means of physical cables. Every cable has a maximum load depending on its diameter and the used material. Figure 3.1 depicts an example of such a network. This network is composed of one substation, one cabinet, and an arbitrary number of smart meters. Every cable has two fuses (to connect or disconnect the cable), one at each endpoint. By opening or closing fuses, one can influence the electricity flow through the network. Figure 3.1a and 3.1b show two possible electricity flows over the same physical network. We depict closed fuses in black and the direction of the electricity flow using black arrows.

Smart meters continuously measure electricity consumption and periodically report it to a central data centre. Based on this information, together with the grid topology, the electric load of cables can be computed. If the reader wants to go more into the details of this computation, we invite him or her to read [HMF+16]. The flow has a big impact on the load cables. In Figure 3.1, we depict two different flows for the same grid topology. In both cases, the measured electric consumption are equal. However, in Figure 3.1a the load on the left cable (thick line) equals $\frac{14+25+10}{2} = 24.5$ whereas it equals $14 + 25 + 10 = 49$ in Figure 3.1b.

The central system monitors electric loads to avoid any overload in the network, implementing the adaptive system vision. In the remaining part of this document, we refer to this goal as the "*no overload*" policy. They have two action points: either on the production side or the consumption side. They can reduce or increase the production by (dis)connecting production unit or the consumption by modifying the maximum permitted consumption. We called these actions: *reduce production*, *increase production*, *reduce amps limit* and *increase amps limit*. However, as all adaptive systems, smart grids are prone to failures [HFK+14a]. In case of a failure, 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?

3.2 Data uncertainty

3.2.1 Impacts of ignoring data uncertainty

Although power grids are becoming more and more automated, today human interventions are still the norm. For example, most fuses are manually modified by technicians rather than automatically reconfigured. The states of fuses are thus manually documented by technicians on the field. This of course results in mistakes. One way to address this problem is to take uncertainty into consideration. This means considering fuse states as uncertain.

As a consequence, this uncertainty propagates to the load calculation formulas, which depend on the fuse states. If the uncertainty of fuse states is not considered, it exists a non-null probability that the observed phenomenon does not reflect the real situation. For example, as we have seen in the previous subsection, a modification of the electricity flow may impact the load of a cable with a factor of 2. Cable load approximations are used to detect cable overloads and to reconfigure the network if necessary. By not considering uncertainty, wrong reconfigurations might be triggered, which could worsen the situation.

3.2.2 Managing uncertainty is not effortless

In the following, we describe how uncertainty is commonly handled by application developers using current state-of-the-art approaches. We show, through code samples, the limitations of these approaches and why we think that these limitations can be addressed by integrating uncertainty management directly at the language level. In the code excerpts below, we compute the average cable load over the whole grid based on uncertain cable loads. A complete version can be found on the GitHub repository. All codes contain at least two classes: *SmartGrid* and *Cable*. The former contains two fields: an array of Cables named *cables* and a function to compute the average load of cables, named *compute_avg_load*. The latter contains one field: an uncertain number which represents the uncertain load.

Manual implementation One approach for managing uncertainty is to manually implement all the required features. Listing 3.1 illustrates this in Python. In addition to the *SmartGrid* and *Cable* class, the excerpt contains an additional one: *UNumber*.

As we can see, this approach comes with several drawbacks. First, developers are required to have deep knowledge of probability theory. For example, uncertainty can be represented by a normal distribution identified by its mean and variance as shown in the constructor (*__init__*), line 2. One needs knowledge on how to represent it, how to add two normal distributions, etc. For example, here we overload the sum and the division operators for two normal distributions (*__plus__* and *__div__* methods).

Second, manual implementation inevitably increases the size of the code base. This will de facto augment the risk of errors in the code. Plus, the code will be more difficult to maintain afterwards.

Third, although some languages offer the possibility to overload operators, some typing errors can only be detected at runtime. For instance, since Python is dynamically typed, performing an addition operation between two types of uncertain numbers (*i.e.,* represented by two classes) fails and raises an exception only at runtime. Whilst, statically typed languages such as C# can detect such typing errors at development time. Nonetheless, the returned exception message would not be particularly meaningful, as can be seen on the example of C#: *Operator '+' cannot be applied to operands of type UNumber1 and UNumber2*.

-from-scratch⟩

```
class UNumber:
  def __init__(self, mean=0, variance=0):
    [...]

  def __add__(self, other):
    [...] # typing error management + casting other to UNumber if it is a Number
    return UNumber(self.mean + other.mean, self.variance + other.variance)

  def __div__(self, other):
    [...] # typing error management + casting other to UNumber if it is a Number
    value = ((self.mean/other.mean)
                        + (self.mean*other.variance)) / pow(other.mean, 3)
    variance = (self.variance / other.mean) +
             (pow(self.mean, 2) * other.variance) / pow(other.mean, 4)
    return UNumber(value, variance)
```

```
class SmartGrid:
  [...]
  def compute_avg_load(self):
    sum_load = 0
    for c in self.cables:
      sum_load += c.load
    return sum_load / len(self.cables)

class Cable:
  def __init__(self, id, load=UNumber(0, 0)):
    [...]
```

Listing 3.1: Manual management of uncertainty in Python

Using existing libraries for probability theory Since uncertainty management relies on probability theory, another approach is using a library implementing common probability distributions. Staying in Python, there is a widely used library called SciPy[3] and in particular the *stats* module, defining different probability distributions. Whereas these libraries are suitable to compute different properties of probability distributions, it remains the responsibility of developers to define how to perform arithmetic operations on distributions, as depicted in Listing 3.2.

Using such libraries does not prevent developers to have a deep understanding of the probability theory. They have been designed to help them manipulating probability distributions, but they still required knowledge about them. For example, as shown in Listing 3.2, one may access the mean and the variance of a normal distribution. However, she or he needs to know how two normal distributions can be added. This also increases the code base and thus is prone to errors. Finally, as it is a library, typing errors will still remain either detected at runtime or not helpful for developers.

⟨lst:limit-lib-proba⟩
```
from scipy.stats import norm

def add_gaussian(g1, g2):
    [...]
    return norm(g1.mean() + g2.mean(), g1.var() + g2.var())
```

[3] https://www.scipy.org/

```
def div_gaussian(g1, g2):
   [...]

class SmartGrid:
    [...]
    def compute_avg_load(self):
        sum_load = 0
        for c in self.cables:
            sum_load = add_gaussian(sum_load, c.load)
        return div_gaussian(sum_load, len(self.cables))

class Cable:
    def __init__(self, id, load=norm(0, 0)):
        [...]
```

Listing 3.2: Limitation using a probability library (Python)

Using existing libraries for uncertainty Another commonly used approach is to use existing libraries for data uncertainty. For example, in Python, using the *uncertainties* [LEB18] library, a developer can manipulate uncertain floats using variables of type *ufloat*. The uncertainty propagation is handled transparently. In Listing 3.3, we depict the code of the load average computation using this library. The only difference from the previous code snippet, is the missing of the *UNumber* class, not required anymore, and the use of the *ufloat* type for the cable load variable.

Using a library can tackle the two first drawbacks existing in the previous approach: required knowledge of probability theory and code complexity. Indeed, the implementation of the probability theory and its complexity are encapsulated inside the library. We can argue that the library will be well tested and documented. However, similarly to the previous example, this approach does not extend the type system.

uncertainties⟩
```
from uncertainties import ufloat

class SmartGrid:
  def __init__(self):
    self.cables = []

  def compute_avg_load(self):
    [...] # same as in Listing 3.1

class Cable:
  def __init__(self, id, load=ufloat(0, 0)):
```

```
    [...]
```

Listing 3.3: Managing uncertainty in Python using uncertainties [LEB18]

Using a probability programming framework Developers can also use a probabilistic programming framework like Infer.NET [MWG+18] or PyMC3 [SWF16]. These frameworks allow defining complex probabilistic models and applying inference algorithms. Contrary to previously described libraries, these frameworks put an effort to set a proper API. They can be thought as internal domain-specific languages (DSL)[Fow10][4]. In Listing 3.4, we show how the cable load can be computed using the Infer.NET framework. This approach exposes probability as a first-class language citizen. Hence, developers are expected to manipulate probability distributions and not uncertain data types. This framework relies on an inference engine to transparently combine probability distributions. Using such an approach requires a decent acquaintance with probability theory. We can assume that they are heavily tested, which limits the number of errors. Plus, in this framework they map arithmetic operators to the combination of probability distributions: developers do not require to create additional work as with the library approach. But, as it is not implemented at the language level, the typing system is not extended. Errors will thus either be raised at runtime or not be helpful for developers.

:limit-proba-prog-fw⟩

```
public class SmartGit {
  public List<Cable> cables { get; private set; }
  private readonly InferenceEngine inference;

  public Gaussian computeAvgLoad() {
    Variable<double> sum = Variable
                           .GaussianFromMeanAndVariance(0,0)
                           .Named("sum");
    int i = 0;
    foreach(Cable c in cables)
      sum = (sum + c.load).Named("sum" + i);
      i++;
    Variable<double> result = (sum / cables.Capacity).Named("AvgLoad");
    return (Gaussian) this.inference.Infer(result);
```

[4]Martin Fowler defines a DSL as "a computer programming language of limited expressiveness focused on a particular domain"[Fow10]. An internal DSL is defined inside a general-purpose language, using only a subset of its concept.

```
  }
}

public class Cable {
  public Variable<double> load { get; set; }
}
```

Listing 3.4: Limitation using a probability programing framework (C#)

Sum-up Using state-of-art solutions to manage uncertainty, developers have three possibilities: manual implementation, using existing libraries or using a probability programming framework. As detailed in this section, these approaches will cope with at least one of these drawbacks: developers require a deep understanding of the probability theory, code base will increases which augments the risk of errors, and the type systems may detect errors at runtime or do not provide helpful messages.

3.3 Long-term action

3.3.1 Examples

m:intro:motiv⟩ **Long-term actions**

elayed_action⟩ In this section, we motivate the need to reason over long-term actions. 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.

Long-term action examples In the previous sections, we have claimed that adaptation processes should handle long-term 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

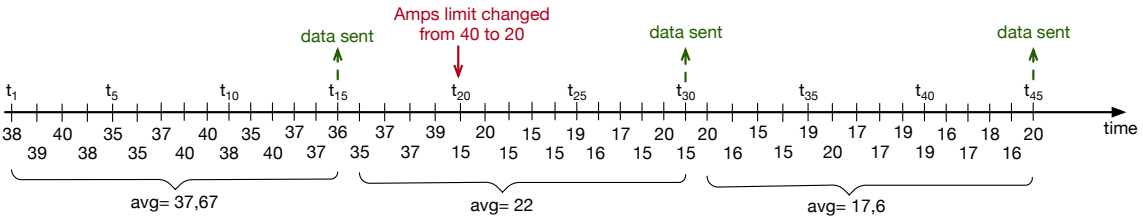


Figure 3.2: Example of consumption measurement before and after a limitation of amps has been executed at t_{20}.

ng-action-amps-limit⟩

remotely controlled. One example is the fuses that still need to be opened or closed by a human operator. Open and close actions in the Luxembourg smart grid both imply technicians who are contacted, drive to fuse locations 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 execute the modifications. For example, our industrial partner asks us to consider that each fuse modification takes 15 min whereas any 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[5] Creos S.A. envisages controlling remotely amps limits of customers in its smart grid project. 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 3.2, we depict a scenario that shows the delay between the action is

[5]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.

executed and the impacts are measured. Each time point represents one minute, with the consumption at this moment.

Let us imagine a customer who has his or her limit set to 40 amps[6] and consumes near this limit. We consider that data is sent every 15 min. After receiving data sent t_{15} and processing them, the adaptation process detects an overload and decides 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[7], the action is received and executed at t_{20}. At t_{30}, new consumption data is sent, and equals 22 amps. Two situations are now possibles. First, this reduction was enough to fix the overload. Even in this idealistic scenario, the adaptation process must wait in the worst case for 15 min (t_{30} - t_{15}) to observe 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 observing the incident as solved and knowing that the decision fixed the incident, the adaptation process needs to 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 long-term actions is to remove a machine from a load balancer, for example during a scale down operation. Scale down operations 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 [WBR11], Wang *et al.,* present a load-balancing algorithm. In their evaluation, they present the figure depicted in Figure 3.3 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

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

[7]The smart grid is not built upon a fast network such a fiber network.

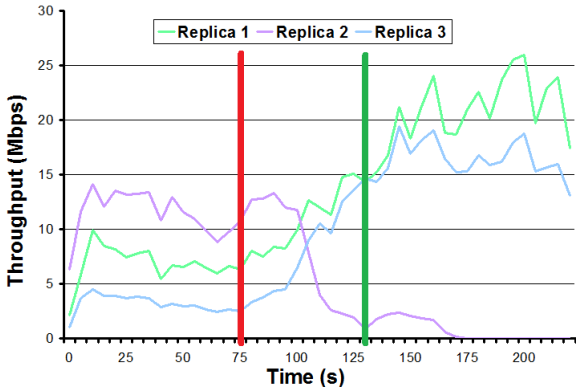


Figure 3.3: Figure extracted from [WBR11]. The red bar depicted the moment when Replica 2 stop receiving new connections. The green one represents the moment where all the rules in the load balancer stop considering R2. Despite these two actions, the throughput of the machine does not drop to 0 due to existing and active connections.

xample-load-balencer⟩

100s between the red bars and the moment where R2 stop being active.

This example shows a long-term 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 long-term 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 expected effect. We detail this point in the following section.

The need to consider effects One may argue that action statuses are already integrated into the knowledge. For example, the OpenStack Watcher framework stores them in a database[8], 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 to 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 will the load 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.

Sum-up 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

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

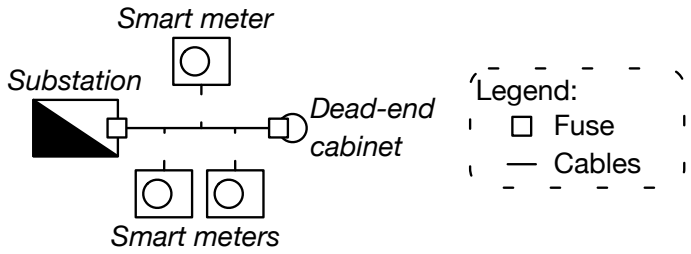


Figure 3.4: Simplified version of a smart grid

⟨fig:tkm:excerptSG⟩

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 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.

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 [MBJ+09; BBF09]. 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 synthesise its internal state as well as its surrounding environment.

3.3.2 Use case scenario

⟨sec:tkm:intro:uc⟩

Excerpt of a smart grid Figure 3.4 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 consumption data at the same timestamp. For this example, we consider one requirement: minimizing the number of overloads. To achieve this, 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 [HFK+14a]. 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?

4

State of the art

⟨chapt:sota⟩

Contents

This chapter reviews works related to the one presented in this dissertation. Two research questions drove this review. The first one aims at investigating state-of-the-art solutions that model adaptive systems to see those that consider long-term action. With the second research question, we search current approaches that handle data uncertainty. Our review shows that none of the studies performed until now take into account long-term action. Additionally, it indicates that different solutions exist to model data uncertainty, but some efforts need to be performed to provide solutions at a higher level.

4.1 Review methodology

This review aims at answering two global research questions. To help us answer them, we split them into three sub-research questions.

Research questions The first research question has been set to study the presence of long-term actions in the modelling layer: (**RQ1**) do state-of-the-art solutions that model adaptive system allow representing and reasoning over long-term actions (design time and runtime)? We split it into the following sub-questions:

- **RQ1.1**: How current approaches model the evolution of the context or the evolution of systems (structure or behaviour) over time?
- **RQ1.2**: What are the solutions that model actions, their circumstances and their effects over time at design time and runtime? Can this model be processed or navigated automatically?
- **RQ1.3**: What are the solutions that enable the reasoning over the evolving context, structure, or behaviour of systems? Do they also enable reasoning over running actions and their effects?

With RQ1.1, we investigated the approaches that model the context, structure, or behaviour of adaptive systems. The second one filters those that also consider long-term actions. Finally, we use RQ1.3 to list solutions that provide a technique, such as an algorithm, to reason over evolving context, structure, or behaviour.

With the second research question, we seek for modelling solutions that consider uncertain data and its propagation: (**RQ2**) do state-of-the-art solutions allow modelling uncertainty of data and its manipulation (propagation, reasoning over)? The three sub-questions are:

- **RQ2.1**: What are the categories of uncertainties that have been addressed by the literature?
- **RQ2.2**: How the uncertainty of data is modelled?
- **RQ2.3**: What are the solutions that enable an imperceptible propagation and reasoning over uncertainty?

We set RQ2.1 to identify the different kinds of uncertainties that bring challenges in software engineering. Then, we search for the technique to model uncertainty, with

a particular interest in data uncertainty with RQ2.2. Lastly, we review approaches that allow developers to propagate uncertainty without writing specific code for that and to reason over uncertainty using RQ2.3.

Methodology To review the literature, we applied a technique inspired by the snowballing approach [Woh14]. But, due to limited resources, we did not fully implement it. The methodology advocates the use of bibliography (backward navigation) and papers that cite (forward navigation) the selected ones to navigate in the literature. Each article should be evaluated according to a set of inclusion and exclusion criteria. And a starting set should be defined.

In our case, we use the bibliography of the papers that ground in this thesis as the starting set (*cf.* Section 1.4). Then, we apply the backward navigation for a subset of them. Then, we select the paper to add in this review according to a set of inclusion and exclusion criteria. To be picked, a paper should satisfy all inclusion criteria and should not fulfil any of the exclusion criteria. These criteria are the following:

- Inclusion criteria (IC):
  - **IC1**: The paper has been published before August 9 2019.
  - **IC2**: The paper is available online and written in English.
  - **IC3**: The paper describes a modelling approach that abstract the context, the structure, or the behaviour of a system, an approach that enables to reason or navigate through a temporal model, an approach that describes a solution used to engineer a adaptive system, an approach that handles uncertainty, or an approach that helps to manipulate probability distributions.
- Exclusion criteria (EC):
  - **EC1**: The paper has at most four pages (short paper).
  - **EC2**: The paper presents a work in progress (workshop papers), a poster, a vision, a position, an exemplar, a data set, a tutorial, a project, or 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.

The first two inclusion criteria are accessibility criteria: they guarantee that the paper is accessible for any reader of this document. With the third one, IC3, we can include all papers that can be used to answer our research questions. We define the exclusion criteria to keep only papers that have been published in a peer-reviewed venue, and that present an approach.

In this review, 412 papers have been processed, and 84 kept for the review. We report our selection results in an Excel file publicly available on GitHub [1]. Results have also been exported to Comma-separated values (CSV) files for those who cannot open Excel files.

4.2 Results RQ1: long-term actions

sota:results:actions⟩ In this section, we detail our findings regarding the first research questions. First, we detail all approaches that propose a solution to model the evolution of a system's context, structure, and behaviour. Then, we list approaches that model actions. Before summarising and answering the research question, we list solutions that model and reason over evolving context, behaviour, and behaviour, *i.e.,* that implement an adaptation process.

4.2.1 Modelling the evolution of system's context, structure, or behaviour

Different categories of approaches exist to represent the context, structure, or the behaviour of a system. In this section, we detail our findings with an overview given in Table 4.1.

Modelling paradigm In the MDE community, researchers defined the models-@run.time paradigm to implement an adaptation process [BBF09; MBJ+09]. This approach is based on a runtime model that reflects the current state of the system.

[1] https://github.com/lmouline/thesis/tree/master/sota/src

Approach	Reference
Modelling paradigm	[BBF09; MBJ+09; HFN+14b; HFN+14a]
Formal model	[WMA12; WHH10; BK11]
Low level model	[MS17]
Object-based model	[HIR02; HFK+14a; TOH17]
Goal model	[CvL17; IW14; MAR14; CPY+14; BPS10]
State machine	[HFK+14a; IW14; ARS15; AGR11; BdMM+17; BBG+13; MCG+15; FGL+11; GS10; DMS18; DBZ14; ZGC09; GPS+13; TGE+10]
Sequential diagram	[TOH17]
Component model	[DL06; DBZ14; GCH+04; FMF+12]
Trace model	[Mao09]
Graph model	[KM90; GvdHT09]

Table 4.1: Approaches to model systems' context and behaviour (RQ1.1)

actions:rq1.1⟩

It can contain information about either the context of the system, its behaviour, or its structure. Moreover, there is a causal link between the model and the system: modifications of the model, made by a stakeholder or a process, trigger modifications in the system. For example, changing the status of a fuse in a model that reflects a smart grid triggers the action to open or close it. Hartmann *et al.,* extended this paradigm to introduce a temporal dimension [HFN+14b; HFN+14a]. It allows designers to store not only the current state of the system but also its past (previous states) and future (predicted). In this thesis, we will use this extension of the paradigm to build our knowledge model, and more precisely to represent long-term actions.

Formal model In [WMA12], Weyns *et al.,* defined a formal model for adaptive systems, called FORMS. Their goal was to establish a reference model, which can be used for discussion or implementation. For self-organisation systems, the literature provides another formal model: MACODO [WHH10]. It uses the Z language [dL04] to formalise the context of the system following the set theory and the first order predicate calculus. The behaviour is formalised with what they call *laws*. A law is a function from one set to another. The third formal model found in our review was specified by Bartels and Kleine [BK11]. This one uses Communicating Sequential Process principles [Hoa78], a formalism designed for reactive and concurrent systems.

However, none of these models includes a time dimension, relevant to abstract long-term action.

Object-based model One category of approach found in the literature is object-based models. These models follow object-oriented principles. First, Henricksen *et al.,* defined a model for pervasive computing systems[2]. In their model, some entities are linked to their attributes through uni-directional association. These associations can be dynamic (can evolve) or static (do not change over time). A dynamic association can also be temporal. In this case, the entity can have several attributes with a timestamp attached. Second, Hartmann *et al.,* use a temporal model to store the context and its history of a smart grid system [HFK+14a]. This model is based on their extension of the models@run.time described above [HFN+14b; HFN+14a]. Third, Tahara *et al.,* [TOH17] use the Maude language [CDE+02], to represent the context. Among these three solutions, only the last one does not include a temporal dimension.

Goal model Goal modelling is a technique used by several contributions in our findings [CvL17; IW14; MAR14; CPY+14; BPS10]. This technique, mainly used in requirement engineering, represents the different goals of an application. By modelling the requirements of a system, we can argue that they *de facto* represent the context by the goals that are achieved or not. Moreover, in [CvL17], the authors add a satisfactory rate on each goal. However, these methods do not include a time dimension.

State machine State machines have the capacity to abstract, in the same model, the context and the behaviour of a system. The different states represent the context while the transitions between the states abstract the behaviour. [HFK+14a; IW14; ARS15; AGR11; GPS+13] use the Final State Machine (FSM) formalism. For example, Hartmann *et al.,* abstracts the behaviour of a smart meter in [HFK+14a]. In [GPS+13], the authors represent the functionalities of the system and their impact with states. And, transitions abstract the different execution flow between the different

[2]A pervasive system is composed of cheap and interconnected devices that are ubiquitous and can support users' tasks.[HIR02]

functionalities. In [BdMM+17; BBG+13], authors apply the labelled transition system [Kel76]. When authors want to consider the stochastic behaviour of the system, then they use probabilistic state machines In [BdMM+17], the authors extended the model with probabilities, which represent the probability for a transition to be executed. Another strategy is to use Markov Chain [MCG+15; FGL+11; GS10; DMS18] A Markov chain can be thought as a FSM with probabilities attached to the transition. In [MCG+15], authors also add information regarding current actions being executed with their progress status. But, no history is kept, the information is lost when actions finish. Other approaches use state machine without specifying the formalism used [DBZ14; ZGC09; TGE+10]. Tajalli *et al.,* use two state machines: one to represent the system context and behaviour, and another one to represent the adaptation mechanism.

Sequential diagram Through our review, we find one approach that uses a sequential diagram to represent the behaviour of the system [TOH17]. However, nothing is mentioned regarding the context of the system and its history.

Component model In order to represent the context of a system, one can use a component model. This model is at the architecture level and described the different entities (component) that compose a system with their interactions. Four contributions apply this technique in our review [DL06; DBZ14; GCH+04; FMF+12]. To also represent the behaviour, some have extended this model with a state machine description in [DBZ14] or with annotation in [GCH+04]. However, no time dimension is considered in these approaches.

Trace model Context and behaviour of a system can be inferred by analysing its logs. In [Mao09], researchers defined an approach to create a model that reflects the runtime state of the system. However, this approach does not keep the history of the system.

Graph model Finally, the last technique used to represent the context of a system is to use a graph model [KM90; GvdHT09; MS17]. In the former, nodes represent process units, and edges the communication between them. In the second one, nodes represent the possible configurations of the system, and the edges represent the

actions to reach a configuration. Only the latter include a time dimension. The graph represents the different configuration over time of the system. Plus, some meta-data about previous adaptations (*e.g.,* average time in this configuration) are added. The latter defined a temporal graph [MS17]. Their temporal graph is a graph that is augmented with two functions. One function serves to indicate if a graph element, a node or an edge, exists for a given period. The other can retrieve the value of a graph element for a given period. Using this temporal graph, one can define a model that abstract long-term actions. However, in our work, we use another temporal graph definition provided by Hartmann *et al.,* and implemented in our research group [HFM+19].

Sum up Different approaches are used in the literature to represent the context, the behaviour, or the structure of systems (*cf.* Table 4.1). However, only a few can be used to keep the history of this information [HFN+14b; HFN+14a; MS17; TOH17; HIR02; HFK+14a]. This feature remains a crucial concern to represent long-term actions as information about delayed effects and previous circumstances (next and previous context of an action). In the next section, we detail approaches that model actions.

4.2.2 Modelling actions, their circumstances, and their effects

In Table 4.2, we regroup the different approaches of our review that model actions. In this section, we describe the different categories that we identified.

Rule-based approach One solution to model actions is to use a rule engine [TOH17; ARS15; BCG+12; GHP+08; PFT03; GCH+04]. A condition and an executable code characterise rules. The executable code is executed if the current state of the system meets the condition. Conditions can thus serve to abstract the circumstances of actions and the executable code as its effect. However, this information is available at design time and lost during the execution. Moreover, these approaches do not allow the representation of the side effects of an action. For example, changing the fuse state has a direct effect on the fuse state. But it also impacts the power grid

Approach	Reference
Rule-based	[TOH17; ARS15; BCG+12; GHP+08; PFT03; GCH+04]
Sate machine	[ARS15; IW14; HFK+14a; MCG+15; FGL+11; DBZ14; ZGC09; GPS+13; TGE+10]
Goal-modelling	[MAR14; BWS+12; BPS10]
Programming language	[CG12]
Event-Condition Action	[DL06; CDM09; PBC+11]
Model transformation	[CPY+14; KM90]
Formal model	[WHH10; BK11; CMR15]
Dynamic Software Product-Lines	[GS10; CCH+13]
Graph model	[GvdHT09]

Table 4.2: Approaches to model actions, their circumstances, and their effects (RQ1.2)
actions:rq1.2⟩

load. We can notice two exceptions in our review: [TOH17] and [ARS15]. In both cases, rules are used to trigger a state modification in a sate machine. We explain the advantages and disadvantages of the state machine approach in the next paragraph.

State machine Several approaches use a state machine to represent the adaptation mechanism [ARS15; IW14; HFK+14a; MCG+15; FGL+11; DBZ14; ZGC09; GPS+13; TGE+10]. States represent the state of the system, and the transition abstract the execution of actions. One advantage of this approach is that they represent both the circumstances and the effects of actions. Additionally, it can be used to represent actions at design time and runtime. But this link remains at a high-level. An entire state is considered as the circumstance or the effect of an action while, in most cases, it just a subset of the elements of the state that triggers the action or is affected by it.

Publish/Subscribe approach In our review, we found one approach that uses the publish/subscribe mechanism to trigger actions [BdMM+17]. The circumstances are thus modelled with the condition of the consumer. In this case, the action is a script. The effects are thus spread, and cannot be navigated. Moreover, this solution only represents the actions at design time and not their executions.

Goal-modelling In addition to the goals of the system, goal models offer the capacity to represent the actions that can achieve them. Three approaches have used this ability to represent action in their model [MAR14; BWS+12; BPS10]. Baresi *et al.,* extended goal model with *adaptive goals*. These goals contain a condition, objectives (to weaken or enforce some goals), and *actions* (here, an action modify goals or operation-executable code- in the goal model). Here, effects can be seen as the fulfilment of a goal. And a condition is when a goal is not satisfied anymore. However, this information is not kept over time. Furthermore, no information about the runtime execution of the actions is kept.

Programming language Among our findings, one approach defined a language to define actions: [CG12]. This language allows developers modelling their actions, with their conditions, and their effects. However, the language does not include a temporal dimension. Plus, the language is suitable to describe actions at design time but provide no mechanism to track them during their execution.

Event-Condition Action To trigger adaptation, one can use the Event-Condition Action approach: the action is triggered if an event respects the condition. In our review, all the works use this methodology in order to weave or remove aspects of the program, following Aspect-Oriented Programming[3] [CDM09; PBC+11; DL06]. However, there is no temporal dimension. Plus, this solution is suitable to describe actions at design time but does not allow navigation through runtime information.

Model transformation Following the models@run.time, one way to adapt a system is to modify the model to trigger the actions. One way to do this, is to use the Query/View/Transformation [Gro16b] approach as Chen *et al.,* did in [CPY+14]. Here, the circumstance and the effects are represented at the model level in the query and the transformation part. When the context is represented as a graph, actions will thus be modelled as graph modifications. In [KM90], authors represent an action by adding or removing graph elements (node and edge). However, these solutions do not take into account any time dimension, side effects or runtime information.

[3]Aspect-Oriented Programming is a programming paradigm that prones the separation of concern. For that, a program is seen as a set of aspects, each aspect implementing one concern.

Formal model In our review, we found three formal models [WHH10; BK11; CMR15]. First, the MACODO formalisation [WHH10] that defines actions as functions from one set to another. These functions represent actions to reorganise a system: adding, removing, or merging group of components. Second, in [BK11], the authors use the process calculus Communicating Sequential Process, which allows engineers formalising reactive and concurrent systems where atomic *events* are handled by *processes* (an infinite transition system). Actions here correspond to a modification of a component configuration in response to an event. Last, Cimatti *et al.,* defined a planning algorithm which includes time [CMR15]. In their algorithm, they formalise actions as elements with a field to store the execution time, possibly uncertain. None of these models adds a temporal dimension or represents the effects of an action. Plus these solutions do not model the execution of actions.

Dynamic software product-lines One approach to represent the possible adaptation of a system is to use a Software Product-Lines model. Engineers model all the possible variations of a system. Then, at runtime, the system selects the one that can achieve the requirements in the current context. This approach is referred to as Dynamic Software Product-Lines and used by two approaches in our findings [GS10; CCH$^+$13]. Here actions can be executed to achieve the selection of the different variation point. Effects are thus represented as the new configuration selected. However, these solutions do not include a time dimension or cannot represent runtime information.

Graph model Finally, the last approach found in our review is to use a graph model. In [GvdHT09], researchers use a graph where nodes represent possible configurations and edges represent the modification of the configuration, using our wording an action. However, there is no time dimension, and only design time information is modelled.

Sum up As shown in Table 4.2, the literature provides different solutions to model actions. However, none of them includes a temporal dimension. Thus, even if they represent effects, they cannot describe them over time. For example, they will be able to model that a new server will be added, but not when. Moreover, these solutions mainly remain at design time, except for those that use a state machine.

No information concerning the runtime, like the status of the execution of an action, the runtime values or the effects, are not represented. One limitation is that they do not represent side-effects over time. For example, they will represent the addition of a server but not the effects on the bandwidth or the workload. Additionally, as they do not represent runtime information, no autonomous solution can be employed to detect any unknown effects of an action to the system. The last limitation is the representation of circumstances. In the approaches of the review, they are mainly represented as the condition that triggers the action. Even if a human can guess it after, there is no direct link between the action execution and the circumstances. No model can allow automatic navigation over this time-related information.

4.2.3 Reasoning over evolving context or behaviour

In this section, we review the solutions that enable reasoning over evolving context or behaviour. That is solutions that can be used to implement adaptive systems. Table 4.3 gives an overview of our findings, detailed in the rest of this section.

Model-based approach Following the MDE methodology, one approach to reason over an evolving context or behaviour is the models@run.time paradigm [BBF09; MBJ+09]. Hartmann *et al.,* extended it to include a temporal dimension [HFN+14b; HFN+14a] for reasoning over the history of a system. Approaches detailed in [BdMM+17] and in [CPY+14] follow this paradigm. However, in their process, they do not consider long-term action, especially those that are under execution of with effects in the near future.

Rule-based adaptation In our review, some approaches employ a rule-based mechanism to define the adaptation mechanism [ARS15; TOH17; GHP+08]. However, they only reason over the context or the behaviour, not on the running actions or their future effects.

Architecture-based adaptation Architecture-based adaptations adjust the architecture of a system to achieve requirements. In our review, different approaches use this mechanism [CG12; GCH+04; GvdHT09; FMF+12]. For example, Cheng *et al.,* define a language to design actions for architecture-based adaptation. Also, the

Approach	**Reference**	Reasoning over long-term action
Model-based	[BBF09; MBJ+09; HFN+14b; HFN+14a; BdMM+17; CPY+14]	None
Rule-based	[ARS15; TOH17; GHP+08]	None
Architecture-based	[CG12; GCH+04; GvdHT09; FMF+12]	None
Simulation-based	[HFK+14a]	Consider effects of actions to select those to execute. Do not consider running actions
Formal model	[WMA12; IW14; WHH10; BK11]	None
Complex event processing	[AFR+10]	None
Graph model	[MS17; KM90]	None
Aspect Oriented Programming	[GB; DL06; CDM09; PBC+11; FA04; PFT03; MBN+09]	None
Component-based	[DL06]	None
State machine	[MCG+15; FGL+11; DBZ14; ZGC09; GPS+13; TGE+10]	Effects modelled but do not contain any runtime information about actions execution
Dynamic software product-lines	[GS10; CCH+13]	None
Requirement-driven	[BPS10]	None
Extension of MAPE-k	[MBE+11]	None

Table 4.3: Approaches to reason over evolving context or behaviour (RQ1.3)

actions:rq1.3⟩

adaptive mechanism can compute a reconfiguration script by comparing the current component model with the expected one [FMF+12]. However, none of these solutions considers running action and their future effects.

Simulation-based adaptation In our review, one approach applies a simulation-based approach [HFK+14a]. In their work, the authors simulated different sequences of actions, evaluate and select the optimal one regarding the requirements. To perform this approach, they have to know the look ahead and consider the impact of each action on the system. However, they do not reason over actions being executed or their future effects.

Formal model In our review, four formal models describe adaptive systems with their adaptation mechanism [WMA12; IW14; WHH10; BK11]. For example, Iftikhar and Weyns use a state machine formalism to specify self-adaptive systems [IW14]. But, none of the formal models includes a time dimension, which would allow stakeholders to consider running long-term actions.

Complex event processing Among our findings, one is using an approach based on a complex event processing engine [AFR+10]. Here, actions are triggered in response to an event, as much complex as necessary. Using this approach, one cannot reason over running actions and their future effects.

Graph model Two approaches in our findings employ a graph model [KM90; MS17]. These approaches reason over a graph to trigger an adaptation process. In [MS17], authors use a temporal graph algebra, which can store the history of the context. But, none of them includes running actions in their model.

Aspect-oriented programming From what we see in our findings, aspect-oriented programming is an approach heavily used by researchers [GB; DL06; CDM09; PBC+11; FA04; PFT03]. More specially, they use dynamic aspect-oriented programming: aspects are automatically weaved and removed aspects in a software. Besides, Morin *et al.,* uses aspect-oriented modelling [MBN+09]. Part of the MDE methodology, this approach models the different aspects with their pointcuts[4]. However, none of them

[4]A pointcut is part of the software where an aspect can be weaved or removed.

allows reasoning over actions being executed or their future effects.

Component-based adaptation One approach uses a component model as a basis to reason for the adaptation mechanism [DL06]. Here, actions modify the configuration of a component to adjust the behaviour or the context of the system. But, in this approach, running actions are not considered by the adaptation process.

State machine Another approach widely adopted in the findings is the use of a state machine [MCG+15; FGL+11; DBZ14; ZGC09; GPS+13; TGE+10]. Transitions represent actions. Effects of action are also known and modelled. However, no information about running actions is represented and thus considered.

Dynamic software product-lines Dynamic software product-lines approaches see the adaptation mechanism as a runtime selection between different software product-lines options. However, the two solutions that use this approach in our finding [GS10; CCH+13] do not use information about running actions and their future effects to take decisions.

Requirement-driven adaptation In our review, Baresi *et al.,* defines a reasoning approach over a goal model [BPS10]. However, running actions are excluded from the adaptation process.

Extension of MAPE-k loop Maurer *et al.,* [MBE+11] extended the traditional MAPE-k loop. Authors added a step before the monitoring one called *adaptation*. In the context of cloud infrastructure, this step allows preparing the entity before deployment, such as contract establishment for service level agreement. But, the MAPE-k loop is not modified to consider running action and their effects.

Sum up In the literature, we can find different approaches to reason over an evolving context or behaviour, as depicted in Table 4.3. However, none of them provides a solution to reason over running execution of their future effects. To make decisions, current solutions only consider current, past, or future context or behaviour.

4.2.4 Modelling and reasoning over long-term actions

In the previous sections, we detailed our review concerning modelling techniques for adaptive systems. First, we saw that just a couple of approaches that model the context, the structure, or the behaviour include a time dimension. Then, despite this element, current approaches do not include actions with their circumstances and their effects over time. Finally, we saw that none of the solutions proposes an approach to reason over running actions. To answer RQ1, our review shows that no solution in the state-of-the-art represents and reason over long-term actions and their executions.

4.3 Results RQ2: data uncertainty

In this section, we detail our findings regarding the modelling of uncertainty. First, we categorise the different kinds of uncertainty addressed in the literature. Second, we explain how state-of-the-art solutions model data uncertainty. Third, we show current approaches propagate uncertainty. Finally, we conclude by answering the second research questions of our review.

4.3.1 Categories of data uncertainty

The literature provides different approaches to tackle challenges that come with different aspects of uncertainty. In Table 4.4, we give an overview of the different categories of these approaches found in our review.

Data uncertainty The main category of uncertainty addressed by the different research community if the uncertainty of data [BBM+18; BDI+17; BGG+13; BMB+18; BMM14; CGH+17; MWV16; SWF16; VMO16; ZAY+19; Hal06; JWB+15; ZSA+16; BGP92; BSH+06; BAV+12; CNR13; JK12; PPT08; Pfe01; RP02; SCG13; SMH11; Thr00; LTB+00; Plu+03]. In these studies, they tackle challenges due to the uncertainty that can be attached to a value, in our word a data.

Design uncertainty The modelling community studied the uncertainty in the design of a model-based solution [FSC12; FC19; EPR15; EPR14; SCH+13; ZSA+16]. An example used by Famelis *et al.,* [FSC12], is the uncertainty in modelling a state machine. One may not know with the most thorough confidence the transitions to

Category	Reference
Data uncertainty	[BBM+18; BDI+17; BGG+13; BMB+18; BMM14; CGH+17; MWV16; SWF16; VMO16; ZAY+19; Hal06; JWB+15; ZSA+16; BGP92; BSH+06; BAV+12; CNR13; JK12; PPT08; Pfe01; RP02; SCG13; SMH11; Thr00; LTB+00; Plu+03]
Design uncertainty	[FSC12; FC19; EPR15; EPR14; SCH+13; ZSA+16]
Requirement uncertainty	[WSB+10; WSB+09; SCH+13]
Uncertainty in model transformations	[BBM+18; EPR15; EPR14]
Time uncertainty	[Gar08]
Uncertainty in business process	[JWB+15]
Environment uncertainty	[EM10; ZSA+16]
Behaviour uncertainty	[ZAY+19]
Hardware uncertainty	[CMR13]

Table 4.4: Categories of uncertainty addressed by the literature (RQ2.1)

lts:duc:rq2.1⟩

add.

Requirement uncertainty Related to design uncertainty, there is the requirement uncertainty. This uncertainty is due to the lack of confidence when stakeholders model the requirements of a system. In our finding, three studies have addressed challenges related to this kind of uncertainty [WSB+10; WSB+09; SCH+13].

Model transformation Due to design uncertainty, researchers advocate the use of partial models [FSC12]. As explained in Section 2.2, model transformation is a cornerstone feature in the MDE community. But, this process also contains uncertainty, tackled by three studies from our review [BBM+18; EPR15; EPR14].

Time uncertainty When occurring with time-related phenomena, uncertainty about when they are expected to occur exists. Garousi *et al.,* studied the time uncertainty of events in a distributed system.

Uncertain process Staying in the modelling approaches, we found one study that deals with the uncertainty in business processes [JWB+15]. In this paper,

JimÃľnez-RamÃŋrez *et al.,* studied the uncertainty in the properties of business processes.

Uncertainty in environment Systems tend to be more and more complex and evolve in an uncertain environment. To face challenges due to this uncertainty, researchers defined a new category of systems refers to as adaptive system. Here, we can add all studies found that answer RQ1 (*cf.* Section 4.2). For the sake of conciseness, here we will just add two studies [EM10; ZSA+16]

Uncertainty in behaviour The uncertainty in the environment can cause uncertainty in the behaviour of a system. It thus may complexify the testing phase of the system. In [ZAY+19], the authors tackle a challenge for the testing phase of a system with uncertain behaviour.

Uncertain hardware Software relies on faulty hardware, which can create errors in the computation and damage them. In our review, we found one study that faces this uncertainty in hardware [CMR13].

Sum up During our review, we found nine different categories of uncertainty studied in the literature shown in Table 4.4. These categories cover different phase of a software lifecycle, from requirement specification to the execution environment. In this thesis, we focus on data uncertainty: the lack of confidence in the data manipulated by a software.

4.3.2 Modelling data uncertainty

In this section, we detail the different techniques present in the literature that an engineer can use to model data uncertainty. We give an overview of the different approaches in Table 4.5.

Data type with a field for uncertainty In [BBM+18; BMB+18; MWV16; VMO16], authors use a complex type, named *UReal*, that contain two fields: one to represent the value and another one to represent the *standard uncertainty*. For example, when manipulating a dimension value, one may say that the value is 19.1cm ± 0.1. With the data type, an instance will have 19.1 as value and 0.1 as *standard uncertainty*. Then, based on these values, they can define a normal distribution where

Approach	Reference	Used for data uncertainty
Data type with a field for uncertainty	[BBM+18; BMB+18; MWV16; VMO16; BGP92; SMH11]	✓
Probabilistic programming	[BDI+17; BMM14; BGG+13; CGH+17; SWF16; BAV+12; CNR13; JK12; PPT08; Pfe01; RP02; SCG13; Thr00; LTB+00; Plu+03]	✓
Multiple possibilities	[FSC12; FC19; EPR15; EPR14; SCH+13; BSH+06]	(✓)
Randomness	[Gar08]	✗
Domain specific language	[WSB+10; WSB+09; JWB+15; CMR13]	✓
Model-level uncertainty	[ZAY+19]	✓
Formal model	[Hal06; ZSA+16]	✓

Table 4.5: Approaches to model data uncertainty (RQ2.2)

lts:duc:rq2.2⟩

the mean equals the value (here 19.1) and the variance the *standard uncertainty* (here 0.1). BarbarÃą *et al.,* [BGP92] used a similar approach in their database model. In their model, a probability value (a value between 0 and 1) is attached to a database value. Finally, Schwarz *et al.,* [SMH11] attached a confidence value to variables in a state machine. However, these solutions limit the representation of uncertainty to one distribution, *e.g.,* a Gaussian distribution. Although all the complexity of probability theory is hidden from the developer, it hinders its ability to choose another probability distribution that could better fit. Depending on the domain, the optimal probability distribution to represent the uncertainty of data varies. For example, the Gaussian distribution suits for metrology data [Met08] whereas Rayleigh distribution fits with GPS location data [Bor13].

Probabilistic programming A research community has investigated how to introduce probability distributions in a programming language. The different approaches are regrouped under the term *probabilistic programming* [GHN+14]. This strategy

remains the main approach used in our review [BDI+17; BMM14; BGG+13; CGH+17; SWF16; BAV+12; CNR13; JK12; PPT08; Pfe01; RP02; SCG13; Thr00; LTB+00; Plu+03]. Using this approach, probability distributions can be manipulated as a variable in a programming language.

One example is the *Uncertain<T>* language developed by Bornholt *et al.,* [BMM14]. To manipulate uncertain data, they defined an interface, *Uncertain<T>*, which can be specialised by any probability distribution. For example, an uncertain double is defined as followed: *Uncertain<double> un = new Gaussian(4,1).* Therefore, the language hides the distribution for developers. However, as in ours, they can still define and use different distributions. We strongly think that using dynamic typing and always hiding the real type, developers may not know or understand what they manipulate. It may end with runtime errors that do not provide any clear explanation. By forcing static types, we can help developers to manipulate uncertain data, but we lose in terms of flexibility [MD04].

As stated by Gordon *et al.,* [GHN+14], "the purpose of a probabilistic program is to implicitly specify a probability distribution". Knowledge about the probability distribution is still required to understand and manipulate a code done by one of these languages. We think that work can be done to abstract the concepts at a higher level. Doing so, engineers can write reasoning algorithms over uncertain data, without knowledge about probability distributions. Moreover, there is a shift in how to apprehend the problem of uncertain data. Using a probabilistic program, engineers will try to see the probability of an event E to be in a situation S whereas in this work they are interested in knowing if the current instance of E is in S. For example, using a probabilistic program, engineers will see the overall probability that the temperature is greater than 20°C. In our problem, they are want to see what is the confidence, *i.e.,* the probability, that the current measurement is greater than 20°C.

Multiple possibilities In order to face design uncertainty, Famelis *et al.,* defined the concept of *partial models* [FSC12; FC19]. A partial model is a graph where elements can be annotated with *TRUE*, *FALSE*, or *MAYBE*. *TRUE* and *FALSE* respectively indicate that the graph element should be present or not. When the

presence of the element is uncertain, a designer can annotate it with *MAYBE*. Eramo *et al.,* apply this approach to handle uncertainty in model transformation [EPR15; EPR14]: the process generate partial models. Partial models can also be used to reflect requirement uncertainty [SCH+13]. Finally, in the database community, Benjelloun *et al.,* [BSH+06] defined an approach where data can have different possible values. This approach is only suitable for data when different possibilities can be listed.

Randomness As seen in the previous section, the time uncertainty may affect some events, which can complexify the test phase of such systems. To address this challenge, Garousi *et al.,* [Gar08] defines an approach where the time of occurrence of events happen with a random parameter. However, this approach can only be used in the testing phase to represent the lack of confidence in a value. Indeed, when one has to reason over received data, such as measurement data, she cannot randomly select the value.

Domain-specific language In our review, we found four studies that define a specific language to handle uncertainty in their domain. First, for uncertainty in requirements, Whittle *et al.,* [WSB+10; WSB+09] designed the RELAX language. The language introduces fuzzy words to reflect uncertainty. For example, a requirement could be: "The workload *SHALL NOT* be greater than *THRESHOLD*". In [JWB+15], the authors present a declarative language for business processes that allow stakeholders adding probability information to properties. For example, to handle uncertain hardware, authors of [CMR13] have implemented a language that can specify reliability constraints on the execution of a function. If a language engineer considers that uncertainty is a first-class citizen concern, as done by these approaches, then she should integrate techniques to handle it in the language.

Model-level uncertainty Zhang *et al.,* [ZAY+19] uses a UML profile to define different kind of uncertainties, called U-Model [ZSA+16]. Following this approach, a designer can define a model to test CPS by generating test cases. One may use this technique to model data uncertainty.

Formal model In our review, we found two formal models for uncertainty [Hal06; ZSA+16]. First, Hall *et al.,* defined a model that implements the recommendation of

Approach	Reference	Imperceptible propagation?
Attached to language operators	[BBM+18; BDI+17; BGG+13; BMB+18; CGH+17; MWV16; SWF16; VMO16; BAV+12; CNR13; JK12; PPT08; Pfe01; RP02; SCG13; Thr00; LTB+00; Plu+03]	✓
Through state machine	[SMH11]	✓
Manual	[BBM+18]	✗

Table 4.6: Approaches to propagate data uncertainty (RQ2.3)

:results:duc:rq2.3.1⟩

Approach	Reference
Access to the confidence parameter	[BBM+18; BMB+18; MWV16; VMO16; BGP92; SMH11]
Access to probability features	[BDI+17; BMM14; BGG+13; CGH+17; SWF16; BAV+12; CNR13; JK12; PPT08; Pfe01; RP02; SCG13; Thr00; LTB+00; Plu+03]

Table 4.7: Approaches to reason over the uncertainty of data (RQ2.3)

:results:duc:rq2.3.2⟩

the GUM [Hal06]. Second, Zhang *et al.,* presented a conceptual model of uncertainty which regroups different kinds of uncertainty in a CPS [ZSA+16]. An engineer can implement one of these formal models to represent data uncertainty.

Sum up As depicted in Table 4.5, in our review, we find seven categories of approach that model uncertainty, with the most widely present: probabilistic programming. As shown in the table, most of these approaches can be used to implement data uncertainty. In this thesis, we mainly focus on using a similar approach as probabilistic programming or defining new data types that contain a field for uncertainty to help developers modelling uncertain data.

4.3.3 Propagation and reasoning over uncertainty

In this section, we study state-of-the-art approaches to propagate uncertainty when uncertain data are manipulated. Plus, we study the different approaches to

reason over uncertainty. Table 4.6 and Table 4.7 present a summary of our findings.

***[Propagation]* Attached to language operators** According to our finding, the most common approach to propagate uncertainty is to attache the propagation mechanism to the language operator [BBM+18; BDI+17; BGG+13; BMB+18; CGH+17; MWV16; SWF16; VMO16; BAV+12; CNR13; JK12; PPT08; Pfe01; RP02; SCG13; Thr00; LTB+00; Plu+03]. From a programming language point of view, (uncertain) data are mainly manipulated through language operators, such as arithmetic operators. In these works, researchers define strategies to map the semantics of a language operator to a process that propagates the uncertainty. For example, mapping the addition operator to the addition of two probability distributions.

***[Propagation]* Through state machine** One study propagate the uncertainty using a state machine [SMH11]. In this work, uncertain events are handled by *interactors*. Due to the uncertainty, multiple interactors can match the event. They are called, and their results are weighted according to the uncertainty of the event. The state machine can thus be in different states, with different confidence. The most probable one is selected.

***[Propagation]* Manual propagation** Finally, one solution requires manual propagation [BBM+18]. This work is used in model transformations. When a designer implements a transformation rule, she also has to implement the code manually to propagate uncertainty.

***[Reasoning]* Access to the confidence parameter** Six approaches in our findings allow accessing to the confidence parameter. For example, using [VMO16] developers can access to the *standard uncertainty*.

***[Reasoning]* Access to properties of probability distribution** Probabilistic programming allow the manipulation of probability distributions as programming language variables. Using such approaches, developers can access different properties of a probability distribution [BDI+17; BMM14; BGG+13; CGH+17; SWF16; BAV+12; CNR13; JK12; PPT08; Pfe01; RP02; SCG13; Thr00; LTB+00; Plu+03]. For example, they can access to the mean or the variance of a Gaussian distribution. Of, they can

compute a sample from the distribution.

Sum up The widely used approach to imperceptibly propagate uncertainty is to attach the propagation uncertainty to the language operators as we can see in Table 4.6. The main advantage of this approach is to keep a language with a syntax as close as possible to what developers are used to. However, current solutions provide, what we call, *low-level* techniques to reason over uncertainty (*cf.* Table 4.7). Indeed, state-of-the-art solutions allow developers to access either the values stored or the properties of a probability distribution.

4.3.4 Modelling of data uncertainty and its manipulation

In this review, we have seen that different kinds of uncertainty have been addressed by the literature (RQ2.1). Among them, in this thesis, we focus on data uncertainty. Different strategies can be used to model the uncertainty, with the most used one, which consists of using a probabilistic program (RQ2.2). This approach offers the propagation of uncertainty by mapping this process to language operators (RQ2.3). Plus, the properties of probability distributions can also be accessed using such techniques (RQ2.3). However, we think that research efforts now can be done to provide a language with a higher level of abstraction, as initiated by Vallecillo *et al.,* [VMO16]. One goal is to help developers implementing reasoning algorithms over uncertain data with a high-level understanding of the probability theory. Moreover, we think that specific operators should be specified to reason over this uncertainty. Another limitation of these works is that they studied uncertainties on numerical values. There are still open research questions to employ these techniques in an object-oriented language, which also contains references, nested objects or hierarchical relations.

4.4 Threat to validity

In this section, we discussed several threats to validity that may damage the results of our review.

To perform this review, we use a strategy inspired by the snowballing methodology [Woh14]. However, due to a lack of resources, we do not adequately perform it.

Contrary to what we should have done, we did not apply the backward and forward propagation on all the selected papers. Additionally, as mentioned by the authors, the quality of the results of a review that follow this methodology highly depends on the starting set, which can suffer from bias.

Moreover, the results of a review are impacted by the accuracy of the data extraction step and the research questions formulated. In this case, no discussion has been done around the formulation of the research questions. And the data extraction has been performed by a unique person, which will increase the inaccuracy.

4.5 Conclusion

In this chapter, we review the state-of-the-art approaches to answer two research questions. First, we look for studies that model adaptive systems, their contexts or behaviours to see if they also consider long-term action (RQ1). Our review shows that none of the current approaches model or enable reasoning over actions with delayed effects. Thus, some research efforts are still required to specify solutions that allow designers to add long-term action in their model and to implement techniques to reason over them. In this thesis, we start studying this problem, and we present a model-based solution, detailed in Chapter 6.

Then, we search for solutions that model data uncertainty. Different solutions have been found in our review, the main one being referred to as probabilistic programming. This solution allows developers to manipulate probability distributions as common variables of a programming language. However, as seen in our review, not all kinds of uncertainty can be represented by this approach. For example, some researchers represent the uncertainty of a value with a set of different possibilities. Therefore, open challenges still need research efforts to handle data uncertainty. Towards solving these challenges, we start by defining a language that integrates data uncertainty as a first-class citizen (*cf.* Chapter 5).

Part II

Towards a modelling frameworks for adaptive systems

5

AinâĂŹtea: managing data uncertainty at the language level

⟨chapt:aintea⟩

Contents

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.

5.1 Introduction

The increasing availability of big data and machine learning techniques brings out the importance of data in modern software systems. Data become a cornerstone piece to autonomously derive decisions from them, or at least to support decision-making processes. This trend has also been recognized by official institutions, like Datalandscape[1], a European institution analyzing the so-called *data market*. A recent example is Industry 4.0 [LFK+14], where factories are increasingly automated by analyzing data coming from different sensors and controlling processes based on this.

However, **data are inherently uncertain**. Data collected from sensors contain the uncertainty due to their accuracy and failure rate [BMM14; Met08] or malicious false data injection [LWZ+17]. Uncertainty can also be the result of human errors when entering data in a system. Also, simulation or forecast techniques carry uncertainty in their results. The same counts for all machine learning algorithms.

A software system **not considering this uncertainty would lead to a potentially catastrophic situation**, as a misperception of the reality of the system may lead to wrong decisions [BMM14]. Uncertainty blurs the human's or algorithm's understanding of a situation or system state. James Bornholt shows in [Bor13] that computing the average speed during a walking activity and ignoring the uncertainty of GPS data lead to unrealistic situations such a walking speed equals to 95 km/h. Another example in the smart grid domain is the load approximation. Ignoring the uncertainty of the grid topology may lead to a false detection of an overloading incident.

On the one hand, developers ignore this uncertainty. As shown by James Bornholt [Bor13], over 100 popular applications which use GPS location, only one considers the uncertainty of data manipulated. On the other hand, scientific computing software devotes a significant portion of their code to the management of uncertainty. This claim is supported by the existence of frameworks such as the GUM [Met08], made by the Joint Committee for Guides in Metrology[2], or OpenTurns [BDI+17], a framework supported by companies expert or using scientific computing.

[1] `http://datalandscape.eu`

[2] `https://www.bipm.org/en/committees/jc/jcgm/`

Developing such software systems requires significant expertise in handling uncertainty (probability and statistics theory) to efficiently manipulate and interpret data. Several techniques based on probability theory and fuzzy logic have been proposed [Zad; Met08; Sha76] to handle uncertainty. The application of these techniques requires a deep understanding of the underlying theories and is a time-consuming task [VMO16]. Moreover, it is hard to test and, perhaps most importantly, very error-prone.

Different works implement the theory behind uncertainty management in the form of libraries or frameworks [MWG+18; BDI+17]. In this chapter, we argue that using a library or a framework to handle uncertainty is efficient in terms of reusability but it fails to prevent the most common errors during development time. For example, all errors and bugs related to type inconsistencies and mismatches (semantics error) will be discovered only at runtime. As done by Vallecillo *et al.,* in [VMO16], we argue that **an efficient solution to handle data uncertainty in software systems should be realized at the language level itself with a first-class citizen**. Having it on the language level enhances library and framework-based approaches by adding a static, uncertainty-aware type system which is able to detect type errors at development time.

In this chapter, we describe the following contributions:

- Description and definition of the main concepts and operators to introduce in a language with uncertainty as a first-class citizen. Doing so, developers will be able to manipulate uncertain and certain data in a similar way, hiding complex concepts behind ones they are used to.
- An uncertainty-aware static type checker enabling the design-time detection of programming errors. This type checker can be used as *implicit documentation*, which helps developers in implementing uncertainty-aware algorithms.
- We implement these two contributions in the Ain'tea language, publicly available[3]. An overview of this language is depicted in Figure 1.4. Then, we use a real-world case study, built with our partner Creos S.A., to answer the two following research questions:
  - *RQ1:* Does the uncertainty management have an impact on the conciseness of a language?

[3] `https://github.com/lmouline/aintea/`

– *RQ2:* Can the type system detect errors related to uncertainty management?

Our use-case implementation shows first that our approach does not impact the conciseness of the language (RQ1). Second, it highlights the feasibility and the advantages (RQ2) of an uncertainty-aware type checking system on the language level.

The remainder of this chapter is as follows. In Section 5.2, we present and discuss the main concepts and operators for uncertainty management as a first-class citizen of a language. Section 5.3 presents our Ain'tea implementation and validates the design time error detection and its impact on language conciseness. Section 5.4 concludes and presents some open research perspectives.

5.2 Uncertainty as a first-class language citizen

⟨sec:aintea:duc⟩

5.2.1 Language overview

Data are uncertain when some data points are not precisely known. This lack of confidence is generally represented by a probability distribution, like the Gaussian distribution [Met08]. Let D be a set of data points and P_D a set of probability distributions. **An uncertain data point $u_d \in U_D$ is therefore defined by a pair (d, p_d), where $d \in D$ and $p_d \in P_D$. This representation permits to store a value, which has been observed, computed, given, or measured with a probability distribution that represents the certainty.**

Thus, we associate one data point to the value of a variable in a programming language. Adding uncertainty as a first-class language citizen thus implies to enable the creation and the manipulation of such pairs. We propose to add new data types to represent such pairs and define operators on top of these data types to manipulate them. Uncertainty propagation is done through these operators. This impacts the different elements of our language [HR04; Deg16]: the syntax (abstract and concrete) and the semantics (static and dynamic). This is summarised in Figure 5.1.

In the abstract syntax, we define the probability distributions and uncertain data types as primitive types. Plus, it contains a definition of the operators to

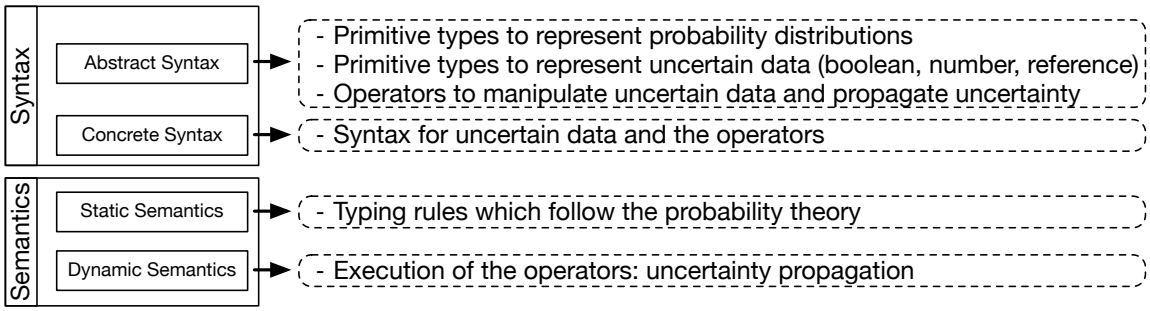


Figure 5.1: Impact of having uncertainty as a first-class language citizen on a language

ntrib-summary⟩

manipulate them. In this article, we do not propose a concrete syntax as part of the contribution, we only give an example through our implementation, Ain'tea. In the static semantics, the type system implements typing rules that follow the probability theory. Moreover, meaningful error messages for end users are provided. Finally, dynamic semantics define how the different operators are executed and thus how uncertainty is propagated. We define the dynamic semantics following the operational approach [Mos01]. It also defines the execution of the creation or deletion of uncertain data in the program.

To facilitate reading, we start by explaining the principles behind the manipulation of uncertain data based on the boolean case and deal with the numeric case (discrete and continuous) afterwards. We define the operators that can be applied on such data as well as their formal semantics.

5.2.2 Uncertain boolean

What an uncertain boolean is?

Boolean variables can take only two possible values, *TRUE* and *FALSE*. For a given data point d, the levels of confidence that d actually takes each of these values are thus proportionally tied to each other. The higher the confidence level of one value is, the lower the confidence level of the other. More precisely, if d takes the value *TRUE* with probability c then it takes the value *FALSE* with a probability $(1 - c)$. That is, $P(b = TRUE) = 1 - P(b = FALSE)$, with $P(X)$ a function that computes the probability of X.

Bernoulli distribution for uncertain boolean values

We use the Bernoulli probability distribution [Wal96] to represent the confidence level of uncertain boolean values. Bernoulli(p) denotes a random variable that equals 1 with a probability of p and 0 with a probability of $(1 - p)$. Without loss of expressiveness, we arbitrarily decide that p represents the probability that the data point takes the *TRUE* value. In other words, a *TRUE* value with a confidence level c_1 is associated with a Bernoulli($p = c_1$), while a *FALSE* value with a confidence level (c_2) is associated with a Bernoulli($p = 1 - c_2$). Thus, the uncertain booleans (*TRUE*, 0.76) and (*FALSE*, 0.24) differ in their observed value but not in the probability distribution.

More generally, in our proposed language, an uncertain boolean is thus represented as a pair (d, p_d), with d is a boolean value and p_d is a Bernoulli distribution set with the probability of the *TRUE* value. **The abstract syntax of our language contains thus the Bernoulli distribution and a definition of the pair (boolean value, Bernoulli distribution).** Existing approaches [BMB+18] store only the Bernoulli distribution and use the aforementioned equivalence to convert a *FALSE* value to a *TRUE* one. We find these approaches to be counter-intuitive as developers would only manipulate *TRUE* values, regardless of the value actually observed. As we want to keep the data manipulation as close as possible to common programming languages, we decide to represent a boolean as the composition of both the value and the Bernoulli distribution. Moreover, our approach keeps the initial value, which has been observed or measured.

The abstract syntax also contains the following operators. First, the definition of boolean (and, or, not) and equality ($=$, $\neq$) operators are extended to uncertain booleans. In addition, we define the cast operator between these two data types. Finally, one may reason over the confidence level. We thus define four novel operators: existence, confidence, dot, and uncertain equality. We detail them in the next section.

Operational semantics

We denote by $(b, B(p))$ an uncertain boolean with a boolean value b and a Bernoulli distribution $B(p)$ with probability p. In order to distinguish between the equality

operator and the mathematical equality symbol, in the rest of this article, we represent the first by = and the second by :=. We define the following uncertain booleans to exemplify the different operators: $u_{b1} := (TRUE, B(0.3))$, $u_{b2} := (TRUE, B(0.65))$, and $u_{b3} := (FALSE, B(0.45))$.

⟨op:existence⟩ **Operator 1** *The **existence operator** returns true if an uncertain data point $u_d \in U_D$ has a value with at least a given probability $t \in C$, where $C := \{c \in \mathbb{R} \mid 0 \leqslant c \leqslant 1\}$, and false otherwise: $U_D \times C \xrightarrow{exists} B$.*

Using this operator, developers know if it exists at least one value for which its confidence level is greater than a given threshold. For example, using it, one may know if a fuse is open ($isOpen := TRUE$) or closed ($isOpen := FALSE$) with a confidence of at least 80%. The operational semantics of this operator is specified by the following function:

$$exists((v, B(p)), t) := (p \geqslant t) \| (1 - p \geqslant t)$$

Applying the operator on u_{b1}, we thus get:

- $exists(u_{b1}, 0) := (0.3 \geqslant 0) \parallel (0.7 \geqslant 0) := TRUE$,
- $exists(u_{b1}, 0.5) := (0.3 \geqslant 0.5) \parallel (0.7 \geqslant 0.5) := TRUE$,
- $exists(u_{b1}, 0.9) := (0.3 \geqslant 0.9) \parallel (0.7 \geqslant 0.9) := FALSE$.

In plain English, these examples mean: considering u_{b1}, there exists a value with a confidence level of at least 0 and 0.5 but there is no value with a confidence level of at least 0.9.

op:confidence⟩ **Operator 2** *The **confidence operator** computes the most confident value D with a minimal confidence level $t \in C$ of a given uncertain data point $u \in U_D$: $U_D \times C \xrightarrow{confidence} D$.*

Taking an example from the smart grid domain, one may use this operator to get the most probable state of the fuse (open or close), if its confidence is superior to 80%.

This operator raises an error in two cases: when (i) no value exists with at least the given confidence level or (ii) the confidence level of both values, true or false, are

equal. Its semantics is thus:

$$confidence((v, B(p)), t) := \begin{cases} true, \text{ when } exists((v, B(p)), t) \wedge p > 1 - p \\ false, \text{ when } exists((v, B(p)), t) \wedge p < 1 - p \\ \bot \text{ , when } p = 0.5 \end{cases}$$

If we apply this operator on u_{b1} with the same base as the previous example, then it returns:

- $confidence(u_{b1}, 0) := FALSE$,
- $confidence(u_{b1}, 0.5) := FALSE$,
- $confidence(u_{b1}, 0.9) := ERROR$.

With a confidence level of at least 0 or 0.5, u_{b1} is more likely to be equal to *FALSE*. Otherwise, we cannot know its value with a minimal confidence level of 0.9.

⟨op:cast⟩ **Operator 3** *The **cast operator** casts an uncertain data point to a certain one and vice-versa. The cast operation from uncertain to certain is defined as follow:* $U_D \xrightarrow{cast} D$. *Formally, the opposite operation is described as follows:* $D \xrightarrow{cast} U_D$.

The cast operation from an uncertain to a certain one can be used to get the most confident value, without any constraint on the confidence (which is done by the confidence operator). This helps developers to reason or to make decisions upon the most probable situation. It can be performed by using the confidence operator with 0 as given confidence level:

$$cast(u_b) := confidence(u_b, 0)$$

The opposite operation, from certain to uncertain, is mainly a syntactic manipulation. Indeed, certain data is always considered as uncertain data with the maximum confidence level. In the case of uncertain boolean, it's 1:

$$cast(b) := \begin{cases} (b, B(1)), \text{ when } b = TRUE \\ (b, B(0)), \text{ when } b = FALSE \end{cases}$$

Performing this operator on our example results in:

- $cast(u_{b1}) := confidence(u_{b1}, 0) := FALSE$,
- $cast(u_{b2}) := confidence(u_{b2}, 0) := TRUE$,
- $cast(TRUE) := (TRUE, B(1))$,
- $cast(FALSE) := (FALSE, B(0))$,

⟨op:dot⟩ **Operator 4** *The **dot operator** allows accessing both elements of uncertain data: the value $d \in D$ or the probability distribution $p_d \in P_D$: $U_D \xrightarrow{dot} D \cup P_D$.*

This operator allows accessing the elements that compose uncertain data. For the value, it permits to resolve what is the value measured, observed, computed or given, without uncertainty consideration using the *value* keyword:

$$(v, B(p)).value := v$$

For the confidence, it gives access to the confidence in order to reason over the probability distribution using the *confidence* keyword:

$$(v, B(p)).confidence := B(p)$$

Used on u_{b1} and u_{b3}, this operator will return:

- $u_{b1}.value := TRUE$,
- $u_{b1}.confidence := B(0.3)$,
- $u_{b3}.value := FALSE$,
- $u_{b3}.confidence := B(0.45)$,

op:u-equality⟩ **Operator 5** *When applied between two uncertain data, **uncertain equality operators** ($=, \neq$) compute the probability that both uncertain values are equal or not: $U_D^2 \xrightarrow{u-equality} U_B$, where U_B is the set of uncertain booleans. When used between an uncertain and a certain value, they return the probability that the uncertain variable is equal to the certain value: $U_D \times D \xrightarrow{u-equality} U_B$.*

In both cases, the resulting value of the uncertain boolean is computed by applying the operator to the boolean values. When used between two uncertain booleans, the

calculated probability corresponds to the probability that both values are equal to *TRUE* or both equal to *FALSE*. To do so, we compute the union of the intersection of the different probabilities. It results therefore in the following semantics:

$$(v, B(p)) = b := \begin{cases} u_b[v = b, B(p)], when\ b = TRUE \\ u_b[v = b, B(1-p)], when\ b = FALSE \end{cases}$$

$$(v_1, B(p_1)) \neq b := \begin{cases} u_b[v \neq b, B(1-p)], when\ b = TRUE \\ u_b[v \neq b, B(p)], when\ b = FALSE \end{cases}$$

$$(v_1, B(p_1)) = (v_2, B(p_2)) := (v_1 = v_2, [B(p_1) \cap B(p_2)] \cup [B(1-p_1) \cap B(1-p_2)])$$

$$(v_1, B(p_1)) \neq (v_2, B(p_2)) := (v_1 \neq v_2, [B(p_1) \cap B(1-p_2)] \cup [B(1-p_1) \cap B(p_2)])$$

To compute the intersection of the union of Bernoulli distributions, first, we should evaluate whether they are disjoint and dependent. We call two variables dependent if they are, directly or indirectly, defined based on at least one common variable (uncertain or not). For example, let *temp* be a temperature, the boolean $b_1 = t \leqslant 0$ and $b_2 = t > 18$ are dependent because they share the same variable t. As both values are directly defined using t, they are dependent. We call two variables disjoint when they do not share the same a common set of possible values. In our example, b_1 and b_2 are disjoint because they don't share any possible values, *i.e.,* $]-\infty, 0] \cap]18, +\infty[= \emptyset$.

Below we illustrate the different formulas to compute the union or intersection of Bernoulli distributions. Overall, there are three cases: (i) disjoint variables, regardless if they are independent or not, (ii) independent and non-disjoint variables, and (iii) dependent and non-disjoint variables. To the best of our knowledge, there is no formula in the latter case. An exception is raised in such cases.

$$B(p_1) \cap B(p_2) = 0 \qquad \text{(Disjoint var.) \{?\}}$$
$$B(p_1) \cup B(p_2) = B(p_1 + p_2)$$
$$B(p_1) \cap B(p_2) = B(p_1 * p_2) \qquad \text{(Indep. and non-disjoint var.) \{?\}}$$
$$B(p_1) \cup B(p_2) = B((p_1 + p_2) - (p_1 * p_2))$$
$$B(p_1) \cap B(p_2) = \bot \qquad \text{(Dep. and non-disjoint var.) \{?\}}$$
$$B(p_1) \cup B(p_2) = \bot$$

Let us consider u_{b1} and u_{b3} two independent and non-disjoint variables. Applying the equality operators on them will result in:

- $u_{b1} = u_{b3} := (TRUE = FALSE, B(0.3 * 0.45) \cup B(0.7 * 0.55))$
  $:= (FALSE, B(0.135 + 0.385)) := (FALSE, B(0.52))$
- $u_{b1} \neq u_{b3} := (TRUE \neq FALSE, B(0.3 * 0.55) \cup B(0.7 * 0.45))$
  $:= (TRUE, B(0.165 + 0.315)) := (FALSE, B(0.48))$

In plain English, u_{b1} and u_{b3} have a probability of 52% of having the same value, *i.e.,* 48% of having different values.

op:s-equality⟩ **Operator 6** ***Identity operators*** *(==, ≠≠) return true if two uncertain data are identical: same value and same probability distribution. Formally,* $U_D^2 \xrightarrow{identity} B$

This operator is an extension of the equal operator available in programming languages or the equals method in Java. It compares the probability parameter of the Bernoulli distribution and the values of the uncertain booleans:

$$(v_1, B(p_1)) == (v_2, B(p_2)) := (p_1 = p_2) \wedge (v_1 = v_2)$$
$$(v_1, B(p_1)) \neq\neq (v_2, B(p_2)) := p_1 \neq p_2 \vee (v_1 \neq v_2)$$

For example, the comparison of u_{b1} and u_{b2} returns:

- u_{b1}==u_{b2}:= (0.65=0.3) $\wedge$ ($TRUE$=$TRUE$) := $FALSE$,
- u_{b1}≠≠u_{b2} := (0.65≠0.3) $\vee$ ($TRUE$≠$TRUE$) := $TRUE$.

Operator 7 ***Uncertain boolean operators*** *($\wedge$, $\vee$, $\neg$) compute the result of the boolean operation and its confidence:* $U_B^2 \xrightarrow{u-booleans} U_B$

The value of the resulting uncertain boolean is computed by applying the boolean algebra on the values of both input uncertain booleans. The second part of the computation combines (union and intersection) their probability distributions:

$$\begin{aligned}(v_1,\ B(p_1))\ \wedge\ (v_2,\ B(p_2)) &:= (v_1\ \wedge\ v_2,\ B(p_1) \cap B(p_2)) \\ (v_1,\ B(p_1))\ \vee\ (v_2,\ B(p_2)) &:= (v_1\ \vee\ v_2,\ B(p_1) \cup B(p_2)) \\ \neg(v_1,\ B(p_1)) &:= (\neg v_1,\ B(1-p_1))\end{aligned}$$

If we assume u_{b1} and u_{b3} independent and non-disjoint variables, applying these operators return:

- u_{b1} && u_{b3} := (*TRUE* && *FALSE*, B(0.65*0.45)) := (*FALSE*, B(0.2925)),
- $u_{b1} \parallel u_{b3}$:= (*TRUE* $\parallel$ *TRUE*, B(0.65 + 0.45 - 0.65*0.45)) := (*TRUE*, B(0.8075)),
- $\neg u_{b1}$:= $\neg$($\neg$*TRUE*, B(1 - 0.65)) := (*FALSE*, B(0.35)).

5.2.3 Uncertain number

What is an uncertain number?

Numerical variables can theoretically take an infinite number of values. They can be either on a continuous domain, simulated by floating-point values, or on a discrete one, simulated by integer values. The confidence level of such data is either **precisely linked to the value** (observed, measured, ...) or **is distributed over the range of possible values**. For example, values get from sensors are always given with a certain accuracy, *e.g.,* 18.4°C $\pm$ 0.1°C. The confidence level of the measured values is thus distributed over a range. Values set by humans are also uncertain due to possible human errors. However, there might be no information concerning the distribution of this uncertainty. The confidence level is thus attached to the precise value.

Representation of uncertain numbers

Like uncertain booleans, uncertain numbers are composed of two elements: a numerical value and a probability distribution that represents its uncertainty. According

Type	Data type	Gaussian	Rayleigh	binomial	Dirac
Continuous	float, double	✓	✓		✓
Discret	byte, short, integer, long			✓	✓

Table 5.1: Which distribution can be used to represent the uncertainty of which data type

g-types-proba⟩

to the nature of the numerical value, developers have to decide which distribution fits the uncertainty of their variable. In probability theory, one distinguishes between continuous distributions and discrete distributions. The former defines the distribution of the probability density over a continuous domain. The latter instead apply to a discrete domain. In our proposed language, we support the following distributions: two continuous probability distributions, Gaussian [Wal96] and Rayleigh [Wal96], and one discrete distribution, the binomial distribution [Wal96]. In addition, we support the Dirac delta function [GS64], which can be used on a continuous or a discrete domain. We refer to Chapter 2 for a detailed description of these distributions.

In our language, we add new data types that represent each distribution. In addition, we add a new type for all possible combinations, described in Table 5.1. Existence, confidence, dot, cast and uncertain equality operators are also defined for these data types. We add operators specific to numerical values: arithmetic and comparison operators. The semantics of these operators are formalised in the next section.

Operational semantics

We denote by (v, p_d) an uncertain number with a numerical value v and a probability distribution $p_d \in P_D$. The following uncertain numbers are defined to be used as examples for the description of the operators:

- $u_{n1} := (7, \mathit{binomial}(20, 0.37))$,
- $u_{n2} := (10, \mathit{binomial}(30, 0.33))$,
- $u_{n3} := (12, \mathit{Gaussian}(12, 9))$,
- $u_{n4} := (23, \mathit{Gaussian}(23, 5))$.

Operator 1 - Existence operator.

In probability distributions, the value with the highest probability is called the

mode [MGB63]. We denote it by m. For discrete uncertain numbers, the operator, defined in Operator 1, returns *TRUE* if the probability of the mode value is greater or equal to a given base:

$$exists((v,p),t) := P(x=m) \geqslant t, \text{ where } P(x=m) \text{ is the probability of } m$$

Based on u_{n1}, we thus get:

- $exists(u_{n1},\, 0) := P(x{=}7) \geqslant 0 := 0.1542985 > 0 := TRUE$,
- $exists(u_{n1},\, 0.5) := P(x{=}7) \geqslant 0.5 := 0.1542985 > 0.5 := FALSE$,
- $exists(u_{n1},\, 0.9) := P(x{=}7) \geqslant 0.9 := 0.1542985 > 0.9 := FALSE$.

This operator allows verifying that there is no value with a confidence level of at least 50 or 90% for u_{n1}, but there is at least one value with a non-null confidence level.

For uncertain continuous numbers, no semantic exists for a continuous domain since the probability of the mode cannot be computed for continuous probability distributions, $\int_m^m P(u_n)\,\mathrm{d}x = 0$:

$$exists((v,p),t) :=\perp$$

Operator 2 - Confidence operator.

The semantics of the confidence operator relies on the existence operator. For uncertain discrete numbers, it returns the mode if the existence operator returns true:

$$confidence((v,p),t) := \begin{cases} m, \text{ if } exists((v,p),t) \\ \perp \text{ , otherwise} \end{cases}$$

Applying this operator on the u_{n1} using the same base as in the previous example, the operator returns the mode value, 7, only for the first case:

- $confidence(u_{n1}, 0) := 7$,
- $confidence(u_{n1}, 0.5) :=\perp$,
- $confidence(u_{n1}, 0.9) :=\perp$.

This means that the most probable value which has at least zero as confidence level is 7 for u_{n1}.

For continuous numbers, as this operator is based on the existence operator, which is not defined, it is also not defined:

$$confidence((v, p), t) := \perp$$

Operator 3 - Cast operator. In addition to the cast operator between uncertain and certain data, this operator is extended to support casting operations between two uncertain numbers, when an approximation function between two distributions exists. In Table 5.2, we summarize the permitted casts in our language.

When casting an uncertain number (discrete or continuous) to a certain one, we return the value with the highest confidence, *i.e.,* the mode m.

$$cast((v, p) := m$$

For example, casting u_{n1} and u_{n3} to certain numbers will result in:

- $cast(u_{n1}) := 7$,
- $cast(u_{n3}) := 12$.

When casting a certain to an uncertain number, we should be able to specify a distribution that represents a confidence level of 100% on a precise value. This is not always possible with all distributions. From the distributions implemented in our language, it is only possible for the binomial distribution and the Dirac delta function. For the other distributions, it is either impossible, like for the Poisson distribution (not implemented here) or it requires domain-specific heuristics. For example, the Gaussian distribution can be initialized with a variance value that is as close as possible to zero. But the definition of "as close as possible" differs according to the domain. For instance, when handling temperature values, 0.01 could be considered sufficiently close to 0. Nonetheless, this accuracy is not sufficient for other fields such as meteorology. As no global value could be selected, we decided to forbid such casting operations when it is not possible without heuristics. We, therefore, define the following semantics:

To \ From	Gaussian	Rayleigh	binomial	Dirac	certain number

From: source type of uncertain number; To: targetted type. For readability purpose, we put the distribution names to represent the different types of uncertain numbers

Table 5.2: Cast operations allowed in our language

able:allowed-cast-op⟩

$$cast(v) := \begin{cases} (v, Dirac(coeff = 1, shift = v), \text{ when the expected distribution} \\ \quad \text{is a Dirac delta function} \\ (v, binomial(n = v, p = 1)), \text{ when the expected distribution} \\ \quad \text{is a binomial distribution delta function and } v \in \mathbb{N} \\ \perp \text{ , otherwise} \end{cases}$$

For example, casting 30 and 6.7 would result in:

- $cast(30) := (v, Dirac(1, 30))$
- $cast(30) := (v, binomial(30, 1))$
- $cast(6.7) := (v, Dirac(1, 6.7))$

Finally, the operator can be applied between two uncertain numbers if and only if a mapping or an approximation between the source and target probability distributions exists in probability theory:

$$cast(v, p_1) := \begin{cases} (v, approximation(p_1)), \text{ if the approximation exists} \\ \perp \text{ , otherwise} \end{cases}$$

Following the casting rules described in Table 5.2, u_{n1} can be cast into an uncertain number with a Gaussian distribution and u_{n3} can be cast into an uncertain number with a binomial distribution:

- $cast(u_{n1}) = (7, Gaussian(7.4, 4.662))$,
- $cast(u_{n3}) = (12, binomial(48, 0.25))$.

Operator 4 - Dot operator. For uncertain numbers, this operator has the same semantics as described in Operator 4:

$$(v, p_d).value := v$$

$$(v, p_d).confidence := p_d$$

When used on u_{n1} and u_{n3}, this operator returns:

- $u_{n1}.value := 7$,
- $u_{n1}.confidence := binomial(20, 0.37)$,
- $u_{n3}.value := 12$,
- $u_{n3}.confidence := Gaussian(12, 9)$.

Operator 8 ***Uncertain arithmetic operators*** *(+, −, ∗, /) compute the arithmetic operation for uncertain numbers* U_N*:* $U_N^2 \xrightarrow{u-arithmetic} U_N$

When performing arithmetic operations on uncertain variables, both values and probability distributions are considered. Two strategies to perform arithmetic operations on uncertain numbers are identified: numerical and analytical. While the second strategy is used in simple expressions, the first strategy is used when the expression is complex by returning an approximation of the arithmetic expression. Our language implements only the analytical method. Therefore, any semantics that requires the execution of a numerical method is considered undefined. If the second one is required, we consider the operator undefined. In Section 5.2.5, we detail the allowed operations in our language.

As with boolean expressions, the independence and joint of the two combined variables impact on the formula used to effectuate an arithmetic operation. Calculations are more complex when they are dependent. For example, although the addition of two independent Gaussian is done by adding the mean and the variance of the two distributions, the covariance matrix must be calculated when they are dependent. In such cases, the numerical approach is used.

Arithmetic operations are applied on both elements that define an uncertain number, the value and the probability distribution:

$$(v_1, p_1) + (v_2, p_2) = (v_1 + v_2, p_1 + p_2)$$
$$(v_1, p_1) - (v_2, p_2) = (v_1 - v_2, p_1 - p_2)$$
$$(v_1, p_1) * (v_2, p_2) = (v_1 * v_2, p_1 * p_2)$$
$$(v_1, p_1)/(v_2, p_2) = (v_1/v2_2, p_1/p_2)$$

For example, adding u_{n3} and u_{n4} will result in: $u_{n3}+u_{n4} := (12+23, Gaussian(12+23, 5+9)) := (35, Gaussian(35, 14))$.

Operator 9 ***Inequality operators*** *($<$, $\leqslant$, $>$, $\geqslant$) computes the confidence that the left side value is (strictly) less or greater than the right one :* $(U_N \times N)^2 \xrightarrow{inequality} U_B$

When the inequality operator is used between an uncertain and a certain number, we compute the confidence that the left-hand side is greater than or equal to the right-hand side:

$$(v, p_d) \prec a := (c > 0,\ B(c));\ c := P(u_n \prec a);\ \ \prec \in \{<, >, \leqslant, \geqslant\}$$
$$a, \text{ a numeric number in the same set as v}$$

For example, comparing u_{n1} and u_{n3} will return:

- $u_{n1} > 10 := (0.17 > 0\ ,\ B(0.17)) := (TRUE\ ,\ B(0.17))$,
- $u_{n3} \leqslant 10 := (0.25 > 0,\ B(0.25)) := (TRUE,\ B(0.25))$.

When the inequality operator is used between two uncertain numbers, the probability variable is substituted by $u_{n1} \prec u_{n2} \Leftrightarrow u_{n1} - u_{n2}$. From there, we can apply the operator, $\prec$, between the result and 0. An operation can be executed, if and only if the subtraction is defined between the two operands.

$$(v_1, p_1) \prec (v_2, p_2) := [(v_1, p_1) - (v_2, p_2)] \prec 0;\ \ \prec \in \{<, >, \leqslant, \geqslant\}$$

For example, comparing u_{n3} and u_{n4} will result in: $u_{n3} \geqslant u_{n4} := u_{n3} - u_{n4} \geqslant 0 := (-11, Gaussian(-11, 4)) \geqslant 0 := (FALSE, B(0.00))$.

Operator 5 - Uncertain equality operator.

When the equality operator is applied to a discrete uncertain and a certain number, it returns the probability that the uncertain number equals (or not) the certain one:

$$(v_1, p_1) \prec a := (c > 0,\ B(c));\ c := P(u_n \prec a);\quad \prec \in \{=, \neq\}$$

For example, applying these operators on u_{n1} or u_{n2} will result in:

- $u_{n1} = 2 := (0.00 > 0,\ B(0.00)) := (FALSE,\ B(0.00))$,
- $u_{n2} \neq 13 := (0.89 > 0,\ B(0.89)) := (TRUE,\ B(0.89))$.

As it is impossible to compute the confidence of a precise number, this operation cannot be performed for continuous uncertain numbers:

$$(v_1, p_1) \prec a := \perp$$

To compare two discrete uncertain numbers, we use the same strategy as for the inequality operator: we subtract both and compare it with zero.

$$(v_1, p_1) \prec (v_2, p_2) := (v_1, p_1) - (v_2, p_2) \prec 0;\quad \prec \in \{=, \neq\}$$

As this comparison is impossible with continuous uncertain numbers, this operator is undefined for such:

$$(v_1, p_1) \prec (v_2, p_2) := \perp$$

Operator 6 - Identity operator. Similarly to uncertain booleans, the identity operator (==) applied to uncertain numbers returns true only if both values and both probability distributions are equal:

$$(v_1, p_1) = (v_2, p_2) := v_1 = v_2 \&\& p_1 = p_2$$

The unequal operator ($\neq\neq$) returns true if both values or distributions are unequal:

$$(v_1, p_1) \neq (v_2, p_2) := v_1 \neq v_2 \| p_1 \neq p_2$$

5.2.4 Uncertain references

What is an uncertain reference?

References, or pointers, allow storing a directed association from one element (*e.g.,* a Java object) to another one. **A reference is defined as uncertain when the existence of this relation is not known with the most thorough confidence.** For example, to represent the topology of the smart grid we can imagine substituting cables by simple references from one entity to another. These references are not known with absolute confidence due to the uncertainty of fuses states.

Mapping to uncertain booleans

Like for uncertain booleans, uncertain references have two states: either the reference exists or not. The confidence level on the existence can thus be represented by an uncertain boolean. We map the *TRUE* value to the existence of the reference and the *FALSE* one to its non-existence. Internally, uncertain references are represented by two components: a reference value and an uncertain boolean.

$$(Reference, c) \mapsto (Reference, (v, B(p = c))),\ v \in \{TRUE, FALSE\}$$

The abstract syntax of our language is thus extended to add a new data type: uncertain reference. The semantics of the existence, confidence, cast, and dot operator are extended to consider this new type.

Operational semantics

We denote by (r, u_b) an uncertain reference with a value r and an uncertain boolean u_b, which represents its uncertainty. The following two uncertain references exemplify this:

- u_{r1}:=(r_1, u_{b1}), u_{b1}:=($TRUE$, $B(0.88)$),

- u_{r2}:=(r_2, u_{b2}), u_{b2}:=($TRUE$, $B(0.34)$).

Operator 1 - Existence operator. The semantics of this operator is defined upon the existence operator of the uncertain boolean that represents the uncertainty:

$$exists((r, u_b), t) := exists(u_b, t)$$

When applied on the two examples from above, this operator would return:

- $exists(u_{r1}, 0.8) := exists(u_{b1}, 0.8) := TRUE$,
- $exists(u_{r2}, 0.8) := exists(u_{b2}, 0.8) := FALSE$.

Operator 2 - Confidence operator. This operator relies on the confidence operator of the uncertain boolean. If this operation tells that the most confident value given a base is $TRUE$, then the existence operator returns the reference when applied on an uncertain reference:

$$confidence((r, u_b), t) := \begin{cases} r, \text{ when } confidence((u_b), t) = TRUE, \\ \perp \text{ , otherwise} \end{cases}$$

If applied to the two examples, it will return:

- $confidence(u_{r1}, 0.8) := r_1$,
- $confidence(u_{r2}, 0.8) := \perp$.

Operator 3 - Cast operator.

When casting an uncertain reference to a certain one, it will return the reference value only if its confidence level is strictly superior to zero:

$$cast(r, u_b) := confidence((r, u_b), 0)$$

Casting both of our results into certain references will thus return in:

- $cast(u_{r1}) := confidence(u_{r1}, 0) := r_1$,
- $cast(u_{r2}) := confidence(u_{r2}, 0) := r_2$.

When casting a reference to an uncertain one, the confidence level of its existence

is set at the maximum, 1:

$$cast(r) := (r, (TRUE,\ 1))$$

For example, casting a reference r_3 to an uncertain one returns: $cast(r_3) := (r_3, (TRUE, 1))$.

Operator 4 - Dot operator.

The dot operator allows accessing the different elements of the uncertain reference:

$$(r, u_b).value := r$$
$$(r, u_b).confidence := u_b$$

Applying this operator on u_{r1} thus returns:

- $u_{r1}.value(u_{r1}) := r_1$,
- $u_{r1}.confidence(u_{r1}) := u_{b1}$.

5.2.5 Static semantic: typing rules

⟨subsec:type-system⟩ This section introduces the particularity of the type-system of our language compared to those without support for uncertainty. In particular, we stress two points. First, the typing of implicit casts between uncertain and certain data. Second, the typing and interactions between different data types when applying arithmetic operations.

Implicit casts In our language, we enable implicit casts. In addition to the casting operations shown in Table 5.2, our proposed type system enables casts between uncertain and certain booleans.

Typing rules for arithmetic operations Probability theory defines how two probability distributions can be combined and what the result should be, when possible. We follow these rules to define the result of the arithmetic operations between uncertain numbers. Below, Table 5.3 depicts the results of each operation between two uncertain numbers.

	Gaussian	Rayleigh	binomial	Dirac	certain nb.
Gaussian	Gaussian	✗	Gaussian	Gaussian	Gaussian
Rayleigh	✗	Rayleigh	✗	Rayleigh	Rayleigh
binomial	Gaussian	✗	binomial	binomial	binomial
Dirac	Gaussian	Rayleigh	binomial	Dirac	Dirac
certain nb.	Gaussian	Rayleigh	binomial	Dirac	certain nb.

Table 5.3: Typing rules for arithmetic operations

rules-arth-op⟩

When an arithmetic operation is applied to uncertain values with two different probability distributions, the resulting distribution may follow a well-defined distribution or not. In case the resulting distribution is different from the distributions supported by our language, we consider it as undefined. For example, performing an arithmetic operation between a Gaussian and a Rayleigh distribution is not permitted since it does not result in a well-defined distribution and should be calculated by applying the convolution rule.

For the implemented distributions, all operations between two numbers represented by the same distribution, result in an uncertain number with the same distribution. For example, an addition between two uncertain numbers with a Gaussian distribution returns another one with a Gaussian distribution. One exception: combining two binomial distributions result in another one only if the parameter p is identical in both cases. If p is different, the resulting distribution is a Poisson distribution, not implemented in our language. Hence, we consider it as undefined.

An operation between a probability distribution and a scalar results in this probability distribution, shifted (for addition or subtraction) or with a coefficient applied (for multiplication or division). We perform these operations on the probability distribution when an uncertain number is combined with a certain one.

A Gaussian distribution can be approximated by a binomial one and vice-versa. When these two are being combined, the type of the final result can be one of these two. There are thus two possible choices: either we cast both into uncertain numbers with a Gaussian distribution or both into uncertain numbers with a binomial distribution. By default, we decide to opt for the first option. Therefore, the arithmetic operator returns an uncertain number with a Gaussian distribution.

5.3 Evaluation

ec:aintea:validation⟩ We validated our approach by implementing a language named Ain'tea, which features an uncertainty-aware static type system and provides the concepts and operators presented in Section 5.2. We evaluate the following two research questions based on our implementation of Ain'tea:

- **RQ1:** Does native uncertainty management on the language level have an impact on the conciseness of a language, *i.e.,* its ability to express concisely statements?
- **RQ2:** Can the type system detect errors related to uncertainty management?

To address them, we compared our implementation to two state-of-the-art frameworks: Infer.NET [MWG+18] and OpenTURNS [BDI+17]. The code used for the experiments of the validation is available on the repository of our implementation.

In this section, we first detail the implementation of the Ain'tea language. Then, we show that our solution can detect type errors earlier than the two solutions we compare our work to. Before discussing the results, we compare the number of lines of codes required for the different solutions to perform uncertainty propagation.

5.3.1 Ain'tea: our implementation

We use the Xtext language workbench[4] to define the concrete syntax and the type checker of our language (static semantics). Based on the concrete syntax, Xtext generates the abstract one. We implement the dynamic semantics with K3[5].

In addition, we implemented different code samples which shows the different features of our language.We also developed a simplified version of our use case example, introduced in Chapter 3. We presented it as a tutorial of our language.All code can be executed using a runner, available on the project repository.

Overview of the language

Ain'tea supports classes, with fields and functions. It has an expression language with arithmetic and boolean expressions, *IF*-conditions and affectation statements.

[4] `http://www.eclipse.org/Xtext/`
[5] `http://diverse-project.github.io/k3/`

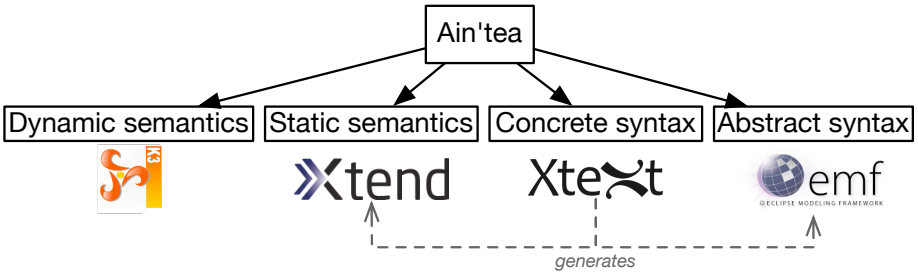


Figure 5.2: Global architecture of Ain'tea

bal-lang-arch⟩

Ain'tea is composed of four elements: an abstract syntax, a concrete syntax, static and dynamic semantics. As depicted in Figure 5.2, the concrete syntax has been implemented using Xtext. As we used a grammar-first approach, the abstract syntax has been automatically generated from the concrete one. In addition, stub-classes are generated for the semantics (static and dynamic). Using Xtend, we implement the static semantics. The dynamic semantics is defined as an operational semantics and implemented using K3 [JCB+15]. The implementation of our uncertainty-aware type checker and semantic uses two strategies. For simple operations (*e.g.* operator semantics for uncertain booleans), we implement them using K3. For more complex operations, like the computation of integral or combining different distributions, we use a third party library: Apache Commons Math Library[6].

Finally, by using the Xtext language workbench, other important features have been generated for our language. Among them, there is the error maker, auto-completion and an Eclipse plugin.

Syntax

The syntax of the language has been inspired by Java: keywords to define classes, fields and functions are similar. Regarding the uncertain operators defined in Section 5.2, we use the same syntax as for certain operators where ever possible. We give an overview of their corresponding syntax in the following list:

- confidence: [ ]
- existence: **exist**

[6] `http://commons.apache.org/proper/commons-math/`

- cast: **as**
- dot: **.**
- inequality: >, >=, <, <=
- (uncertain) equality: ==, **!=**
- identity: ===, **!==**
- uncertain arithmetic: +, **-**, *, /
- uncertain boolean: **&&**, ||, **!**

Inequality, arithmetic and boolean operators have identical syntax for uncertain and certain data. By making this choice, combining uncertain or certain data is done with the same syntax. For example, adding two uncertain numbers is performed by the following syntax: $u_{N1} + u_{N2}$.

When the equality operator (==, !=) is used with at least one uncertain data type, we apply the uncertain equality operator. For example, the following code $u_{N1} == u_{N2}$ returns the confidence that both values are equal. To use the identity operator, we use the following syntax: === and !==. For example, this code $u_{N1} === u_{N2}$ checks if both uncertain numbers are identical (cf. Operator 6).

Following his name, the syntax of the dot operator is a dot (.). The value can be accessed using the *value* keyword. For example, to get the value of an uncertain boolean, one will write $u_b.value$. To get the probability distribution, the *confidence* keyword has been introduced. For example, the Bernoulli distribution of an uncertain boolean is resolved using the following syntax: $u_b.confidence$.

For the cast operator, we add the *as* keyword. For instance, to cast an uncertain number to a certain one, one will do u_n *as* $Number$. Here, *Number* represents any number type of our language: short, integer, long, float and double.

An internal method is used to call the existence operator: *exist*. For example, in order to check if a value of an uncertain boolean exists with at least 0.8 confidence, one needs to add the following line in its code: exist(u_B, 0.8).

Finally, the confidence operator is called using square brackets: [ ]. For example, to get the most confidence value of an uncertain number with at least 0.7 as confidence, one will write: $u_N[0.7]$

The type of uncertain data can be thought of as a pair of two types: one for the raw data and one for the uncertainty representation. We decide to use a syntax similar to the one used in Java to declare generic types: Type1<Type2>. We arbitrarily choose to put the probability distribution for the first type and the type of the raw data point for the second. For example, the declaration of an uncertain boolean is done using the following syntax: Bernoulli<bool>.

Dependency and joint detection

As we explained in Section 5.2, dependency and joint between two variables impact the semantics of the different operators. In our implementation, we first consider that any variable initialized by a constant is independent. Then, we compute a dependency graph between the different variables. However, we use an algorithm which covers only the simple cases.

Concerning the joint, we also apply a simple and naive algorithm to compute the domain covered by boolean variables. But, this is only performed on boolean resulting from comparison operators with one static constant. We let for future work the implementation of an algorithm that applies the same rules at runtime. Below, an example of what our implementation can do.

Let us imagine three uncertain booleans b_1, b_2, and b_3 defined as follow, with u_n an uncertain number:

- $b_1 := u_n >= 10$
- $b_2 := u_n < 15$
- $b_3 := u_n > 20$

From this definition, b_1 is defined on the range $[10, +\infty)$, b_2 on $(-\infty, 15)$, and b_3 on $(20, +\infty)$. b_1 and b_2 are thus non-disjoint, like b_1 and b_3. Indeed, b_1 and b_2 are both defined on the range $[10, 15)$, and b_1 and b_3 on $(20, +\infty)$. However, b_2 and b_3 are disjoint as it does not exist any common range between their definition.

5.3.2 Conciseness

Regarding the syntax, we distinguish three kinds of operations which manipulate data: writing[7], reading, and combination. For these three kinds of operations, we compare the number of lines of codes required by our language and the two frameworks. To perform this, we implemented a simplified version of our case study using these three solutions. Excerpts of the source code are shown in Listing 5.1-5.3.

Writing and reading operations remain similar to what exists in many other programming languages. There is no difference between the three solutions to this aspect. In the listings, we can notice that all lines of code relative to it have the same size: line 6, 11, and 20 in Listing 5.1; line 7, 11-12, and 21 in Listing 5.2; and line 9, 16, and 27 in Listing 5.3.

Both languages used for developing the frameworks (C#[8] and Python[9]) allow overloading operators. As highlighted in the listings, the three solutions have the same syntax to combine (through arithmetic or boolean operators) uncertain and certain data: lines 9 and 23 in Listing 5.1; lines 10 and 23 in Listing 5.2; and lines 13 and 34 in Listing 5.3.

In contrary to OpenTurns and our solution, Infer.NET requires an explicit call to the engine which computes and propagates the probabilities. We can see these calls in line 10, 16 and 22 in Listing 5.1. This framework is mainly done for probabilistic programming, where developers implement a model, then executes it. From our understanding of this tool, it was not designed to allow an iterative propagation of uncertainty and a control flow that depends on this propagation. Thus, we need to split the propagation in different models and explicitly call their inference engine to enable such a process.

Finally, all three solutions allow reasoning on uncertain data. In Listing 5.1 at line 17, Listing 5.2 at line 18, and Listing 5.3 at line 24, we highlight an IF-condition

[7]This operation also includes the creation and the deletion. The first one corresponds to the first write operation and the last one is done by setting the value to *NULL*.

[8]`https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/statements-expressions-operators/overloadable-operators`

[9]`https://docs.python.org/3/reference/datamodel.html#special-method-names`

based on the confidence of an uncertain boolean[10]. Infer.NET provides a method to help manipulate uncertain booleans, by providing methods to access the probability of the *TRUE* and *FALSE* value. In this case, our confidence operator can be thought of as a sugar syntax of these methods.

d-expr-csharp⟩

```
public void ComputeLoadNoCable(Substation substation) {
  Variable<bool> noCableConn = Variable.Bernoulli(1);
  if(substation.Fuses.Count > 0)
    noCableConn = substation.Fuses[0].IsClosed;
  else
    substation.Load = GaussianFromMeanAndVariance(0, 0.001);
    return;
  for(int i=1; i<substation.Fuses.Count; i++)
    noCableConn = noCableConn & substation.Fuses[i].IsClosed;
  Bernoulli bernoulli = (Bernoulli) InferenceEngine.Infer(noCableConn);
  substation.Load = GaussianFromMeanAndVariance(0, bernoulli.GetVariance());
}

public void ComputeLoad(Substation substation) {
  Variable<bool> isDisco = substation.IsDisconnected();
  Bernoulli bern = (Bernoulli) InferenceEngine.Infer(isDisco);
  if(bern.GetProbTrue() >= 0.95)
    ComputeLoadNoCable(substation);
    return;
  Variable<double> load = GaussianFromMeanAndVariance(0, 0.001);
  foreach(Fuse fuse in substation.Fuses)
    Bernoulli isClosedBern = (Bernoulli) InferenceEngine.Infer(fuse.IsClosed);
    load = load + (fuse.Cable.Load * isClosedBern.GetProbTrue());
  substation.Load = load;
}
```

Listing 5.1: Excerpt of the Infer.NET implementation (C#)

d-expr-python⟩

```
def compute_load_no_cable(substation):
  if not isinstance(substation, Substation):
      raise TypeError('Wrong type')
  if len(substation.fuses) > 0:
      no_cable_conn = substation.fuses[0].isClosed
  else:
      substation.load = ot.Normal(0, 0.001)
      return
```

[10]As OpenTurns does not include an uncertain boolean implementation, we implemented one similar to the one we provide in Ain'tea, using their Bernoulli implementation. The implementation is accessible in our GitHub repository

```
  for fuse in substation.fuses[1:]:
      no_cable_conn = no_cable_conn & fuse.isClosed
  substation.load = ot.Normal(0,
    no_cable_conn.getStandardDeviation()*no_cable_conn.getStandardDeviation())

def compute_load(substation):
  if not isinstance(substation, Substation):
      raise TypeError('Wrong type')
  is_disco = substation.is_disconnected()
  if is_disco.exist(0.95) and is_disco.value_with_confidence(0.95):
      compute_load_no_cable(substation)
      return
  load = ot.Normal(0, 0.001)
  for fuse in substation.fuses:
      load = load + (fuse.cable.load*fuse.isClosed.confidence.getP())
  substation.load = load
```

Listing 5.2: Excerpt of the OpenTurns implementation (Python)

st:valid-expruscript⟩

```
void ComputeLoadNoCable(Substation substation) {
  Bernoulli<bool> noCableConn = new Bernoulli<bool>(true, 1);
  Fuse[] fuses = substation.fuses;
  Fuse f;
  if(fuses.length > 0)
    f = fuses[0];
    noCableConn = f.isClosed;
  else
    substation.load = new Gaussian<double>(0, 0.001);
    return;
  for(int i=1; i<fuses.length; i=i+1)
    f = fuses[i];
    noCableConn = noCableConn && f.isClosed;
  Bernoulli bern = noCableConn.confidence;
  if(bern.probability >= 0.97) {
    substation.load = new Gaussian<double>(0, 0.0289);
  } else {
    [...]
  }
}

void ComputeLoad(Substation substation) {
  Bernoulli<bool> isDisco = substation.isDisconnected();
  if(exist(isDisco, 0.95) && isDisco[0.95])
    ComputeLoadNoCable(substation);
    return;
  Gaussian<double> load = new Gaussian<double>(0, 0.001);
```

```
  Fuse[] fuses = substation.fuses;                                    28
  for(int i=0; i<fuses.length; i=i+1)                                 29
    Fuse f = fuses[i];                                                30
    Cable c = f.cable;                                                31
    Bernoulli<bool> isClosed = f.isClosed;                            32
    Bernoulli bern = isClosed.confidence;                             33
    load = load + c.load * bern.probability;                          34
  substation.load = load;                                             35
}                                                                     36
```

Listing 5.3: Excerpt of the Ain'tea implementation

RQ1 aims at evaluating the impact of native uncertainty management on the conciseness of a language. As shown by the above evaluation, adding uncertainty as a first-class citizen can be done without damaging the conciseness. As we have seen through this paper, managing uncertainty impacts the semantics of the language, by complexifying the traditional operators (arithmetic, boolean, comparison). A few operators can also be added in order to enable reasoning over the uncertainty. The impact on the concrete syntax is thus limited.

However, developers will use the same syntax to manipulate more complex concepts. Adding two uncertain numbers and two numbers is semantically different. One threat to validity is the lack of impact assessment on developers, who may have difficulty manipulating these concepts. However, as we hide this complexity behind traditional operators, the risk is rather low.

5.3.3 Error handling at development time

Among the different root causes of typing errors, one is code refactoring. In order to validate our approach, we thus define a scenario based on it.

Let's assume a developer implements the load approximation for a smart grid. She/he must consider different scenarios that can exist from a simple cable to a more complex situation which looks like a small graph. As the code is complex, in one place (*e.g.,* a file) she/he defines the smart grid classes and she/he implements this computation in another one. At first, the developer decides to use the Dirac delta function to represent the uncertainty of the substation load. During the development process, she/he notices that the Gaussian one suit more his/her problem. However,

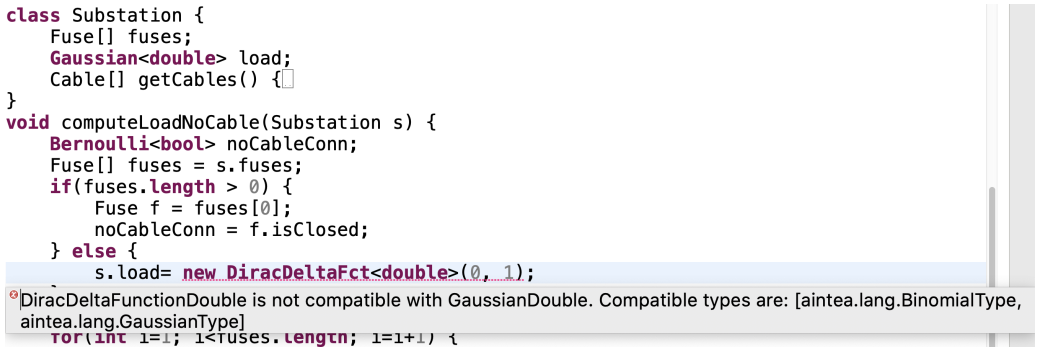


Figure 5.3: Detection of a type error

valid-error-handling⟩

all the operations are not compatible with this new type, for example when there is no cable connected.

As we depict in Figure 5.3, our approach can detect this type of error statically. Plus, we can also notice that we can help developers fix this kind of issue by listing the compatible types with the new substation type ($Gaussian<double>$).

Among the two selected framework, only Infer.NET is implemented in a language which supports static type checking: C#[11]. We also implement a simplified example that raises a typing error. To do so, we perform an addition between two incompatible probability distributions: a Gaussian and a Gamma[12]. As we show in Figure 5.4, the error is only detected at runtime.

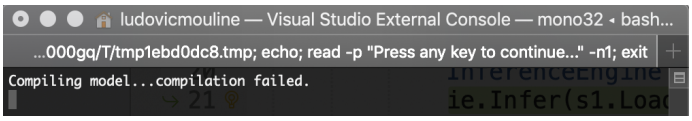
In addition, we see in Figure 5.3 that by specifying the type system, the language is able to provide high-level explanations of the errors. Here, it explains that the two distributions are not compatible plus gives the list of compatible distributions with the field type. In contrary, in Figure 5.4, the error is detected later than its root cause and informs that the operation is not supported. Indeed, as depicted in Figure 5.4a, the error of line 19 is detected at line 21 (highlighted in green) without any reference to this operation. Plus, the error message detailed all type mismatches between the declaration of the method and the given types.

This evaluation validates the ability of a type system to detect errors related to

[11] `https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/types/index`
[12] The Rayleigh distribution is not present in the framework. We thus choose another one.

```
10  Substation s1 = new Substation();
11  Cable c1 = new Cable
12  {
13      Load = Variable.GaussianFromMeanAndVariance(10, 0.5).Named("Load_Cable_1")
14  };
15  Cable c2 = new Cable
16  {
17      Load = Variable.GammaFromMeanAndVariance(10, 0.5).Named("Load_Cable_2")
18  };
19  s1.Load = (c1.Load + c2.Load).Named("Load_Substation_1");
20  InferenceEngine ie = new InferenceEngine();
21  ie.Infer(s1.Load);
```

(a) Code with a type error, detected at the green highlighted line



(b) Console output

Microsoft.ML.Probabilistic.Compiler.CompilationFailedException has been thrown

MessageTransform failed with 3 error(s) and 0 warning(s):
Error 0: This model is not supported with ExpectationPropagation due to Factor.Plus(double Sum, double A, double B). Try using a different algorithm or expressing the model differently in Factor.Plus(Load_Cable_1_use, Load_Cable_2_use)
Details: [0] System.ArgumentException: Gaussian is not of type WrappedGaussian for argument 1 of method DoublePlusOp.SumAverageConditional(WrappedGaussian a = Gaussian, WrappedGaussian b = Gamma)
[1] System.ArgumentException: Gaussian is not of type WrappedGaussian for argument 1 of method DoublePlusOp.SumAverageConditional(WrappedGaussian a = Gaussian, double b = Gamma)

(c) Excerpt of the error message

Figure 5.4: Infer.NET detect the error at runtime.

csharp-global⟩

uncertainty management (RQ2). When a type system is implemented, error messages are defined to help developers. And so, support to develop uncertainty-aware software can be provided.

5.3.4 Discussion

By mapping arithmetic and boolean operators to the propagation of uncertainty, we hide the complexity of the combination of probability distributions. It helps developers to stay in the paradigm that they use every day, introducing as few concepts as possible. This being said, as we intend to have a language that manipulates several kinds of uncertainties, they still need a high-level knowledge of probability distributions. For

example, they should understand the difference between a Gaussian and a Rayleigh. But they do not need to know if or how they can be combined. Indeed, thanks to our type checker, the language will provide information or raise errors when it is not possible.

However, if the uncertainty of a domain can be represented using only one probability distribution, thus this distribution could also be hidden for the developer. Therefore, they will only manipulate uncertain and certain data. For example, using the approach introduced in [MWV16], a developer will only manipulate *UReal* and *Real* for uncertain and certain numbers.

By hiding the complexity of the combination of probability distributions, we also hide the computation overhead introduced. This work has not been done with performance as a goal. We do not guarantee any performance of our language. The evaluation lacks a quantification of this overhead.

5.4 Conclusion

ec:aintea:conclusion⟩ Data are inherently uncertain. This uncertainty can modify the understanding of the elements represented by these data. Furthermore, it may damage the decision made upon this knowledge. Managing this uncertainty calls for a strong understanding of probability theory. However, it can be far from developers expertise. In order to help them manipulating this uncertainty, we define Ain'tea, a language which integrates concepts related to data uncertainty. Mainly, we define uncertain data types and we map uncertainty propagation to arithmetic, boolean, and comparison operators. In addition, we define operators specific to the manipulation of the uncertainty representation. In our validation, we show that our language is as concise as state-of-the-art solutions. Contrary to these solutions, we also show that our solution can detect errors earlier. Thanks to the semantics, which supports uncertainty, errors message help developers in their development of algorithms that use uncertain data.

6

A temporal knowledge metamodel of adaptive systems

⟨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 [Tea15] 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 these have been taken. We identified two challenges caused by this difference of rates:

- How to model the decisions of an adaptation process to diagnose it?
- 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 the decisions, the adaptation process could consider ongoing actions with their expected effects, also called impacts. Besides, in case of faulty decisions, developers may trace back effects to their circumstances. Our current approach is limited to the representation of measurable effects of any decision, and therefore action.

Our approach allows 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 3). Then, we provide a formal definition of these concepts in Section 6.2. After, 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.

6.2 Knowledge formalization

m:k-formalism⟩ 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. 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 3.3.2.

6.2.1 Formalization of the temporal axis

lism:timeAxis⟩ The formalism described below has been defined with two goals in mind. First, the definition of the time space should allow the distinction between past and future. Making this distinction enables the differentiation between measured data

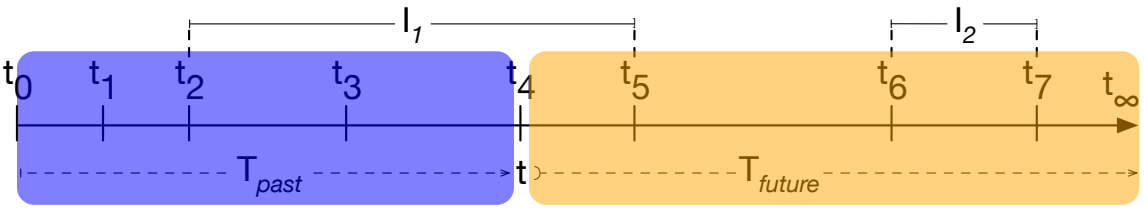


Figure 6.1: Time definition used for the knowledge formalism

g:tkm:formalismeTime⟩

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 6.1, this set can be divided into 3 different subsets $T = T_{past} \cup \{t\} \cup T_{future}$, where:

- T_{past} is the subdomain $\{t_0;t_1;\ldots;t_{current-1}\}$ 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 $\{t_{current+1};\ldots;t_\infty\}$ 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 6.1 displays an example of a time space $T_1 = \{t_0, t_1, t_2, t_3, t_4, t_5, t_6, t_7\}$. In this case, 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

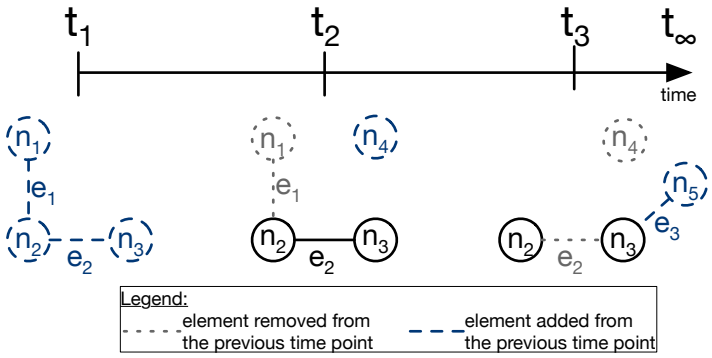


Figure 6.2: Evolution of a temporal graph over time

km:validityEx⟩

interval could be defined on different subsets, in this case it is on all of them (T_{past}, t, and T_{future}).

6.2.2 Formalism of the knowledge

ism: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: $\forall n \in N, n = (id, P)$, 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 6.2.1) during which the graph element is valid.

Figure 6.2 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 6.2, 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 6.2, 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 $P_{[t_l,t_m]}$, $A_F : K \times I \to K \times I$.

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, $E_{A_E} : A_E \to I$.

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 in D is defined as a set of executed actions, *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: $\forall a \in A, \forall d1, d2 \in D \mid a \in d1 \wedge a \in d2 \to d1 = d2$.

The temporal function E_{A_E} is extended to decisions in order to represent the execution time: $E_{A_E} : (A \cup D) \to I$. 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 $l = \min_{a \in A_d}\{E_{A_E}(a)[start]\}$ and $u = \max_{a \in A_d}\{E_{A_E}(a)[end]\}$.

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)
- 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 3.3.2.

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 6.3 shows the nodes and edges of this knowledge.

Description of 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 6.3, which represents the requirement of this example

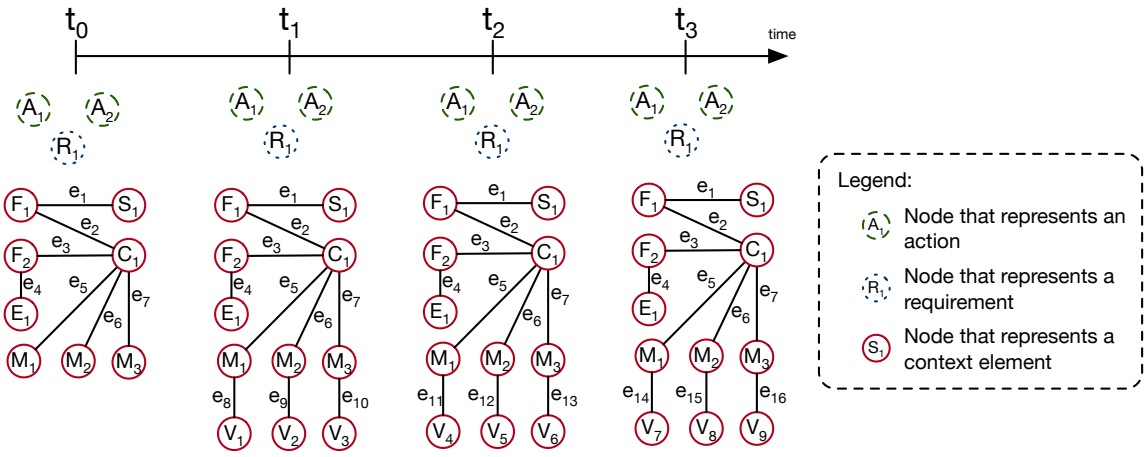


Figure 6.3: Application of the formalism with a temporal graph that represents the knowledge of the smart grid described in Section 3.3.2

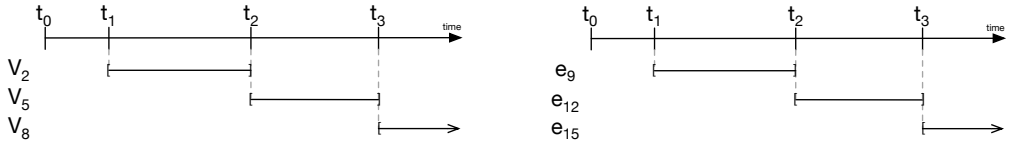
ormalism:application⟩

(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 6.3, 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,



(a) Consumption values C_2

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

Figure 6.4: Validity periods of consumption values and their edges to the smart meter M_2

itself replaced by V_8 at t_3. Applying the V_{SG}^T on these different values, results are thus:

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

These validity periods are shown in Figure 6.4a. As meters send the new consumption values at the same time, this example can also be applied to C_1 and C_3.

From these validity periods, the Z_{SG}^T 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_{SG}^T relation:

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

Description of 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: $\forall i \in [1..7], V^T_{SG}(e_i) = [t_0, t\infty)$. 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:

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

These validity periods are depicted in Figure 6.4b. As they are driven by those of consumption values (V_2, V_5, and V_8), they are equal.

As for nodes, the Z^T_{SG} 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^T_{SG} relation as follows:

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

Description of D_{SG}, $A_{E_{SG}}$, and $R_{A_{SG}}$ As described in the scenario (cf. Section 3.3.2), 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 3.3.1, 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 6.5– 6.8. 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 dimension: *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).

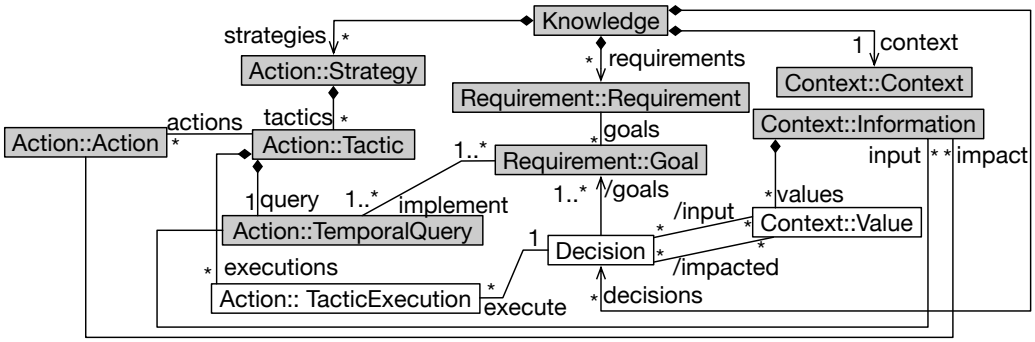


Figure 6.5: Excerpt of the knowledge metamodel

:knowledge-mm⟩

6.3.2 Knowledge metamodel

m: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 6.5 depicts this metamodel.

Knowledge package is composed of a *context*, a set of *requirements*, a set of *strategies*, and a set of *decisions*. A decision can be seen as the output of the Analyze and Plan steps in the 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. 6.8). 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

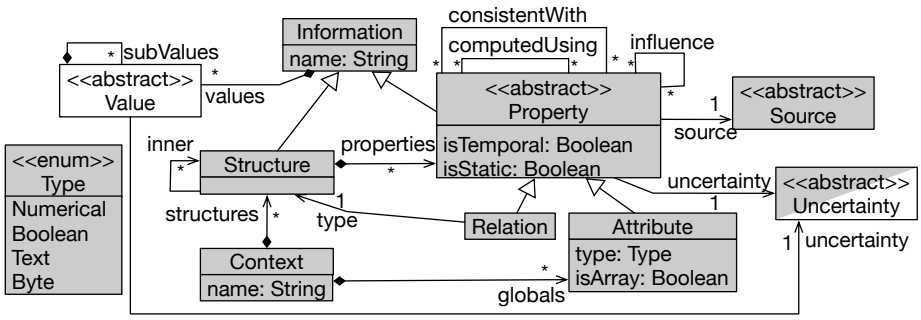


Figure 6.6: Excerpt of the context metamodel

⟨fig:context-model⟩

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*). An example for such in interactive diagnosis can be found in Section 3.3.2.

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 6.6. 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 2.1.3 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.

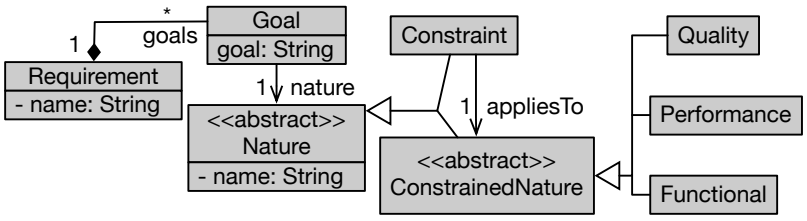


Figure 6.7: Requirement metamodel

irement-model⟩

6.3.4 Requirement metamodel

As different solutions to model system requirements exist (*e.g.,* KAOS [DvLF93], i* [Yu11] or Tropos [BPG+04]), in this metamodel, we abstract their shared concepts. The requirement model, depicted in Figure 6.7, 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 2.1.3. 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 [Egy01] is one solution among others. This work is out of the scope of the thesis 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

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 6.8 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 (*TemporalQuery*) and uses different data from the context as input. In future work, we will investigate the use of preconditions to schedule the executions

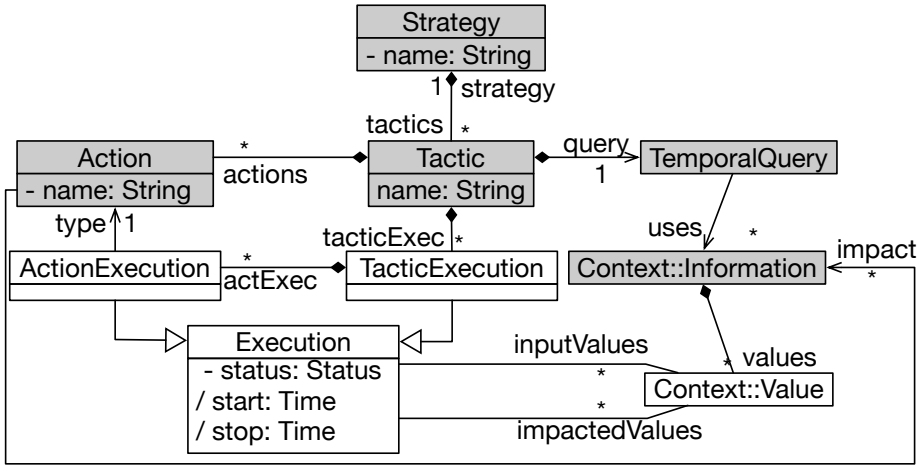


Figure 6.8: Excerpt of the action metamodel

⟨fig:action-mm⟩

order of the actions, similarly to existing formalisms such as Stitch [CG12]. 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 thesis and planned

for future work.

6.4 Validation

km:validation⟩ To validate and evaluate our approach, we implemented a prototype publicly available online[1]. This implementation leverages the GreyCat framework[2], more precisely the modelling plugin, which allows designing a metamodel 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.

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 6.9. Due to space limitation, we only present an ex-

[1]https://github.com/lmouline/LDAS
[2]https://github.com/datathings/greycat

cerpt 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 6.9 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 6.9.

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

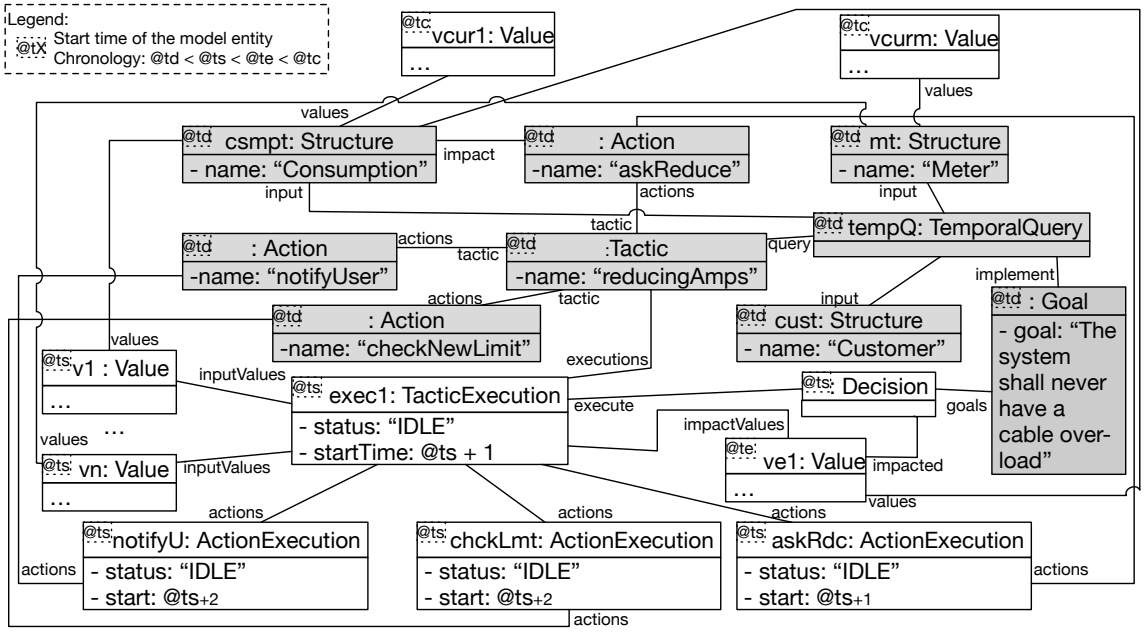


Figure 6.9: Excerpt of the knowledge object model related to our smart grid example

km:valid:diag⟩

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 3.3.2, 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.* Chapter 3).

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 3.3.1, this action is considered as long-term 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.,* [HFJ+17; Har16].

⟨code:traversal-used⟩

```
MATCH (input)-[*4]->(output)
WHERE input.id IN [randomly generated set]
RETURN output
LIMIT 0
```

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[3]. 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.

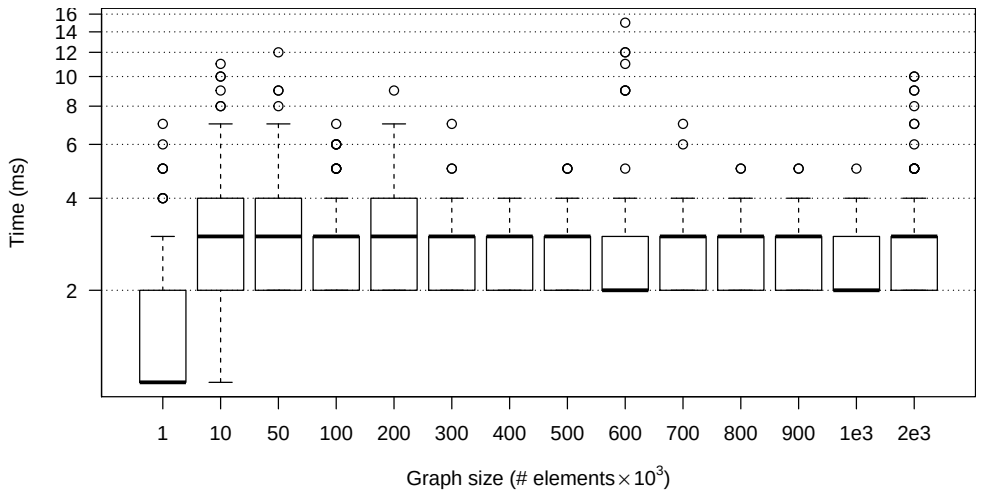
We executed our experimentation on a MacBook Pro with an Intel Core i7 processor (2.6 GHz, 4 cores, 16GB main memory (RAM), macOS High Sierra version 10.13.2). We used the Oracle JDK version 1.8.0_65.

How is the performance 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 6.10.

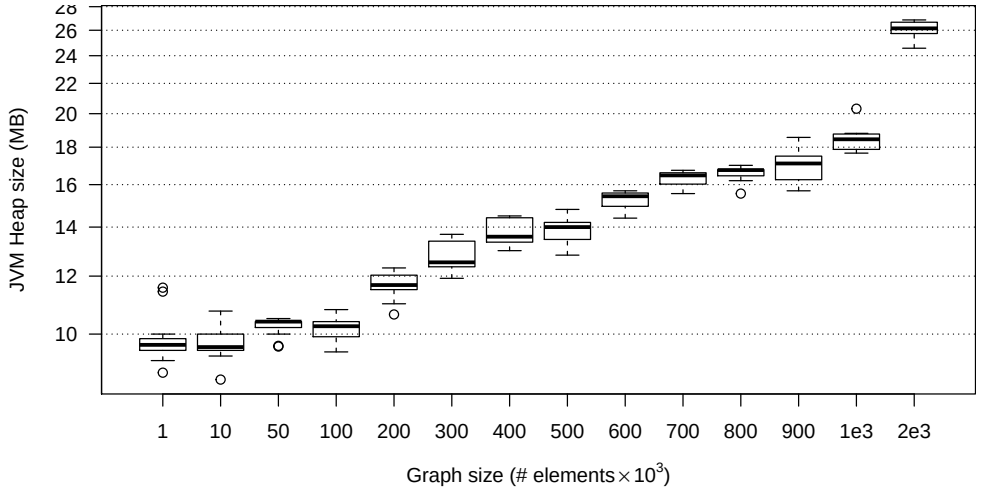
As we can notice, the graph size does not have a significant impact on the execution time of diagnosis algorithms. For graphs with up to 2,000,000 elements, execution time remains between 2 ms and four 4 ms. We can also notice that the memory consumption insignificantly increases. Thanks to the implementation of a lazy loading 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 is the performance influenced by the input size (I)? The second experiment aims to show the impact of the input size (I) on the execution of diagnosis

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



(a) Execution time evolution



(b) Memory evolution

Figure 6.10: Experimentation results when the knowledge based size increases

⟨fig:exp1⟩

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 6.11 (straight lines).

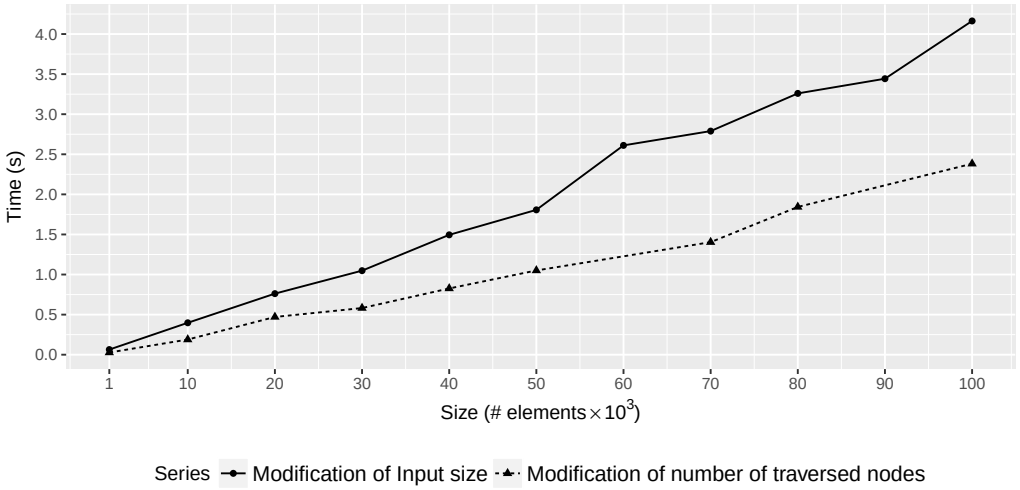
Unlike to the previous experiment, we notice that the input size (I) impacts the performance, both in terms of memory consumption and execution time. This is because our framework keeps in memory all the traversed elements, namely the input elements. The increase in memory consumption follows a linear 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 is the performance 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 6.11 (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.

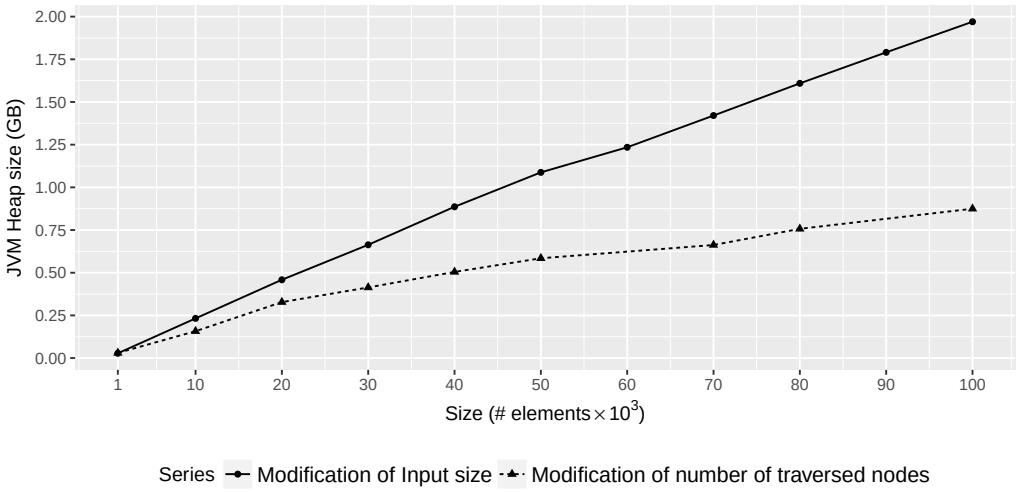
6.4.4 Discussion

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 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



(a) Evolution of the execution time



(b) Evolution of the memory consumption

Figure 6.11: Results of experiments when the number of traversed or input elements increases

⟨fig:exp-res⟩

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

[4]Previously, our studies uses the data described in [HFK+14b], 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.

Part III

Conclusion and future works

7

Conclusion

pt:conclusion⟩

Contents

In this chapter, we conclude our dissertation by summarising the challenges and the contributions presented in this document. Then, we detail a set of future works. We finish this dissertation by giving an outlook on this thesis.

7.1 Summary

Software systems are more and more pervasive and evolve in a more and more complex environment. This complexity comes with a price: the environment is less and less known with high confidence. To face this challenge, systems became adaptive: their structure and their behaviour can be adjusted at runtime in response to changes in the environment, behaviour, or even their specification. One example of such a system is a smart grid: a power grid which includes ICT to optimise energy delivery and service quality. They continuously monitor cable loads to detect any overloads or outages. If one is detected, the system, triggers a modification of power flow to heal the system.

One way to implement an adaptive system, like a smart grid, is to apply the models@run.time paradigm. This paradigm is part of the MDE methodology, which advocates for the use of models in software engineering. As mentioned in the name, models@run.time defines runtime model with a causal link to the system. A runtime model reflects the state of a system, during its execution. Any modification of the system triggers an update of the model, and vice-versa (causal link).

However, we have identified five problems that can reduce the usability of this approach (*cf.* Chapter 1). First, data collected are, for the most part, uncertain, which can mislead the understanding of a system's behaviour, structure, or environment. Second, adaptation actions are never immediate, take time to be executed, and have long-term effects. We refer to these actions as long-term actions. Third, these systems can have emergent behaviour. Four, the different components of a system can evolve at different rates. Last, the evolution of the system is linked with time.

Within this dissertation, we argue towards a novel modelling framework that efficiently encapsulates time and uncertainty as first-class concepts. Therefore, in this thesis, we focus on three challenges:

1. How to ease the manipulation of data uncertainty?
2. How to enable reasoning over unfinished actions and their expected effects?
3. How to diagnose the self-adaptation process?

In order to solve them, we propose two contributions in this dissertation. First,

to answer the first challenge, we defined a language that integrates concepts related to data uncertainty as a first-class citizen named Ain'tea (*cf.* Chapter 5). In addition to traditional object-oriented language type (boolean, numeric, reference), the language has uncertain boolean, uncertain numeric, and uncertain reference. These types encapsulate two elements: one element to represent the value, and one to represent the uncertainty. Following what has been done in probabilistic programming, the uncertainty is abstracted by a probability distribution. Ain'tea manages five distributions. One distribution is used for uncertain boolean and reference: Bernoulli. The four other distributions are employed for uncertain numbers: Gaussian, Rayleigh, binomial, and Dirac delta function.

As we add new data types, we also modify the language operators and the type system in consequence. Following what is done in the literature, we map the different operators (arithmetic, boolean, comparison) to a process to propagate uncertainty. Therefore, a developer can combine uncertain data as she is used to doing with certain data. Additionally, we define specific operators to reason over uncertainty, such as the *confidence* and *exist* operators. The type system natively considers these new data types. Statically, it checks if the combination of uncertain data follows the constraint of uncertainty propagation, that is the probability theory. If not, error messages are thrown during the development time to guide developers in their choices.

In our validation, we show that our language is as concise as state-of-the-art solutions. Contrary to these solutions, we also show that our solution can detect errors earlier. Thanks to the semantics, which supports uncertainty, errors message help developers in their development of algorithms that use uncertain data.

Our second contribution, which tackles the two other challenges, is a temporal knowledge metamodel (*cf.* Chapter 6). Contrary to state-of-the-art solutions, this metamodel adds the concept of long-term action. 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 effects. Moreover, the metamodel is also supported by a graph-based formalism. We formalise the different elements of the knowledge of an adaptive system: requirements, actions (design time and runtime), and context.

When a designer defines a model conforms to our metamodel, she will benefit from an API. An engineer can use this API to interact with the model at runtime. In our evaluation, we showed that our solution can efficiently handle up to 100 000 elements in a single machine. This size is comparable to 5 days history of the Luxembourg smart grid.

These contributions do not answer all questions related to the problems identified. Besides, they suffer from limitations that can lead to further research efforts. In the next section, we detailed these limitations with the perspectives.

7.2 Future works

Both our contributions suffer from limitations that call for further research. First, our language Ain'tea do not include all the different kinds of uncertainty representations. Moreover, uncertainty is only applied to simple data type, and the propagation is limited to some of the language operators (arithmetic, boolean, and comparison). Second, when we define our temporal knowledge model, we assumed that designers can link actions with their expected impacts at design time. Then, the final metamodel remains large. In this section, we describe research directions that could remove these limitations, and other perspectives let by our work.

7.2.1 Other uncertainty representation

Our current work addresses only uncertainty on values and with a limited number of probability distributions. However, other kinds of uncertainties exist, such as uncertainty of existence or temporal uncertainty. The first one corresponds to the confidence that a value exists or not. It can result from faulty data sources that send wrong data. The second one can be used to represent the loss of confidence in value over time. Moreover, researchers defined other strategies to represent uncertain data, like keeping multiple possibilities.

First future work would be to investigate how to introduce such techniques inside a programming language. Then, research efforts can focus on the definition of a language that uses different strategies to handle uncertainty. It will open new challenges regarding the type system, the semantics, and the syntax of the language.

Second, introducing new probability distributions lead to complex combinations of probability distributions. In the current approach, we use an analytical approach (we compute the exact solution). However, this cannot be performed between some probability distributions. In such a situation, a numerical method should be applied. This leads to challenges regarding the threshold between the performance of the language and the accuracy of the method.

7.2.2 More complex data structure

In our language, we focus our studies on the primitive data types (numeric and boolean) and references (1:1 relation). However, it exists several other data structures, from the simplest ones like arrays to complex ones like graphs or trees. In the UML specification, different kinds of relations have been defined (1:1, 1:n, n:n, *etc.*).

While these data structure and relations are useful to build algorithms to reason over data, they come with new challenges. First, it opens new questions concerning the meaning of an uncertain data structure: what an uncertain array? What an uncertain tree, graph? Let us take an uncertain array. The uncertainty can be related to the full collection, or on each element, or both. Additionally, research efforts have to be done to specify the semantics of operators on these uncertain structures. For example, one may focus on how the uncertainty of the array will be impacted by an *add* or *remove* operation. Lastly, one may investigate the impact of introducing uncertain data structure in common algorithms. For example: how to sort an uncertain collection? How to balance an uncertain tree? How to compute the shortest path?

7.2.3 Impact of uncertainty to control flow

In our language, we map the uncertainty propagation to operators. We did not study the impact of control flow statements on the uncertainty propagation. Introducing uncertainty as a first-class citizen will inevitably modify current behaviour of control flow statements such as *IF*-conditions. We strongly think that this new data type in the type system will lead to further research directions. We identified two other situations that should be considered, with their challenges. The first one is the

propagation through control flow statements: how the uncertainty is propagated after an *IF*-condition, a *FOR*-loop and a *WHILE*-loop. The second one is the propagation from one kind of uncertainty to another one. For example, how the uncertainty of presence should be propagated to the uncertainty of a sum, average or variance computation?

Conditional expressions, which have a boolean type, modify the control flow by forking it. With certain boolean, the expression is evaluated at runtime, and one branch is selected according to the result. With uncertain ones, the executor cannot decide which branch to execute. We thus identified two possible controls flows: the classical and the uncertain execution.

For the classical one, a cast operation should be performed to get a certain one. It can be done using at least two strategies. First, a random selection from the probability distribution can be made. Second, a cast can be done using the *cast* or *confidence* operator of our language (*cf.* Chapter 5).

For the uncertain execution, as the executor cannot decide which branch should be executed, it should execute all of them and propagate the uncertainty. For example, let us imagine the following code:

```
uncertain_bool b = (TRUE, 0.4)
uncertain_int n;
if(uncertain_bool)
 n = (5, 0.8)
else
 n = (-5, 0.8)
```

Listing 7.1: Example for uncertain control flow

As b is true with a confidence of 40%, n should be equal to 5 with a confidence of (40% * 80% = 32%) and to -5 with a confidence of (60% * 80% = 48%) (considering b and n independent).

These two execution semantics come with several open questions. One is related to the impact of such techniques to performances (CPU, memory, *etc.*). For the uncertain execution, additional question persists such as: should the execution be parallel or not? How to ensure that the execution does not have side effect(s)? And a final one: can other semantics be defined?

7.2.4 Unknown effects of actions

When we defined our temporal knowledge model (*cf.* Chapter 6), we assumed that designers can link actions with their expected effects at design time with the most thorough confidence. However, as systems are more and more complex, we think that they do not know all the impacts in advance. As done by Donald Rumsfeld in a famous US Department of Defense press briefing, we can identify two levels of unknowns: unknown knowns, and unknown unknowns. Moreover, these relations are uncertain. All traceability links that we model in this paper should not be considered as entirely accurate.

Research efforts are, therefore, needed to define techniques to discover these unknown relations. One approach could be to use a learning algorithm to find them. Besides, research efforts are still required to combine our two contributions and achieve our vision of an uncertain and time-aware modelling framework.

7.2.5 Manipulation and population of the large model

We are conscious that the metamodel defined in Chapter 6 remains large. A model that conforms this metamodel can be challenging to be defined and manipulated manually.

Research efforts are required to define DSL to facilitate the manipulation of the model. Moreover, autonomous processes can be defined to populate a model conforms to our metamodel automatically. For example, one can specify a method for analysing the code or the model that describes actions to populate the temporal knowledge with long-term actions. Another process can be formalised to add model elements related to context information, the status of action executions, *etc.*.

7.2.6 Uncertainty evaluation and reduction

In our work on uncertainty, we focus on the definition of uncertain data and the propagation. However, while talking about uncertainty, there are two other key concepts: evaluation and reduction. The former is to evaluate the uncertainty at runtime of a received data, *e.g.,* to find the appropriated probability distribution with its parameter. The latter is to increase the confidence of received value.

The literature provides techniques to evaluate the uncertainty [WKE08; Met08] and to reduce it [Sha76]. Thus, we think that perspectives for the modelling community are open. The community should focus on a model that abstracts these techniques to help engineers to manipulate and integrate them in their model.

Glossary

adaptive system In this document, we modified the definition of self-adaptive systems provided by Cheng *et al.,* in [CdLG+09]. 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.
action In this document, we use the definition provided by IEEE Standards [III17]: "Process that, given the context and requirements as input, adjusts the system behaviour".

behaviour We refer to system behaviour.

circumstance State of the knowledge when a decision has been taken.
context In this document, we use the definition provided by Anind K. Dey [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".

data uncertainty Data are uncertain when some data points are not precisely known..
decision A set of actions taken after comparing the state of the knowledge with the requirement.
DSL In this document, we use the definition provided by Deursen *et al.,* [vDKV00]: "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.".

environment See system environment.

knowledge The knowledge of an adaptive system gathers information about the context, actions and requirements.

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

MAPE-k A theoretical model of the adaptation process proposed by Kephart and Chess [KC03]. It divides the process in four stages: monitoring, analysing, planning and executing. These four stages share a knowledge.

metamodel In this document, we use the definition provided by Douglas C. Schmidt [Sch06]: "[Metamodels] define the relationships among concepts in a domain and precisely specify the key semantics and constraints associated with these domain concepts".

model In this document, we use the definition provided by Brambilla *et al.,* [BCW17]: "[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 [BJT05].

models@run.time In this document, we use the definition provided by Blair *et al.,* [BBF09]: "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".

requirement In this document, we use the definition provided by IEEE Standards [III17]: "(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".

self-adaptive system See adaptive system.

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

self-healing system In this document, we use the definition provided by Kephart and Chess [KC03]: "[A self-healing] system automatically detects, diagnoses, and repairs localised software and hardware problems.".

SLE In this document, we use the definition provided by Anneke Kleppe [Kle08]: "the application of systematic, disciplined, and measurable approaches to the development, use, deployment, and maintenance of software languages".

smart grid In this document, we use the definition provided by the National Institute of Standards and Technology (NIST) [oSN]: "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.".

software language In this document, we use the definition provided by Anneke Kleppe [Kle08]: "any language that is created to describe and create software systems".

structure See system structure.

Abbreviations

API Application Programming Interface. 14, 31, 158

CPS Cyber-Physical System. 4, 77, 78
CPU Central Processing Unit. 7, 160
CSV Comma-separated values. 56

DOM Document Object Model. 32
DSL Domain-Specific Language. 32, 161, *Glossary:* DSL
DSML Domain Specific Modelling Language. 26–28, 32

E-MOF Essential MOF (EMOF). 30
EMF Eclipse Modelling Framework. 29, 30

FSM Final State Machine. 59, 60

GCM GreyCat Modelling Environment. 30, 31
GPL General Purpose Language. 32
GUM Guide to the expression of Uncertainty in Measurement (GUM). 78

ICT Information and communication technology. 4, 156
IoT Internet of Things. 30

KMF Kevoree Modelling Framework. 30

MAPE-k Monitor, Analyse, Plan, and Execute over knowledge. 20, 21, 25, 67, 70, 137, *Glossary:* MAPE-k

MDE Model-Driven Engineering. 4, 5, 11, 15, 17, 21, 25–29, 32, 56, 66, 69, 72, 156
MOF Meta Object Facility. 29, 30

OCL Object Constraint Language. 29
OMG Object Management Group. 28

SLE Software Language Engineering. 17, 31, 33, *Glossary:* SLE

UML Unified Modelling Language. 26, 28, 77, 159

XMI XML Metadata Interchange. 29

List of publications and tools

Papers included in the dissertation

- 2017
  - Amine Benelallam, Thomas Hartmann, Ludovic Mouline, François Fouquet, Johann Bourcier, Olivier Barais, and Yves Le Traon. Raising time awareness in model-driven engineering: vision paper. In *20th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS*, 2017. URL: `https://doi.org/10.1109/MODELS.2017.11`
- 2018
  - Ludovic Mouline, Amine Benelallam, François Fouquet, Johann Bourcier, and Olivier Barais. A temporal model for interactive diagnosis of adaptive systems. In *2018 IEEE International Conference on Autonomic Computing, ICAC*, 2018. URL: `https://doi.org/10.1109/ICAC.2018.00029`
  - Ludovic Mouline, Amine Benelallam, Thomas Hartmann, François Fouquet, Johann Bourcier, Brice Morin, and Olivier Barais. Enabling temporal-aware contexts for adaptative distributed systems. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing, SAC*, 2018. URL: `https://doi.org/10.1145/3167132.3167286`
- in the process of submission
  - Ludovic Mouline, Amine Benelallam, Thomas Hartmann, Johann Bourcier, Olivier Barais, and Maxime Cordy. Ain'tea: managing data uncertainty at the language level. *Forthcoming*, forthcoming

Papers not included in the dissertation

- 2017
  - Ludovic Mouline, Thomas Hartmann, François Fouquet, Yves Le Traon, Johann Bourcier, and Olivier Barais. Weaving rules into models@run.time for embedded smart systems. In *Companion to the first International Conference on the Art, Science and Engineering of Programming, Programming*, 2017. URL: `https://doi.org/10.1145/3079368.3079394`
- 2018
  - Alejandro Sánchez Guinea, Andrey Boytsov, Ludovic Mouline, and Yves Le Traon. Continuous identification in smart environments using wrist-worn inertial sensors. In *Proceedings of the 15th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services, MobiQuitous*, 2018. URL: `https://doi.org/10.1145/3286978.3287001`

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 metamodel of knowledge for adaptive systems
  - `https://github.com/lmouline/LDAS`

List of figures

List of tables

Bibliography

[AdLM+09] Jesper Andersson, Rogério de Lemos, Sam Malek, and Danny Weyns. Reflecting on self-adaptive software systems. In *ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems, SEAMS*, 2009. URL: https://doi.org/10.1109/SEAMS.2009.5069072 (cited on page 16).

[AFR+10] Darko Anicic, Paul Fodor, Sebastian Rudolph, Roland Stühmer, Nenad Stojanovic, and Rudi Studer. A rule-based language for complex event processing and reasoning. In *Web Reasoning and Rule Systems RR*, 2010. URL: https://doi.org/10.1007/978-3-642-15918-3%5C_5 (cited on pages 63, 64).

[AGR11] Paolo Arcaini, Angelo Gargantini, and Elvinia Riccobene. CoMA: conformance monitoring of java programs by abstract state machines. In *Runtime Verification RV*, 2011. URL: https://doi.org/10.1007/978-3-642-29860-8_17 (cited on pages 55, 56).

[AK03] Colin Atkinson and Thomas Kühne. Model-driven development: a metamodeling foundation. *IEEE Software*, 2003. URL: https://doi.org/10.1109/MS.2003.1231149 (cited on page 24).

[ARS15] Paolo Arcaini, Elvinia Riccobene, and Patrizia Scandurra. Modeling and analyzing MAPE-K feedback loops for self-adaptation. In *10th IEEE/ACM International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS*, 2015. URL: https:

//doi.org/10.1109/SEAMS.2015.10 (cited on pages 55, 56, 58, 59, 62, 63).

[AY09] Charu C. Aggarwal and Philip S. Yu. A survey of uncertain data algorithms and applications. *IEEE Transactions on Knowledge and Data Engineering*, 2009. URL: https://doi.org/10.1109/TKDE.2008.190 (cited on page 5).

[BAV+12] Sooraj Bhat, Ashish Agarwal, Richard W. Vuduc, and Alexander G. Gray. A type theory for probability density functions. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*, 2012. URL: https://doi.org/10.1145/2103656.2103721 (cited on pages 66, 67, 69, 70, 72, 73).

[BBF09] Gordon S. Blair, Nelly Bencomo, and Robert B. France. Models@run.time *IEEE Computer*, 2009. URL: https://doi.org/10.1109/MC.2009.326 (cited on pages 4, 19, 48, 54, 55, 62, 63, ii).

[BBG+13] Nelly Bencomo, Amel Bennaceur, Paul Grace, Gordon S. Blair, and Valérie Issarny. The role of models@run.time in supporting on-the-fly interoperability. *Computing*, 2013. URL: https://doi.org/10.1007/s00607-012-0224-x (cited on pages 55, 57).

[BBH+10] Claudio Bettini, Oliver Brdiczka, Karen Henricksen, Jadwiga Indulska, Daniela Nicklas, Anand Ranganathan, and Daniele Riboni. A survey of context modelling and reasoning techniques. *Pervasive and Mobile Computing*, 2010. URL: https://doi.org/10.1016/j.pmcj.2009.06.002 (cited on pages 20, 21).

[BBM+18] Loli Burgueño, Manuel F. Bertoa, Nathalie Moreno, and Antonio Vallecillo. Expressing confidence in models and in model transformation elements. In *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS*, 2018. URL: https://doi.org/10.1145/3239372.3239394 (cited on pages 66–69, 72, 73).

[BCC+15] Erwan Bousse, Jonathan Corley, Benoît Combemale, Jeffrey G. Gray, and Benoit Baudry. Supporting efficient and advanced omniscient debugging for xdsmls. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering, SLE*, 2015. URL: https://dl.acm.org/citation.cfm?id=2814262 (cited on page 10).

[BCG+12] Roberto Bruni, Andrea Corradini, Fabio Gadducci, Alberto Lluch-Lafuente, and Andrea Vandin. Modelling and analyzing adaptive self-assembly strategies with maude. In *Rewriting Logic and Its Applications WRLA 2012*, 2012. URL: https://doi.org/10.1007/978-3-642-34005-5_7 (cited on pages 58, 59).

[BCW17] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice, Second Edition*. 2017. URL: https://doi.org/10.2200/S00751ED2V01Y201701SWE004 (cited on pages 4, 24, ii).

[BDI+17] Michaël Baudin, Anne Dutfoy, Bertrand Iooss, and Anne-Laure Popelin. OpenTURNS: an industrial software for uncertainty quantification in simulation. *Handbook of uncertainty quantification*, 2017. URL: https://doi.org/10.1007/978-3-319-12385-1_64 (cited on pages 7, 66, 67, 69, 70, 72, 73, 80, 81, 102).

[BdMM+17] Davi Monteiro Barbosa, Rómulo Gadelha de Moura Lima, Paulo Henrique Mendes Maia, and Evilásio Costa Junior. Lotus@runtime: a tool for runtime monitoring and verification of self-adaptive systems. In *12th IEEE/ACM International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS*, 2017. URL: https://doi.org/10.1109/SEAMS.2017.18 (cited on pages 8, 11, 55, 57, 59, 62, 63).

[BGG+13] Johannes Borgström, Andrew D. Gordon, Michael Greenberg, James Margetson, and Jurgen Van Gael. Measure transformer semantics for bayesian machine learning. *Logical Methods in Computer Science*, 2013.

URL: https://doi.org/10.2168/LMCS-9(3:11)2013 (cited on pages 66, 67, 69, 70, 72, 73).

[BGP92] Daniel Barbará, Hector Garcia-Molina, and Daryl Porter. The management of probabilistic data. *IEEE Transactions on Knowledge and Data Engineering*, 1992. URL: https://doi.org/10.1109/69.166990 (cited on pages 66, 67, 69, 72).

[BGS+14] Amine Benelallam, Abel Gómez, Gerson Sunyé, Massimo Tisi, and David Launay. Neo4EMF, a scalable persistence layer for EMF models. In *Modelling Foundations and Applications ECMFA*, 2014. URL: https://doi.org/10.1007/978-3-319-09195-2%5C_15 (cited on page 25).

[BHH05] George E. P. Box, J. Stuart Hunter, and William G. Hunter. *Statistics for Experiments: Design, Innovation, and Discovery, 2nd Edition*. Wiley-Interscience, 2005. ISBN: 978-0-471-71813-0 (cited on page 33).

[BHL+01] Tim Berners-Lee, James Hendler, Ora Lassila, et al. The semantic web. *Scientific american*, 2001. URL: https://doi.org/10.1038/scientificamerican0501-34 (cited on page 23).

[BHM+17] Amine Benelallam, Thomas Hartmann, Ludovic Mouline, François Fouquet, Johann Bourcier, Olivier Barais, and Yves Le Traon. Raising time awareness in model-driven engineering: vision paper. In *20th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS*, 2017. URL: https://doi.org/10.1109/MODELS.2017.11 (cited on pages 5, vii).

[BJT05] Jean Bézivin, Frédéric Jouault, and David Touzet. Principles, standards and tools for model engineering. In *10th International Conference on Engineering of Complex Computer Systems (ICECCS)*, 2005. URL: https://doi.org/10.1109/ICECCS.2005.68 (cited on pages 25, ii).

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008. ISBN: 978-0-262-02649-9 (cited on page 24).

[BK11] Björn Bartels and Moritz Kleine. A CSP-based framework for the specification, verification, and implementation of adaptive systems. In *2011 ICSE Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS*, 2011. URL: https://doi.org/10.1145/1988008.1988030 (cited on pages 55, 59, 61, 63, 64).

[BKF+17] Cornel Barna, Hamzeh Khazaei, Marios Fokaefs, and Marin Litoiu. Delivering elastic containerized cloud applications to enable devops. In *12th IEEE/ACM International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS*, 2017. URL: https://doi.org/10.1109/SEAMS.2017.12 (cited on page 4).

[BLC+18] Erwan Bousse, Dorian Leroy, Benoît Combemale, Manuel Wimmer, and Benoit Baudry. Omniscient debugging for executable dsls. *Journal of Systems and Software*, 2018. URL: https://doi.org/10.1016/j.jss.2017.11.025 (cited on pages 25, 29).

[BLW05] Paul Baker, Shiou Loh, and Frank Weil. Model-driven engineering in a large industrial context - motorola case study. In *Model Driven Engineering Languages and Systems, MoDELS*, 2005. URL: https://doi.org/10.1007/11557432%5C_36 (cited on page 24).

[BMB+18] Manuel F. Bertoa, Nathalie Moreno, Gala Barquero, Loli Burgueño, Javier Troya, and Antonio Vallecillo. Expressing measurement uncertainty in OCL/UML datatypes. In *Modelling Foundations and Applications, ECMFA*, 2018. URL: https://doi.org/10.1007/978-3-319-92997-2_4 (cited on pages 66–69, 72, 73, 84).

[BMM14] James Bornholt, Todd Mytkowicz, and Kathryn S. McKinley. Uncertain: a first-order type for uncertain data. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS*, 2014. URL: https://doi.org/10.1145/2541940.2541958 (cited on pages 5, 6, 66, 67, 69, 70, 72, 73, 80).

[Bor13] James Bornholt. *Abstractions and techniques for programming with uncertain data*. PhD thesis, Honors thesis, Australian National University, 2013. URL: https://homes.cs.washington.edu/~bornholt/papers/thesis13.pdf (cited on pages 33, 69, 80).

[BPG+04] Paolo Bresciani, Anna Perini, Paolo Giorgini, Fausto Giunchiglia, and John Mylopoulos. Tropos: an agent-oriented software development methodology. *Autonomous Agents and Multi-Agent Systems*, 2004. URL: https://doi.org/10.1023/B:AGNT.0000018806.20944.ef (cited on page 129).

[BPS10] Luciano Baresi, Liliana Pasquale, and Paola Spoletini. Fuzzy goals for requirements-driven adaptation. In *RE, 18th IEEE International Requirements Engineering Conference*, 2010. URL: https://doi.org/10.1109/RE.2010.25 (cited on pages 18, 55, 56, 59, 60, 63, 65).

[BSG+09] Yuriy Brun, Giovanna Di Marzo Serugendo, Cristina Gacek, Holger Giese, Holger M. Kienle, Marin Litoiu, Hausi A. Müller, Mauro Pezzè, and Mary Shaw. Engineering self-adaptive systems through feedback loops. In *Software Engineering for Self-Adaptive Systems [outcome of a Dagstuhl Seminar]*, 2009. URL: https://doi.org/10.1007/978-3-642-02161-9%5C_3 (cited on pages 16, 18).

[BSH+06] Omar Benjelloun, Anish Das Sarma, Alon Y. Halevy, and Jennifer Widom. ULDBs: databases with uncertainty and lineage. In *Proceedings of the 32nd International Conference on Very Large Data Bases*, 2006. URL: http://dl.acm.org/citation.cfm?id=1164209 (cited on pages 66, 67, 69, 71).

[BWS+12] Nelly Bencomo, Kristopher Welsh, Pete Sawyer, and Jon Whittle. Self-explanation in adaptive systems. In *17th IEEE International Conference on Engineering of Complex Computer Systems, ICECCS*, 2012. URL: http://doi.ieeecomputersociety.org/10.1109/ICECCS.2012.34 (cited on pages 9, 59, 60).

[CA07] Betty H. C. Cheng and Joanne M. Atlee. Research directions in requirements engineering. In *Workshop on the Future of Software Engineering, FOSE*, 2007. URL: https://doi.org/10.1109/FOSE.2007.17 (cited on page 22).

[CCH+13] Maxime Cordy, Andreas Classen, Patrick Heymans, Axel Legay, and Pierre-Yves Schobbens. Model checking adaptive software with featured transition systems. In *Assurances for Self-Adaptive Systems - Principles, Models, and Techniques*. 2013. URL: https://doi.org/10.1007/978-3-642-36249-1_1 (cited on pages 59, 61, 63, 65).

[CDE+02] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Jose F. Quesada. Maude: specification and programming in rewriting logic, 2002. URL: https://doi.org/10.1016/S0304-3975(01)00359-0 (cited on page 56).

[CdLG+09] Betty H. C. Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, Jeff Magee, Jesper Andersson, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Giovanna Di Marzo Serugendo, Schahram Dustdar, Anthony Finkelstein, Cristina Gacek, Kurt Geihs, Vincenzo Grassi, Gabor Karsai, Holger M. Kienle, Jeff Kramer, Marin Litoiu, Sam Malek, Raffaela Mirandola, Hausi A. Müller, Sooyong Park, Mary Shaw, Matthias Tichy, Massimo Tivoli, Danny Weyns, and Jon Whittle. Software engineering for self-adaptive systems: a research roadmap. In *Software Engineering for Self-Adaptive Systems [outcome of a Dagstuhl Seminar]*, 2009. URL: https://doi.org/10.1007/978-3-642-02161-9_1 (cited on pages 4, 16, i).

[CDM09] Anis Charfi, Tom Dinkelaker, and Mira Mezini. A plug-in architecture for self-adaptive web service compositions. In *IEEE International Conference on Web Services, ICWS*, 2009. URL: https://doi.org/10.1109/ICWS.2009.125 (cited on pages 59, 60, 63, 64).

[CG12] Shang-Wen Cheng and David Garlan. Stitch: a language for architecture-based self-adaptation. *Journal of Systems and Software*, 2012. URL:

https://doi.org/10.1016/j.jss.2012.02.060 (cited on pages 59, 60, 62, 63, 130).

[CGF+09] Carlos Cetina, Pau Giner, Joan Fons, and Vicente Pelechano. Autonomic computing through reuse of variability models at runtime: the case of smart homes. *IEEE Computer*, 42(10):37–43, 2009. DOI: 10.1109/MC.2009.309. URL: https://doi.org/10.1109/MC.2009.309 (cited on page 19).

[CGH+17] Bob Carpenter, Andrew Gelman, Matthew D. Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. Stan : a probabilistic programming language. *Journal of Statistical Software*, 2017. ISSN: 1548-7660. URL: https://doi.org/10.18637/jss.v076.i01 (cited on pages 66, 67, 69, 70, 72, 73).

[CGK+] Radu Calinescu, Lars Grunske, Marta Z. Kwiatkowska, Raffaela Mirandola, and Giordano Tamburrelli. Dynamic QoS management and optimization in service-based systems. *IEEE Transactions on Software Engineering*. URL: https://doi.org/10.1109/TSE.2010.92 (cited on page 18).

[Che76] Peter P. Chen. The entity-relationship model - toward a unified view of data. *ACM Transactions on Database Systems*, 1976. URL: https://doi.org/10.1145/320434.320440 (cited on page 23).

[CMR13] Michael Carbin, Sasa Misailovic, and Martin C. Rinard. Verifying quantitative reliability for programs that execute on unreliable hardware. In *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA*, 2013. URL: https://doi.org/10.1145/2509136.2509546 (cited on pages 67–69, 71).

[CMR15] Alessandro Cimatti, Andrea Micheli, and Marco Roveri. Strong temporal planning with uncontrollable durations: a state-space approach.

In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2015. URL: http://www.aaai.org/ocs/index.php/AAAI/AAAI15/paper/view/9728 (cited on pages 59, 61).

[CNR13] Arun Tejasvi Chaganty, Aditya V. Nori, and Sriram K. Rajamani. Efficiently sampling probabilistic programs via program analysis. In *Proceedings of the Sixteenth International Conference on Artificial Intelligence and Statistics, AISTATS*, 2013. URL: http://proceedings.mlr.press/v31/chaganty13a.html (cited on pages 66, 67, 69, 70, 72, 73).

[Com+06] Autonomic Computing et al. An architectural blueprint for autonomic computing. *IBM White Paper*, 31, 2006. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.150.1011&rep=rep1&type=pdf (cited on pages 4, 16–19).

[CPY+14] Bihuan Chen, Xin Peng, Yijun Yu, Bashar Nuseibeh, and Wenyun Zhao. Self-adaptation through incremental generative model transformations at runtime. In *36th International Conference on Software Engineering, ICSE*, 2014. URL: https://doi.org/10.1145/2568225.2568310 (cited on pages 55, 56, 59, 60, 62, 63).

[CvL17] Antoine Cailliau and Axel van Lamsweerde. Runtime monitoring and resolution of probabilistic obstacles to system goals. In *12th IEEE/ACM International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS*, 2017. URL: https://doi.org/10.1109/SEAMS.2017.5 (cited on pages 55, 56).

[DBZ14] Brahim Djoudi, Chafia Bouanaka, and Nadia Zeghib. Model checking pervasive context-aware systems. In *2014 IEEE 23rd International WETICE Conference, WETICE*, 2014. URL: https://doi.org/10.1109/WETICE.2014.11 (cited on pages 55, 57, 59, 63, 65).

[Deg16] Thomas Degueule. *Composition and Interoperability for External Domain-Specific Language Engineering*. PhD thesis, University of Rennes

1, France, 2016. URL: https://tel.archives-ouvertes.fr/tel-01427009 (cited on page 82).

[Dem98] M Beth L Dempster. *A self-organizing systems perspective on planning for sustainability*. PhD thesis, Citeseer, 1998. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.180.6090 (cited on page 17).

[Dey01] Anind K. Dey. Understanding and using context. *Personal and Ubiquitous Computing*, 2001. URL: https://doi.org/10.1007/s007790170019 (cited on pages 20, i).

[dL04] Mark d'Inverno and Michael Luck. *Understanding agent systems, Second Edition*. Springer series on agent technology. 2004. ISBN: 978-3-540-40700-3 (cited on page 55).

[DL06] Pierre-Charles David and Thomas Ledoux. An aspect-oriented approach for developing self-adaptive fractal components. In *Software Composition, SC*, 2006. URL: https://doi.org/10.1007/11821946_6 (cited on pages 55, 57, 59, 60, 63–65).

[dLGM+10] Rogério de Lemos, Holger Giese, Hausi A. Müller, Mary Shaw, Jesper Andersson, Marin Litoiu, Bradley R. Schmerl, Gabriel Tamura, Norha M. Villegas, Thomas Vogel, Danny Weyns, Luciano Baresi, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Ronald J. Desmarais, Schahram Dustdar, Gregor Engels, Kurt Geihs, Karl M. Göschka, Alessandra Gorla, Vincenzo Grassi, Paola Inverardi, Gabor Karsai, Jeff Kramer, Antónia Lopes, Jeff Magee, Sam Malek, Serge Mankovski, Raffaela Mirandola, John Mylopoulos, Oscar Nierstrasz, Mauro Pezzè, Christian Prehofer, Wilhelm Schäfer, Richard D. Schlichting, Dennis B. Smith, João Pedro Sousa, Ladan Tahvildari, Kenny Wong, and Jochen Wuttke. Software engineering for self-adaptive systems: a second research roadmap. In *Software Engineering for Self-Adaptive Systems II - International Seminar, Dagstuhl Castle*, 2010.

URL: https://doi.org/10.1007/978-3-642-35813-5_1 (cited on page 21).

[DMS18] Lucio Mauro Duarte, Paulo Henrique Mendes Maia, and Ana Carolina Sanchotene Silva. Extraction of probabilistic behaviour models based on contexts. In *Proceedings of the 10th International Workshop on Modelling in Software Engineering, MiSE*, 2018. URL: https://doi.org/10.1145/3193954.3193963 (cited on pages 55, 57).

[DSB+17] Gwendal Daniel, Gerson Sunyé, Amine Benelallam, Massimo Tisi, Yoann Vernageau, Abel Gómez, and Jordi Cabot. NeoEMF: a multi-database model persistence framework for very large models. *Science of Computer Programming*, 2017. URL: https://doi.org/10.1016/j.scico.2017.08.002 (cited on page 25).

[DSC16] Gwendal Daniel, Gerson Sunyé, and Jordi Cabot. Mogwaï: a framework to handle complex queries on large models. In *IEEE International Conference on Research Challenges in Information Science, RCIS*, 2016. URL: https://doi.org/10.1109/RCIS.2016.7549343 (cited on page 25).

[DvLF93] Anne Dardenne, Axel van Lamsweerde, and Stephen Fickas. Goal-directed requirements acquisition. *Science of Computer Programming*, 1993. URL: https://doi.org/10.1016/0167-6423(93)90021-G (cited on page 129).

[EB10] Moritz Eysholdt and Heiko Behrens. Xtext: implement your language faster than the quick and dirty way. In *Companion to the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, SPLASH/OOPSLA*, 2010. URL: https://doi.org/10.1145/1869542.1869625 (cited on page 30).

[Egy01] Alexander Egyed. A scenario-driven approach to traceability. In *Proceedings of the 23rd International Conference on Software Engineering,*

ICSE 2001, 2001. URL: https://doi.org/10.1109/ICSE.2001.919087 (cited on page 129).

[EM10] Naeem Esfahani and Sam Malek. Uncertainty in self-adaptive software systems. In *Software Engineering for Self-Adaptive Systems II - International Seminar, Dagstuhl Castle*, 2010. URL: https://doi.org/10.1007/978-3-642-35813-5_9 (cited on pages 67, 68).

[EPR14] Romina Eramo, Alfonso Pierantonio, and Gianni Rosa. Uncertainty in bidirectional transformations. In *6th International Workshop on Modeling in Software Engineering, MiSE 2014*, 2014. URL: https://doi.org/10.1145/2593770.2593772 (cited on pages 66, 67, 69, 71).

[EPR15] Romina Eramo, Alfonso Pierantonio, and Gianni Rosa. Managing uncertainty in bidirectional model transformations. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering, SLE*, 2015. URL: https://dl.acm.org/citation.cfm?id=2814259 (cited on pages 66, 67, 69, 71).

[FA04] Paolo Falcarin and Gustavo Alonso. Software architecture evolution through dynamic AOP. In *Software Architecture, First European Workshop, EWSA*, 2004. DOI: 10.1007/978-3-540-24769-2_5 (cited on pages 63, 64).

[Far10] Hassan Farhangi. The path of the smart grid. *IEEE power and energy magazine*, 2010. URL: https://doi.org/10.1109/MPE.2009.934876 (cited on page 4).

[FC19] Michalis Famelis and Marsha Chechik. Managing design-time uncertainty. *Software and Systems Modeling*, 2019. URL: https://doi.org/10.1007/s10270-017-0594-9 (cited on pages 66, 67, 69, 70).

[FGL+10] Jean-Marie Favre, Dragan Gasevic, Ralf Lämmel, and Ekaterina Pek. Empirical language analysis in software linguistics. In *Software Lan-*

guage Engineering, SLE, 2010. URL: https://doi.org/10.1007/978-3-642-19440-5%5C_21 (cited on page 28).

[FGL+11] Antonio Filieri, Carlo Ghezzi, Alberto Leva, and Martina Maggio. Self-adaptive software meets control theory: a preliminary approach supporting reliability requirements. In *26th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2011. URL: https://doi.org/10.1109/ASE.2011.6100064 (cited on pages 55, 57, 59, 63, 65).

[FHM14] Antonio Filieri, Henry Hoffmann, and Martina Maggio. Automated design of self-adaptive software with control-theoretical formal guarantees. In *36th International Conference on Software Engineering, ICSE*, 2014. URL: https://doi.org/10.1145/2568225.2568272 (cited on pages 17, 18).

[FMF+12] François Fouquet, Brice Morin, Franck Fleurey, Olivier Barais, Noël Plouzeau, and Jean-Marc Jézéquel. A dynamic component model for cyber physical systems. In *Proceedings of the 15th ACM SIGSOFT Symposium on Component Based Software Engineering, CBSE*, 2012. URL: https://doi.org/10.1145/2304736.2304759 (cited on pages 4, 55, 57, 62–64).

[FMX+12] Xi Fang, Satyajayant Misra, Guoliang Xue, and Dejun Yang. Smart grid - the new and improved power grid: a survey. *IEEE Communications Surveys and Tutorials*, 2012. URL: https://doi.org/10.1109/SURV.2011.101911.00087 (cited on page 4).

[FNM+12] François Fouquet, Grégory Nain, Brice Morin, Erwan Daubert, Olivier Barais, Noël Plouzeau, and Jean-Marc Jézéquel. An eclipse modelling framework alternative to meet the models@runtime requirements. In *Model Driven Engineering Languages and Systems, MODELS*, 2012. URL: https://doi.org/10.1007/978-3-642-33666-9_7 (cited on page 27).

[FNM+14] François Fouquet, Grégory Nain, Brice Morin, Erwan Daubert, Olivier Barais, Noël Plouzeau, and Jean-Marc Jézéquel. Kevoree modeling framework (KMF): efficient modeling techniques for runtime use. *CoRR*, 2014. arXiv: 1405.6817. URL: http://arxiv.org/abs/1405.6817 (cited on page 27).

[Fou10] Eclipse Foundation. Ecore. https://wiki.eclipse.org/Ecore, 2010. [Accessed July 2019] (cited on page 27).

[Fow10] Martin Fowler. *Domain-specific languages*. Pearson Education, 2010. ISBN: 978-0321712943. URL: https://www.martinfowler.com/books/dsl.html (cited on page 42).

[FR07] Robert B. France and Bernhard Rumpe. Model-driven development of complex software: a research roadmap. In *International Conference on Software Engineering, ISCE*, 2007. URL: https://doi.org/10.1109/FOSE.2007.14 (cited on pages 23, 26).

[FSC12] Michalis Famelis, Rick Salay, and Marsha Chechik. Partial models: towards modeling and reasoning with uncertainty. In *34th International Conference on Software Engineering, ICSE*, 2012. URL: https://doi.org/10.1109/ICSE.2012.6227159 (cited on pages 66, 67, 69, 70).

[fSta16] International Organization for Standardization (ISO). Structured query language (SQL). https://www.iso.org/standard/63555.html, 2016. [Accessed July 2019] (cited on page 24).

[Gar08] Vahid Garousi. Traffic-aware stress testing of distributed real-time systems based on UML models in the presence of time uncertainty. In *First International Conference on Software Testing, Verification, and Validation, ICST*, 2008. URL: https://doi.org/10.1109/ICST.2008.7 (cited on pages 67, 69, 71).

[GB] Phil Greenwood and Lynne Blair. A framework for policy driven auto-adaptive systems using dynamic framed aspects. URL: https://doi.org/10.1007/11922827_2 (cited on pages 63, 64).

[GBM+13] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. Internet of things (IoT): a vision, architectural elements, and future directions. *Future Generation Computer Systems*, 2013. URL: https://doi.org/10.1016/j.future.2013.01.010 (cited on page 27).

[GBM+18] Alejandro Sánchez Guinea, Andrey Boytsov, Ludovic Mouline, and Yves Le Traon. Continuous identification in smart environments using wrist-worn inertial sensors. In *Proceedings of the 15th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services, MobiQuitous*, 2018. URL: https://doi.org/10.1145/3286978.3287001 (cited on page viii).

[GCH+04] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley R. Schmerl, and Peter Steenkiste. Rainbow: architecture-based self-adaptation with reusable infrastructure. *IEEE Computer*, 2004. URL: https://doi.org/10.1109/MC.2004.175 (cited on pages 17, 18, 55, 57–59, 62, 63).

[GHN+14] Andrew D. Gordon, Thomas A. Henzinger, Aditya V. Nori, and Sriram K. Rajamani. Probabilistic programming. In *Proceedings of the on Future of Software Engineering, FOSE*, 2014. URL: https://doi.org/10.1145/2593882.2593900 (cited on pages 7, 69, 70).

[GHP+08] Paul Grace, Danny Hughes, Barry Porter, Gordon S. Blair, Geoff Coulson, and François Taïani. Experiences with open overlays: a middleware approach to network heterogeneity. In *Proceedings of the 2008 EuroSys Conference*, 2008. URL: https://doi.org/10.1145/1352592.1352606 (cited on pages 58, 59, 62, 63).

[Gli07] Martin Glinz. On non-functional requirements. In *15th IEEE International Requirements Engineering Conference, RE*, 2007. URL: https://doi.org/10.1109/RE.2007.45 (cited on page 22).

[GPS+13] Carlo Ghezzi, Leandro Sales Pinto, Paola Spoletini, and Giordano Tamburrelli. Managing non-functional uncertainty via model-driven adaptivity. In *35th International Conference on Software Engineering, ICSE*, 2013. URL: https://doi.org/10.1109/ICSE.2013.6606549 (cited on pages 55, 56, 59, 63, 65).

[Gro14] Object Management Group. Object constraint language (OCL), version 2.4. http://www.omg.org/spec/OCL/, 2014. [Accessed July 2019] (cited on page 26).

[Gro15] Object Management Group. XML metadata interchange (XMI), version 2.5.1. http://www.omg.org/spec/XMI/, 2015. [Accessed July 2019] (cited on page 26).

[Gro16a] Object Management Group. Meta object facility (MOF), version 2.5.1. http://www.omg.org/spec/MOF/, 2016. [Accessed July 2019] (cited on pages 26, 27).

[Gro16b] Object Management Group. Query/view/transformation (OCL), version 1.3. https://www.omg.org/spec/QVT, 2016. [Accessed July 2019] (cited on page 60).

[Gru95] Thomas R. Gruber. Toward principles for the design of ontologies used for knowledge sharing? *International Journal of Human-Computer Studies*, 1995. URL: https://doi.org/10.1006/ijhc.1995.1081 (cited on page 23).

[GS10] Carlo Ghezzi and Amir Molzam Sharifloo. Dealing with non-functional requirements for adaptive systems via dynamic software product-lines. In *Software Engineering for Self-Adaptive Systems II - International Seminar, Dagstuhl Castle*, 2010. URL: https://doi.org/10.1007/978-3-642-35813-5_8 (cited on pages 55, 57, 59, 61, 63, 65).

[GS64] Israel M. Gelfand and Georgi E. Shilov. *Generalized Functions, Volume 1, Properties and Operations*. American Mathematical Society, 1964. ISBN: 978-1-4704-2658-3 (cited on page 91).

[GVD19] Simos Gerasimou, Thomas Vogel, and Ada Diaconescu. Software engineering for intelligent and autonomous systems: report from the GI dagstuhl seminar 18343. *CoRR*, 2019. URL: http://arxiv.org/abs/1904.01518 (cited on page 4).

[GvdHT09] John C. Georgas, André van der Hoek, and Richard N. Taylor. Using architectural models to manage and visualize runtime adaptation. *IEEE Computer*, 2009. URL: https://doi.org/10.1109/MC.2009.335 (cited on pages 55, 57, 59, 61–63).

[Hal06] B. D. Hall. Component interfaces that support measurement uncertainty. *Computer Standards & Interfaces*, 2006. URL: https://doi.org/10.1016/j.csi.2005.07.009 (cited on pages 66, 67, 69, 71, 72).

[Har16] Thomas Hartmann. *Enabling Model-Driven Live Analytics For Cyber-Physical Systems: The Case of Smart Grids*. PhD thesis, University of Luxembourg, 2016. URL: http://orbilu.uni.lu/handle/10993/28924 (cited on pages 11, 28, 136).

[HBB15] Sara Hassan, Nelly Bencomo, and Rami Bahsoon. Minimizing nasty surprises with better informed decision-making in self-adaptive systems. In *10th IEEE/ACM International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS*, 2015. URL: https://doi.org/10.1109/SEAMS.2015.13 (cited on pages 8, 11).

[HFJ+17] Thomas Hartmann, François Fouquet, Matthieu Jimenez, Romain Rouvoy, and Yves Le Traon. Analyzing complex data in motion at scale with temporal graphs. In *The 29th International Conference on Software Engineering and Knowledge Engineering*, 2017. URL: https://doi.org/10.18293/SEKE2017-048 (cited on page 136).

[HFK+14a] Thomas Hartmann, François Fouquet, Jacques Klein, Grégory Nain, and Yves Le Traon. Reactive security for smart grids using models@run.time-based simulation and reasoning. In *Smart Grid Security - Second International Workshop, SmartGridSec*, 2014. URL: https://doi.org/

10.1007/978-3-319-10329-7_9 (cited on pages 4, 5, 37, 49, 55, 56, 58, 59, 63, 64).

[HFK+14b] Thomas Hartmann, François Fouquet, Jacques Klein, Yves Le Traon, Alexander Pelov, Laurent Toutain, and Tanguy Ropitault. Generating realistic smart grid communication topologies based on real-data. In *2014 IEEE International Conference on Smart Grid Communications, SmartGridComm*, 2014. URL: https://doi.org/10.1109/SmartGridComm.2014.7007684 (cited on pages 4, 36, 141).

[HFM+16] Nicolas Harrand, Franck Fleurey, Brice Morin, and Knut Eilif Husa. ThingML: a language and code generation framework for heterogeneous targets. In *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, 2016. URL: http://dl.acm.org/citation.cfm?id=2976812 (cited on page 24).

[HFM+19] Thomas Hartmann, François Fouquet, Assaad Moawad, Romain Rouvoy, and Yves Le Traon. GreyCat: efficient what-if analytics for data in motion at scale. *Information Systems*, 2019. URL: https://doi.org/10.1016/j.is.2019.03.004 (cited on pages 28, 58).

[HFN+14a] Thomas Hartmann, François Fouquet, Grégory Nain, Brice Morin, Jacques Klein, Olivier Barais, and Yves Le Traon. A native versioning concept to support historized models at runtime. In *Model-Driven Engineering Languages and Systems, MODELS*, 2014. URL: https://doi.org/10.1007/978-3-319-11653-2_16 (cited on pages 55, 56, 58, 62, 63).

[HFN+14b] Thomas Hartmann, François Fouquet, Grégory Nain, Brice Morin, Jacques Klein, and Yves Le Traon. Reasoning at runtime using time-distorted contexts: a models@run.time based approach. In *The 26th International Conference on Software Engineering and Knowledge Engineering*, 2014. URL: http://ksiresearchorg.ipage.com/seke/Proceedings/seke/SEKE2014_Proceedings.pdf (cited on pages 10, 20, 28, 55, 56, 58, 62, 63).

[HHP+08] Svein O. Hallsteinsen, Mike Hinchey, Sooyong Park, and Klaus Schmid. Dynamic software product lines. *IEEE Computer*, 2008. URL: https://doi.org/10.1109/MC.2008.123 (cited on page 19).

[HIR02] Karen Henricksen, Jadwiga Indulska, and Andry Rakotonirainy. Modeling context information in pervasive computing systems. In *Pervasive Computing, First International Conference, Pervasive*, 2002. URL: https://doi.org/10.1007/3-540-45866-2_14 (cited on pages 20, 21, 55, 56, 58).

[HMF+16] Thomas Hartmann, Assaad Moawad, François Fouquet, Yves Reckinger, Jacques Klein, and Yves Le Traon. Near real-time electric load approximation in low voltage cables of smart grids with models@run.time. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, 2016. URL: https://doi.org/10.1145/2851613.2853125 (cited on pages 36, 37).

[HMF+19] Thomas Hartmann, Assaad Moawad, François Fouquet, and Yves Le Traon. The next evolution of MDE: a seamless integration of machine learning into domain modeling. *Software and System Modeling*, 2019. URL: https://doi.org/10.1007/s10270-017-0600-2 (cited on page 28).

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 1978. URL: https://doi.org/10.1145/359576.359585 (cited on page 55).

[HR04] David Harel and Bernhard Rumpe. Meaningful modeling: what's the semantics of "semantics"? *IEEE Computer*, 2004. URL: https://doi.org/10.1109/MC.2004.172 (cited on pages 29, 82).

[HRW11] John Edward Hutchinson, Mark Rouncefield, and Jon Whittle. Model-driven engineering practices in industry. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE*, 2011. URL: https://doi.org/10.1145/1985793.1985882 (cited on page 24).

[HWR+11] John Edward Hutchinson, Jon Whittle, Mark Rouncefield, and Steinar Kristoffersen. Empirical assessment of MDE in industry. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE*, 2011. URL: https://doi.org/10.1145/1985793.1985858 (cited on page 24).

[IA09] Ali Ipakchi and Farrokh Albuyeh. Grid of the future. *IEEE power and energy magazine*, 2009. URL: https://doi.org/10.1109/MPE.2008.931384 (cited on page 4).

[III17] ISO, IEC, and IEEE. Systems and software engineering – vocabulary. In *ISO/IEC/IEEE 24765: 2017 (E)*. 2017. URL: https://doi.org/10.1109/IEEESTD.2017.8016712 (cited on pages 21, 22, i, ii).

[IW14] M. Usman Iftikhar and Danny Weyns. ActivFORMS: active formal models for self-adaptation. In *9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS*, 2014. URL: https://doi.org/10.1145/2593929.2593944 (cited on pages 55, 56, 59, 63, 64).

[Jac97] Michael Jackson. The meaning of requirements. *Annals of Software Engineering*, 1997. URL: https://doi.org/10.1023/A:1018990005598 (cited on page 17).

[JCB+15] Jean-Marc Jézéquel, Benoît Combemale, Olivier Barais, Martin Monperrus, and François Fouquet. Mashup of metalanguages and its implementation in the kermeta language workbench. *Software and Systems Modeling*, 2015. URL: https://doi.org/10.1007/s10270-013-0354-4 (cited on page 103).

[JG17] Seyyed Ahmad Javadi and Anshul Gandhi. DIAL: reducing tail latencies for cloud applications via dynamic interference-aware load balancing. In *2017 IEEE International Conference on Autonomic Computing, ICAC*, 2017. URL: https://doi.org/10.1109/ICAC.2017.17 (cited on page 4).

[JK12] Szymon Jaroszewicz and Marcin Korzen. Arithmetic operations on independent random variables: a numerical approach. *SIAM J. Scientific Computing*, 2012. URL: https://doi.org/10.1137/110839680 (cited on pages 66, 67, 69, 70, 72, 73).

[JWB+15] Andrés Jiménez-Ramírez, Barbara Weber, Irene Barba, and Carmelo Del Valle. Generating optimized configurable business process models in scenarios subject to uncertainty. *Information & Software Technology*, 2015. URL: https://doi.org/10.1016/j.infsof.2014.06.006 (cited on pages 66, 67, 69, 71).

[KC03] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *IEEE Computer*, 2003. URL: https://doi.org/10.1109/MC.2003.1160055 (cited on pages 4, 16–19, ii, iii).

[Kel76] Robert M. Keller. Formal verification of parallel programs. *Communications of ACM*, 1976. URL: https://doi.org/10.1145/360248.360251 (cited on page 57).

[Ken02] Stuart Kent. Model driven engineering. In *Integrated Formal Methods, Third International Conference, IFM*, 2002. URL: https://doi.org/10.1007/3-540-47884-1_16 (cited on pages 4, 23, 24).

[KH10] Maximilian Koegel and Jonas Helming. EMFStore: a model repository for EMF models. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, ICSE*, 2010. URL: https://doi.org/10.1145/1810295.1810364 (cited on page 10).

[Kle08] Anneke Kleppe. *Software language engineering: creating domain-specific languages using metamodels*. 2008. ISBN: 0-321-60646-9. URL: https://www.pearson.com/us/higher-education/program/Kleppe-Software-Language-Engineering-Creating-Domain-Specific-Languages-Using-Metamodels/PGM162096.html (cited on pages 28, iii).

[KM90] Jeff Kramer and Jeff Magee. The evolving philosophers problem: dynamic change management. *IEEE Transactions on Software Engineering*, 1990. URL: https://doi.org/10.1109/32.60317 (cited on pages 17, 55, 57, 59, 60, 63, 64).

[Kra07] Jeff Kramer. Is abstraction the key to computing? *Communications of the ACM*, 2007. URL: https://doi.org/10.1145/1232743.1232745 (cited on page 23).

[KT12] Bilal Kanso and Safouan Taha. Temporal constraint support for OCL. In *Software Language Engineering, SLE*, 2012. URL: https://doi.org/10.1007/978-3-642-36089-3_6 (cited on page 10).

[LEB18] Eric O. LEBIGOT. Uncertainties: a python package for calculations with uncertainties. http://pythonhosted.org/uncertainties/, 2018. Accessed: 2018-10-11 (cited on pages 41, 42).

[Lew03] Bil Lewis. Debugging backwards in time. *CoRR*, cs.SE/0310016, 2003. URL: http://arxiv.org/abs/cs.SE/0310016 (cited on page 25).

[LFK+14] Heiner Lasi, Peter Fettke, Hans-Georg Kemper, Thomas Feld, and Michael Hoffmann. Industry 4.0. *Business & Information Systems Engineering*, 2014. URL: https://doi.org/10.1007/s12599-014-0334-4 (cited on page 80).

[LGC17] Philippe Lalanda, Eva Gerbert-Gaillard, and Stéphanie Chollet. Self-aware context in smart home pervasive platforms. In *2017 IEEE International Conference on Autonomic Computing, ICAC*, 2017. URL: https://doi.org/10.1109/ICAC.2017.1 (cited on page 4).

[LTB+00] David J. Lunn, Andrew Thomas, Nicky Best, and David J. Spiegelhalter. WinBUGS - a bayesian modelling framework: concepts, structure, and extensibility. *Statistics and Computing*, 2000. URL: https://doi.org/10.1023/A:1008929526011 (cited on pages 66, 67, 69, 70, 72, 73).

[LWZ+17] Gaoqi Liang, Steven R Weller, Junhua Zhao, Fengji Luo, and Zhao Yang Dong. The 2015 ukraine blackout: implications for false data injection attacks. *IEEE Transactions on Power Systems*, 2017. URL: https://doi.org/10.1109/TPWRS.2016.2631891 (cited on page 80).

[Mao09] Shahar Maoz. Using model-based traces as runtime models. *IEEE Computer*, 2009. URL: https://doi.org/10.1109/MC.2009.336 (cited on pages 55, 57).

[MAR14] Danilo F. Mendonça, Raian Ali, and Genaína Nunes Rodrigues. Modelling and analysing contextual failures for dependability requirements. In *9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS*, 2014. URL: https://doi.org/10.1145/2593929.2593947 (cited on pages 55, 56, 59, 60).

[MBE+11] Michael Maurer, Ivan Breskovic, Vincent C. Emeakaroha, and Ivona Brandic. Revealing the MAPE loop for the autonomic management of cloud infrastructures. In *Proceedings of the 16th IEEE Symposium on Computers and Communications, ISCC*, 2011. URL: https://doi.org/10.1109/ISCC.2011.5984008 (cited on pages 63, 65).

[MBF+18] Ludovic Mouline, Amine Benelallam, François Fouquet, Johann Bourcier, and Olivier Barais. A temporal model for interactive diagnosis of adaptive systems. In *2018 IEEE International Conference on Autonomic Computing, ICAC*, 2018. URL: https://doi.org/10.1109/ICAC.2018.00029 (cited on pages 14, vii).

[MBH+18] Ludovic Mouline, Amine Benelallam, Thomas Hartmann, François Fouquet, Johann Bourcier, Brice Morin, and Olivier Barais. Enabling temporal-aware contexts for adaptative distributed systems. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing, SAC*, 2018. URL: https://doi.org/10.1145/3167132.3167286 (cited on pages 14, vii).

[MBH+ng] Ludovic Mouline, Amine Benelallam, Thomas Hartmann, Johann Bourcier, Olivier Barais, and Maxime Cordy. Ain'tea: managing data uncertainty at the language level. *Forthcoming*, forthcoming (cited on pages 13, vii).

[MBJ+09] Brice Morin, Olivier Barais, Jean-Marc Jézéquel, Franck Fleurey, and Arnor Solberg. Models@run.time to support dynamic adaptation. *IEEE Computer*, 2009. URL: https://doi.org/10.1109/MC.2009.327 (cited on pages 4, 5, 19, 48, 54, 55, 62, 63).

[MBN+09] Brice Morin, Olivier Barais, Grégory Nain, and Jean-Marc Jézéquel. Taming dynamically adaptive systems using models and aspects. In *31st International Conference on Software Engineering, ICSE*, 2009. URL: https://doi.org/10.1109/ICSE.2009.5070514 (cited on pages 63, 64).

[MCG+15] Gabriel A. Moreno, Javier Cámara, David Garlan, and Bradley R. Schmerl. Proactive self-adaptation under uncertainty: a probabilistic model checking approach. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE*, 2015. URL: https://doi.org/10.1145/2786805.2786853 (cited on pages 55, 57, 59, 63, 65).

[MD04] Erik Meijer and Peter Drayton. Static typing where possible, dynamic typing when needed: the end of the cold war between programming languages, 2004 (cited on page 70).

[Met08] JCfGi Metrology. Evaluation of measurement data - guide to the expression of uncertainty in measurement. *Bureau International des Poids et Mesures*, 2008. URL: https://www.bipm.org/en/publications/guides/gum.html (cited on pages 5, 6, 11, 33, 69, 80–82, 152).

[MGB63] Alexander M. Mood, Franklin A. Graybill, and Duane C. Boes. *Introduction to the Theory of Statistics*. McGraw Hill; 3rd edition, 1963. ISBN: 978-0070854659 (cited on page 92).

[MHF+15] Assaad Moawad, Thomas Hartmann, François Fouquet, Grégory Nain, Jacques Klein, and Yves Le Traon. Beyond discrete modeling: a continuous and efficient model for iot. In *18th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MoDELS*, 2015. URL: https://doi.org/10.1109/MODELS.2015.7338239 (cited on page 28).

[MHF+17] Ludovic Mouline, Thomas Hartmann, François Fouquet, Yves Le Traon, Johann Bourcier, and Olivier Barais. Weaving rules into models@run.time for embedded smart systems. In *Companion to the first International Conference on the Art, Science and Engineering of Programming, Programming*, 2017. URL: https://doi.org/10.1145/3079368.3079394 (cited on page viii).

[Mos01] Peter D. Mosses. The varieties of programming language semantics. In *Perspectives of System Informatics, PSI*, 2001. URL: https://doi.org/10.1007/3-540-45575-2_18 (cited on page 83).

[MPS15] Marina Mongiello, Patrizio Pelliccione, and Massimo Sciancalepore. AC-Contract: run-time verification of context-aware applications. In *10th IEEE/ACM International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS*, 2015. URL: https://doi.org/10.1109/SEAMS.2015.11 (cited on pages 8, 11).

[MS17] Vera Zaychik Moffitt and Julia Stoyanovich. Temporal graph algebra. In *Proceedings of The 16th International Symposium on Database Programming Languages, DBPL*, 2017. URL: https://doi.org/10.1145/3122831.3122838 (cited on pages 55, 57, 58, 63, 64).

[MSS13] Prodromos Makris, Dimitrios N. Skoutas, and Charalabos Skianis. A survey on context-aware mobile and wireless networking: on networking and computing environments' integration. *IEEE Communications Surveys and Tutorials*, 2013. URL: https://doi.org/10.1109/SURV.2012.040912.00180 (cited on pages 20, 21).

[MWG+18] T. Minka, J.M. Winn, J.P. Guiver, Y. Zaykov, D. Fabian, and J. Bronskill. Infer.net 0.3, 2018. URL: https://dotnet.github.io/infer/. [Accessed July 2019] (cited on pages 7, 42, 81, 102).

[MWV16] Tanja Mayerhofer, Manuel Wimmer, and Antonio Vallecillo. Adding uncertainty and units to quantity types in software models. In *International Conference on Software Language Engineering*, 2016. URL: http://dl.acm.org/citation.cfm?id=2997376 (cited on pages 66–69, 72, 73, 112).

[OMG17] Object Management Group (OMG). OMG Unified Modeling Language, version 2.5.1, 2017. URL: https://www.omg.org/spec/UML/2.5.1. [Accessed July 2019] (cited on pages 23, 26).

[oSN] National Institute of Standards and Technology (NIST). Smart grid: a beginner's guide. https://www.nist.gov/engineering-laboratory/[redacted]smart-grid/smart-grid-beginners-guide#what. [Accessed June 2019] (cited on page iii).

[PBC+11] Carlos Parra, Xavier Blanc, Anthony Cleve, and Laurence Duchien. Unifying design and runtime software adaptation using aspect models. *Science of Computer Programming*, 2011. URL: https://doi.org/10.1016/j.scico.2010.12.005 (cited on pages 59, 60, 63, 64).

[Pfe01] Avi Pfeffer. IBAL: a probabilistic rational programming language. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence, IJCAI*, 2001 (cited on pages 66, 67, 69, 70, 72, 73).

[PFT03] Mónica Pinto, Lidia Fuentes, and José M. Troya. DAOP-ADL: an architecture description language for dynamic component and aspect-based development. In Frank Pfenning and Yannis Smaragdakis, editors, *Generative Programming and Component Engineering, Second International Conference, GPCE*, 2003. URL: https://doi.org/10.1007/978-3-540-39815-8_8 (cited on pages 58, 59, 63, 64).

[Plu+03] Martyn Plummer et al. Jags: a program for analysis of bayesian graphical models using gibbs sampling. In *Proceedings of the 3rd international workshop on distributed statistical computing*, 2003 (cited on pages 66, 67, 69, 70, 72, 73).

[PPT08] Sungwoo Park, Frank Pfenning, and Sebastian Thrun. A probabilistic language based on sampling functions. *ACM Transactions on Programming Languages and Systems*, 2008. URL: https://doi.org/10.1145/1452044.1452048 (cited on pages 66, 67, 69, 70, 72, 73).

[PSR10] Dhirendra Pandey, Ugrasen Suman, and AK Ramani. An effective requirement engineering process model for software development and requirements management. In *Advances in Recent Technologies in Communication and Computing (ARTCom), 2010*, 2010. URL: https://doi.org/10.1109/ARTCom.2010.24 (cited on page 22).

[PZC+14] Charith Perera, Arkady B. Zaslavsky, Peter Christen, and Dimitrios Georgakopoulos. Context aware computing for the internet of things: a survey. *IEEE Communications Surveys and Tutorials*, 2014. URL: https://doi.org/10.1109/SURV.2013.042313.00197 (cited on page 20).

[RP02] Norman Ramsey and Avi Pfeffer. Stochastic lambda calculus and monads of probability distributions. In *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2002. URL: https://doi.org/10.1145/503272.503288 (cited on pages 66, 67, 69, 70, 72, 73).

[RRV08] José Eduardo Rivera, José Raúl Romero, and Antonio Vallecillo. Behavior, time and viewpoint consistency: three challenges for MDE. In *Models in Software Engineering, Workshops and Symposia at MODELS*, 2008. URL: https://doi.org/10.1007/978-3-642-01648-6_7 (cited on page 10).

[SBM+08] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: eclipse modeling framework, 2nd Edition*. Addison-Wesley Professional, 2008. ISBN: 0-321-33188-5. URL: http://www.informit.com/store/emf-eclipse-modeling-framework-9780321331885 (cited on page 27).

[SCG13] Sriram Sankaranarayanan, Aleksandar Chakarov, and Sumit Gulwani. Static analysis for probabilistic programs: inferring whole program properties from finitely many paths. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*, 2013. URL: https://doi.org/10.1145/2491956.2462179 (cited on pages 66, 67, 69, 70, 72, 73).

[SCH+13] Rick Salay, Marsha Chechik, Jennifer Horkoff, and Alessio Di Sandro. Managing requirements uncertainty with partial models. *Requirements Engineering*, 2013. URL: https://doi.org/10.1007/s00766-013-0170-y (cited on pages 66, 67, 69, 71).

[Sch06] Douglas C. Schmidt. Guest editor's introduction: model-driven engineering. *IEEE Computer*, 2006. URL: https://doi.org/10.1109/MC.2006.58 (cited on pages 4, 23–25, ii).

[Sha76] Glenn Shafer. *A mathematical theory of evidence*, volume 42. Princeton university press, 1976. ISBN: 9780691100425 (cited on pages 7, 11, 81, 152).

[SMH11] Julia Schwarz, Jennifer Mankoff, and Scott E. Hudson. Monte carlo methods for managing interactive state, action and feedback under uncertainty. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*, 2011. URL: https://doi.org/10.1145/2047196.2047227 (cited on pages 66, 67, 69, 72, 73).

[ST09] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: landscape and research challenges. *TAAS*, 2009. URL: https://doi.org/10.1145/1516533.1516538 (cited on pages 17, 18).

[SWF16] John Salvatier, Thomas V. Wiecki, and Christopher Fonnesbeck. Probabilistic programming in python using PyMC3. *PeerJ Computer Science*, 2016. URL: https://doi.org/10.7717/peerj-cs.55 (cited on pages 42, 66, 67, 69, 70, 72, 73).

[Tea15] Watcher Drivers Team. Openstack watcher. https://wiki.openstack.org/wiki/Watcher, 2015. [Accessed April 2019] (cited on pages 4, 114).

[TGE+10] Hossein Tajalli, Joshua Garcia, George Edwards, and Nenad Medvidovic. PLASMA: a plan-based layered architecture for software model-driven adaptation. In *25th IEEE/ACM International Conference on Automated Software Engineering*, 2010. URL: https://doi.org/10.1145/1858996.1859092 (cited on pages 55, 57, 59, 63, 65).

[Thr00] Sebastian Thrun. Towards programming tools for robots that integrate probabilistic computation and learning. In *Proceedings of the 2000 IEEE International Conference on Robotics and Automation, ICRA*, 2000. URL: https://doi.org/10.1109/ROBOT.2000.844075 (cited on pages 66, 67, 69, 70, 72, 73).

[TOH17] Yasuyuki Tahara, Akihiko Ohsuga, and Shinichi Honiden. Formal verification of dynamic evolution processes of UML models using aspects. In *12th IEEE/ACM International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS*, 2017. URL: https://doi.org/10.1109/SEAMS.2017.4 (cited on pages 9, 12, 55–59, 62, 63).

[vDK98] Arie van Deursen and Paul Klint. Little languages: little maintenance? *Journal of Software Maintenance*, 1998. URL: https://doi.org/10.1002/(SICI)1096-908X(199803/04)10:2%3C75::AID-SMR168%3E3.0.CO;2-5 (cited on page 29).

[vDKV00] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: an annotated bibliography. *SIGPLAN Notices*, 2000. URL: https://doi.org/10.1145/352029.352035 (cited on pages 29, i).

[VMO16] Antonio Vallecillo, Carmen Morcillo, and Priscill Orue. Expressing measurement uncertainty in software models. In *10th International Conference on the Quality of Information and Communications Technology, QUATIC*, 2016. URL: http://doi.ieeecomputersociety.org/10.1109/QUATIC.2016.013 (cited on pages 11, 66–69, 72–74, 81).

[Voe14] Markus Voelter. *Generic tools, specific languages*. Citeseer, 2014. ISBN: 978-94-6203-586-7. URL: https://www.voelter.de/data/books/GenericToolsSpecificLanguages-1.0-web.pdf (cited on page 29).

[W3C05] World Wide Web Consortium (W3C). Document object model (DOM). https://www.w3.org/DOM/, 2005. [Accessed July 2019] (cited on page 29).

[WA13] Danny Weyns and Tanvir Ahmad. Claims and evidence for architecture-based self-adaptation: a systematic literature review. In *Software Architecture ECSA*, 2013. URL: https://doi.org/10.1007/978-3-642-39031-9%5C_22 (cited on page 18).

[Wal96] Christian Walck. Hand-book on statistical distributions for experimentalists. Technical report, University of Stockholm, 1996 (cited on pages 84, 91).

[WBR11] Richard Wang, Dana Butnariu, and Jennifer Rexford. Openflow-based server load balancing gone wild. In *USENIX Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services, Hot-ICE*, 2011. URL: https://www.usenix.org/conference/hot-ice11/openflow-based-server-load-balancing-gone-wild (cited on pages 45, 46).

[Wey19] Danny Weyns. Software engineering of self-adaptive systems. In *Handbook of Software Engineering*. 2019. URL: https://doi.org/10.1007/978-3-030-00262-6%5C_11 (cited on pages 16, 17).

[WH04] Tom De Wolf and Tom Holvoet. Emergence versus self-organisation: different concepts but promising when combined. In *Engineering Self-Organising Systems, Methodologies and Applications*, 2004. URL: https://doi.org/10.1007/11494676_1 (cited on page 17).

[WHH10] Danny Weyns, Robrecht Haesevoets, and Alexander Helleboogh. The MACODO organization model for context-driven dynamic agent organizations. *TAAS*, 2010. URL: https://doi.org/10.1145/1867713.1867717 (cited on pages 55, 59, 61, 63, 64).

[WHR] Jon Whittle, John Edward Hutchinson, and Mark Rouncefield. The state of practice in model-driven engineering. *IEEE Software*. URL: https://doi.org/10.1109/MS.2013.65 (cited on page 23).

[WHR+13] Jon Whittle, John Edward Hutchinson, Mark Rouncefield, Håkan Burden, and Rogardt Heldal. Industrial adoption of model-driven engineering: are the tools really the problem? In *Model-Driven Engineering Languages and Systems, MODELS*, 2013. URL: https://doi.org/10.1007/978-3-642-41533-3%5C_1 (cited on page 24).

[WKE08] Gerd Wübbeler, Michael Krystek, and Clemens Elster. Evaluation of measurement uncertainty and its numerical calculation by a monte carlo method. *Measurement science and technology*, 2008. URL: https://doi.org/10.1088/0957-0233/19/8/084009 (cited on page 152).

[WMA12] Danny Weyns, Sam Malek, and Jesper Andersson. FORMS: unifying reference model for formal specification of distributed self-adaptive systems. *TAAS*, 2012. URL: https://doi.org/10.1145/2168260.2168268 (cited on pages 17, 18, 55, 63, 64).

[Woh14] Claes Wohlin. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *18th International Conference on Evaluation and Assessment in Software Engineering, EASE*, 2014. URL: https://doi.org/10.1145/2601248.2601268 (cited on pages 53, 74).

[WSB+09] Jon Whittle, Peter Sawyer, Nelly Bencomo, Betty H. C. Cheng, and Jean-Michel Bruel. RELAX: incorporating uncertainty into the specification of self-adaptive systems. In *International Requirements Engineering Conference*, 2009. URL: https://doi.org/10.1109/RE.2009.36 (cited on pages 67, 69, 71).

[WSB+10] Jon Whittle, Peter Sawyer, Nelly Bencomo, Betty H. C. Cheng, and Jean-Michel Bruel. RELAX: a language to address uncertainty in self-adaptive systems requirement. *Requirements Engineering*, 2010. URL: https://doi.org/10.1007/s00766-010-0101-0 (cited on pages 67, 69, 71).

[Yu11] Eric Yu. Modelling strategic relationships for process reengineering. *Social Modeling for Requirements Engineering*, 2011. URL: http://ftp.cs.utoronto.ca/public_html/dist/eric/DKBS-TR-94-6.pdf (cited on page 129).

[ZA11] Enrico Zio and Terje Aven. Uncertainties in smart grids behavior and modeling: what are the risks and vulnerabilities? how to analyze them? *Energy Policy*, 2011. URL: https://doi.org/10.1016/j.enpol.2011.07.030 (cited on page 5).

[Zad] Lotfi A Zadeh. Fuzzy sets. In *Fuzzy Sets, Fuzzy Logic, And Fuzzy Systems: Selected Papers by Lotfi A Zadeh*. URL: https://doi.org/10.1142/9789814261302_0001 (cited on pages 11, 81).

[ZAY+19] Man Zhang, Shaukat Ali, Tao Yue, Roland Norgren, and Oscar Okariz. Uncertainty-wise cyber-physical system test modeling. *Software and Systems Modeling*, 2019. URL: https://doi.org/10.1007/s10270-017-0609-6 (cited on pages 66–69, 71).

[ZGC09] Ji Zhang, Heather Goldsby, and Betty H. C. Cheng. Modular verification of dynamically adaptive systems. In *Proceedings of the 8th International Conference on Aspect-Oriented Software Development,*

AOSD, 2009. URL: https://doi.org/10.1145/1509239.1509262 (cited on pages 55, 57, 59, 63, 65).

[ZSA+16] Man Zhang, Bran Selic, Shaukat Ali, Tao Yue, Oscar Okariz, and Roland Norgren. Understanding uncertainty in cyber-physical systems: a conceptual model. In *Modelling Foundations and Applications, ECMFA*, 2016. URL: https://doi.org/10.1007/978-3-319-42061-5_16 (cited on pages 66–69, 71, 72).