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 impact(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 requirement. In this thesis, we propose a metamodel*[2] to represent the decisions made by the adaptation process over time (*cf.* Chapter 5). Plus, we define a language, Ain'tea, to propagate uncertainty in the computation made during the adaptation process (*cf.* Chapter 4).

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 complexity the understanding. For example, a climatologist models all elements that impact the global climate (wind, ocean current, temperature, ...), ignoring local information, like the temperature under a forest, or global ones, like the solar system. In contrast, astronomers model the solar system, ignoring all information regarding the Earth climate.

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

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 [WHR14]. 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 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

[3]The Semantic Web is defined as en extension of the Web to enable the processing by machines.
[4]The definition of concern is intentionally left undefined as it is domain-specific.

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 new methods such as NeoEMF [BGS+14; DSB+17], which enable the storage of large models*, and Mogwaï [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 meta-

[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].

model* 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 models* of models*. In the MDE* community, they are generally defined using the class diagram of the UML* specification [OMG17]. 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. 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, which we detail in the next section.

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

2.2.3 Models-@run.-time*

At its origins, MDE* mainly addresses the challenges of designing, validating, and maintaining complex software systems. However, these systems were facing a new challenge: the uncertainty of the environment in which they are deployed. As we detail in section 2.1, systems have been improved to enable runtime adaptation. This process can only be done if the adaptation process has a deep understanding of the structure, the behaviour, and the environment of the system. This deep understanding can be provided by a proper abstraction of these runtime elements. Extending MDE* principles and visions, the models-@run.-time* paradigm spans the use of models* from design time to runtime. The model, as an abstraction of a real system, can be used during runtime to reason about the state of the actual system. A conceptual link between the model and the real system allows modifying the actual system through the model and vice versa. The models@run.time paradigm uses metamodels* to define the domain concepts of a real system, together with its surrounding environment. Consequently, the runtime model depicts an abstract and yet rich representation of the system context that conforms to (is an instance of) its meta-model.

2.2.4 Tooling

Tooling is an 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 (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.

[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."

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

[10] `https://greycat.ai/`

2.2.5 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 5).

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 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

[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.

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].

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 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

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

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 4).

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

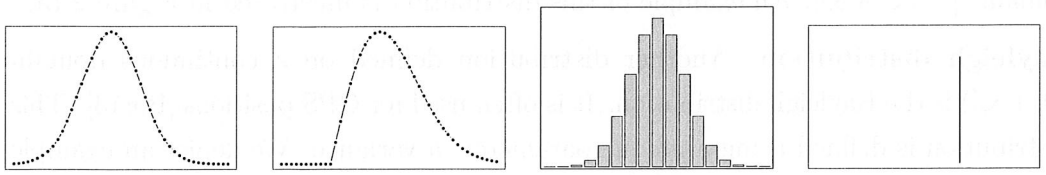
A probability distribution of a random variable X is a function that gives the probability that X takes on the values x, for all possible values of the sample space. Here, we can distinguish two cases: discrete and continuous distributions. A distribution is said discrete when it is based on a discrete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



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

Figure 2.1: Probability distributions used in this thesis

oba:examples)?

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

It has been proven that this distribution is similar to a Gaussian distribution in certain cases [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 2.1a.

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 2.1b.

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

3

State of the art

?⟨chapt:sota⟩?

Contents

This chapter reviews 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.*

3.1 Introduction

In this thesis, we focus on the capacity of adaptive systems* to adjust their behaviour* or struc-ture* in response to changes in their environment* or struc-ture* (*cf.* Chapter 1). Following the models-@run.-time* paradigm, these systems can use a modelling layer that abstracts all information needed to enable reasoning processes (*cf.* Chapter 2). This process leads to the execution of a set of adaptation actions*, among which we identified long-term action* (*cf.* Chapter 1).

However, we have identified three open challenges due to long-term action*, the uncertainty of data received, and the emergence of the behaviour* of these systems (*cf.* Section 1.4). In the past years, researchers heavily studied adaptive systems* and uncertainty. We thus need to identify any potential work that has already tackled, partially or totally, these challenges. Besides, we have to review approaches that model adaptive systems*, struc-ture*, behaviour*, environment*, or manipulate uncertain data.

We set two research questions to drive this review process:

- **RQ1:** Do state-of-the-art solutions that model adaptive system* allow representing and reasoning over long-term actions* (design time and runtime[1])?
- **RQ2:** Do state-of-the-art solutions allow modelling uncertainty of data and its manipulation (propagation, reasoning over)?

Through this review, we found that none of the state-of-the-art solutions model actions*, their effects, and their circumstances* over time (RQ1), both at design time and at runtime. Some approaches will partially model these elements, for example, actions* and their effects at design time. However, these models cannot be automatically navigated by a process, like a reasoning engine. Moreover, some approaches do not consider the temporal dimension of the effects.

Our findings show that different community have studied uncertainty in software. For instance, modelling community has focused on design uncertainty and adaptive system* community has studied environment* uncertainty. Researchers have put a lot of efforts to encapsulate probability distributions behind programming language

[1]Here we refer to the execution of an action*.

concepts such as variables. But we strongly think that higher-level abstraction should be defined to help engineers. Additionally, tools to help developers to manipulate uncertain data are needed.

The remainder of this chapter is structured as follows. First, we describe the methodology used to perform this review in Section 3.2. In Section 3.3 and Section 3.4, we present and discuss our findings. Before concluding in Section 3.6, we detail some threats to validity of this literature review.

3.2 Review methodology

sota:methodo)?

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 (struc-ture* 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*, struc-ture*, 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*, struc-ture*, 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*, struc-ture*, 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 1.5). 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 struc-ture*, 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 [2]. Results have also been exported to Comma-separated values (CSV)* files for those who cannot open Excel files.

3.3 Results RQ1: long-term actions*

ults: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*, struc-ture*, 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.

[2] 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 3.1: Approaches to model systems' context and behaviour* (RQ1.1)

sults:actions:rq1.1⟩?

3.3.1 Modelling the evolution of system's context*, struc-ture*, or behaviour*

Different categories of approaches exist to represent the context*, struc-ture*, or the behaviour* of a system. In this section, we detail our findings with an overview given in Table 3.1.

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. It can contain information about either the context* of the system, its behaviour*, or its struc-ture*. 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[3]. 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

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

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 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