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 struc-ture* of systems (*cf.* Table 3.1). 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.

| 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 3.2: Approaches to model actions*, their circumstances*, and their effects (RQ1.2)

sults:actions:rq1.2⟩?

### 3.3.2 Modelling actions*, their circumstances*, and their effects

In Table 3.2, 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 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[4] [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.

**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

[4]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.

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 3.2, 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.

| 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 | [GB06; 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 3.3: Approaches to reason over evolving context or behaviour* (RQ1.3)

sults:actions:rq1.3⟩?

### 3.3.3 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 3.3 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 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 [GB06; 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[5]. However, none of them 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.

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

**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 3.3. 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*.

### 3.3.4 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 struc-ture*, 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.

| 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 3.4: Categories of uncertainty addressed by the literature (RQ2.1)

a:results:duc:rq2.1⟩?

## 3.4 Results RQ2: data uncertainty*

ec:sota:results:duc⟩? 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.

### 3.4.1 Categories of data uncertainty

The literature provides different approaches to tackle challenges that come with different aspects of uncertainty. In Table 3.4, 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 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, Jiménez-Ramírez *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 3.3). 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 3.4. 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.

### 3.4.2 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 3.5.

**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 the mean equals the value (here 19.1) and the variance the *standard uncertainty* (here 0.1). Barbará *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

| 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 3.5: Approaches to model data uncertainty* (RQ2.2)

ts:duc:rq2.2)?

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

| 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 3.6: Approaches to propagate data uncertainty* (RQ2.3)

results:duc:rq2.3.1⟩?

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 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 3.5, 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.

### 3.4.3 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

| 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 3.7: Approaches to reason over the uncertainty of data (RQ2.3)

:duc:rq2.3.2⟩?

reason over uncertainty. Table 3.6 and Table 3.7 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 3.6. 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 3.7). Indeed, state-of-the-art solutions allow developers to access either the values stored or the properties of a probability distribution.

### 3.4.4 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.

## 3.5 Threat to validity

ota: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.

## 3.6 Conclusion

a: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 5.

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 4).