2020 IEEE/ACM 15th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)

Towards Classes of Architectural Dependability Assurance for Machine-Learning-Based Systems

Max Scheerer
Jonas Klamroth
scheerer@fzi.de
klamroth@fzi.de
FZI Research Center for Information Technology
Karlsruhe, Germany

Ralf Reussner
reussner@kit.edu
Karlsruhe Institute of Technology
Karlsruhe, Germany
FZI Research Center for Information Technology
Karlsruhe, Germany

Bernhard Beckert
beckert@kit.edu
Karlsruhe Institute of Technology
Karlsruhe, Germany

ABSTRACT

Advances in Machine Learning (ML) have brought previously hard to handle problems within arm's reach. However, this power comes at the cost of unassured reliability and lacking transparency. Overcoming this drawback is very hard due to the probabilistic nature of ML. Current approaches mainly tackle this problem by developing more robust learning procedures. Such algorithmic approaches, however, are limited to certain types of uncertainties and cannot deal with all of them, e.g., hardware failure. This paper discusses how this problem can be addressed at architectural rather than algorithmic level to assess systems dependability properties in early development stages. Moreover, we argue that Self-Adaptive Systems (SAS) are more suited to safeguard ML w.r.t. various uncertainties. As a step towards this we propose classes of dependability in which ML-based systems may be categorized and discuss which and how assurances can be made for each class.

CCS CONCEPTS

• **Software and its engineering → Software architectures**; **Formal methods**; • **Computing methodologies → Machine learning**.

KEYWORDS

dependability, artificial intelligence, machine learning, architectural-driven self-adaptation, software quality

ACM Reference Format:
Max Scheerer, Jonas Klamroth, Ralf Reussner, and Bernhard Beckert. 2020. Towards Classes of Architectural Dependability Assurance for Machine-Learning-Based Systems. In *IEEE/ACM 15th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS '20), October 7–8, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3387939.3388613

SEAMS '20, October 7–8, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7962-5/20/05…$15.00
https://doi.org/10.1145/3387939.3388613

1 INTRODUCTION

Machine Learning (ML) has attracted a lot of attention in recent years. More specifically, advances in *Deep Learning* (DL), a subfield of ML, opened a lot of new possibilities for various application scenarios. For instance, there has been successful work on the application of DL in autonomous driving [6], robotic manipulation [13], or smart manufacturing [34]. The application of ML in complex learning scenarios, however, comes at the cost of model complexity and, thus, reduced interpretability [15] and transparency [14]. For instance, adversarial examples exploit small variations of the input space along the decision boundary to achieve misclassification, while appearing unmodified to human observers [25]. This is a severe problem for safety-critical systems, as a misclassification may result in physical or economical damage.

There are several approaches that aim to improve quality attributes like safety [33], robustness [38] or reliability [14] at an algorithmic level. However, many classical approaches to ensure the correctness of systems, like deductive verification and model checking, are barely applicable to ML due to its inherently probabilistic and non-linear nature.

Seshia et al. [29] pointed out that one possibility to prove the correctness of Artificial Intelligence (AI) based systems is to specify the whole system instead of just the AI-component. Extending this idea, we argue that shifting the problem of addressing quality assurance to the architectural level, relieves the ML-algorithms of the burden to satisfy quality attributes at algorithmic level, and allows them to focus on risk minimization only, and is in conformance with the *Separation of Concern* principle. For example, instead of algorithmically addressing security aspects of deep neural networks, architectural security patterns [27] can be employed to eliminate security breaches and keep the system secure in the first place.

One of the main problems of ML-based software is that the operating environment does not always behave as the training phase suggests, which can lead to incorrect predictions. Therefore, we advocate the use of Self-adaptive Systems (SAS), as their primary goal is to adapt the structure or behaviour of the system to deal with changing environmental conditions. We consider SAS as a safeguard for ML algorithms that is mainly used to maintain dependability properties over time, as envisioned by de Lemos and Grzes [8]. In the literature, there are several model-based approaches that abstract the environment and software architecture of the system to analyse specific properties of the SAS at design-time (e.g. [3, 26, 36]). One of the biggest advantages of model-based approaches is that

system properties can be analysed before a single line of code is even written.

In this paper, we address dependability properties of ML-based systems from a model-based architectural perspective. Since there is no "test oracle" that determines whether or not a prediction of an ML algorithm is correct, it is inherently hard to test ML-based software [24]. Therefore, we propose four classes of architectural dependability in which any ML-based system can be classified according to various criteria. Each class assesses the system and the operating environment in terms of the degree of abstraction through models, the analytical potential, and the assurances that can be given at different stages of development. These classes and their implications can be considered by software architects as guidelines to engineer self-adaptive ML-based software.

Our contributions are as follows:

- We present four classes of dependability assurances for ML-based systems.
- We show how these classes may be used to make assurances at an architectural level.

The remainder of this paper is organized as follows: Section 2 gives a brief overview of some basic concepts used in the following sections. In Section 3, we introduce the notion of *Analytic Capacity* which is a measure of analysability. Section 4 presents our classes of architectural dependability assurances and the criteria for classification. Finally, Section 5 reviews related work before we conclude in Section 6.

2 PRELIMINARIES

2.1 Interpretability and Explainability in ML

In this section, we briefly discuss the notions of *interpretability* and *explainability* in ML. This section is inspired by the definitions presented by Guidotti et al. in [15].

To *interpret* means to "explain or present some concept in [human] understandable terms" (Merriam-Webster online-dictionary, accessed 2020/1/14). This definition, however, strongly relies on the meaning of "to explain", but as Doshi-Velez and Kim point out: in most cases, the definition of *explanation* remains elusive [9]. Guidotti assumes that explanations are at least self-contained and may be understood as an interface between a human and the decision-maker (e.g., an ML-based system). He also mentions rules, linear models, and decision trees as three classes of human-understandable models (given they do not exceed some degree of complexity). However, he also points out that "understandable" is no absolute term but can be highly subjective depending on factors like available time and expertise.

In this paper, we will assume that "human-understandable" and thus "explainable" is some abstract predicate that may be further specified depending on the context of an application (e.g., a decision tree with a maximum depth of 10).

In his survey on the interpretability of black boxes, Guidotti presents several different kinds of interpretability, of which we are mainly interested in two: model-explainability and outcome-explainability. A system is model-explainable if it is possible to generate an explanation which fully describes its behavior. In contrast to that, outcome explainability captures the idea that, despite not being able to explain the system's behavior, one may be able to explain a certain output. So a system is outcome-explainable if one can generate an explanation for a given output of said system.

As an example, consider a cancer detection application. Given some medical records, this system decides whether the patient has cancer or not. If this system is model explainable, we are able to provide an explanation of how the system's decision process works. For example, the system always decides that the patient has cancer, whenever tumors have been detected. This is a very simple decision procedure, which is easily model-explainable. On the other hand, the model is outcome explainable iff after the system has made its verdict it is able to generate an explanation for its decision.

2.2 Probabilistic Dynamics of Self-Adaptive Systems

In this section, we formally describe the dynamic behaviour of an SAS. We generalize the problem to be addressed to conduct model-based design-time analysis and the engineering challenge to be solved. More precisely, we consider an SAS as a *Markov Decision Process* (MDP). MDPs are prevalent mathematical models in the SAS community to describe the probabilistic nature of SAS [2, 23]. An MDP $(S, A, \mathcal{T}_a(s_i, s_j), \mathcal{R}_a(s_i, s_j))$ comprises a set S of states, a set A of actions, a function $\mathcal{T}_a(s_i, s_j)$ which evaluates to the probability of transitioning to the next state s_j given the current state s_i and the action a, and a function $\mathcal{R}_a(s_i, s_j)$ returning a real number, called reward, that can be interpreted as a quality measure of selecting action a in state s_i and transitioning to state s_j. The goal is to find a policy $\pi : S \rightarrow A$ determining the action to take at any state that maximizes the sum of rewards produced over time. In the context of SAS, we consider the engineering problem of developing a reconfiguration or adaptation strategy π so that the quality objectives of the system are satisfied over time; captured by $\mathcal{R}_a(s_i, s_j)$.

We consider an MDP of an SAS as a process induced by two components, namely the environmental dynamics and the reconfiguration process. The environmental dynamics refer to a set $E := \{\varepsilon_1, \varepsilon_2, \ldots\}$ of discrete states. More specifically, we consider all variables in the environment that potentially influence quality objectives as part of the environmental state, e.g., harsh weather conditions or hardware failures. We define the environmental dynamics as a stochastic process and as the primary component responsible for the non-deterministic behaviour of SAS over time. As in many other scientific works (e.g. [2, 5, 23]), we describe the environmental dynamics as a *Discrete-Time Markov Chain* (DTMC) (E, p_{ij}, E_0) which comprises the set E of environmental states, a transition matrix p_{ij} determining the probability to transition to state ε_j given state ε_i, and a set $E_0 \subseteq E$ of initial environmental states. Note, that DTMCs and MDPs underlie the *Markov Assumption*; that is, the probability of a transition to state ε_j from ε_i depends solely on ε_i.

The second component is the reconfiguration process, which highly depends on the environmental dynamics as the process is only triggered when the environment transitions to states that cannot be handled by the current system configuration. Since we address the adaptation problem at the architectural level, we denote the set of system configurations as *Architectural Configuration Space* $\Gamma := \{\gamma_1, \gamma_2, \ldots\}$. The reconfiguration process can be formally described by the deterministic function $\phi : \Gamma \times \Delta \rightarrow \Gamma$

which maps an architectural configuration γ_i to an architectural configuration γ_j w.r.t. a set $\Delta := \{\delta_1, \delta_2, \ldots\}$ of reconfigurations. By definition, Δ contains the element $\delta_\emptyset$ that is to be understood as identity reconfiguration with the property $\forall \gamma \in \Gamma : \phi(\delta_\emptyset, \gamma) = \gamma$. Note, that since the environmental dynamics and the reconfiguration process constitute the MDP, the state space is defined as follows: $S := E \times \Gamma$.

3 ANALYTIC CAPACITY OF ML SOFTWARE

In this section, we introduce the notion of analytic capacity, which can be considered as metric of analysability of ML-based software.

3.1 Analysability

In the following, we present four classes of analysability of systems. These are "verifiable", "fully monitorable", "partially monitorable" and "non-monitorable". Note, that the notion of monitorability in our context is different from what mostly is understood by that term in the literature on formal methods. Explainability and monitorability are closely related. Given an explanation, it is possible to check whether this explanation conforms to some formal constraints or not [9]. Thus, having an explainer is often sufficient to build a monitor. Consequently, the classes introduced in here are inspired by the classes of explainability presented in Section 2.

Before we dive into the definitions of our classes, we would like to emphasize that these classes are somewhat subjective in the sense that different persons/organizations may categorize the same system differently. This is intended, and we will go into detail as to why that is. These classes are meant to be useful for analysis at design-time. Classifying a system is thus to be seen as a design-time decision made by the system architect. We illustrate how such a decision at design-time, in turn, influences the assurances that can be made about the system.

We define all classes w.r.t. a given system property that we denote by φ. We assume φ to be objectively observable.

Definition 1. *A system is* verifiable *w.r.t. φ iff it is possible to prove with* justifiable *effort that it (always) satisfies φ.*

In the following, when not referring to a specific property φ, we call a system verifiable if it satisfies its full specification. Additionally, we intentionally do not further specify when the verification effort is "justifiable" as that heavily depends on the context in which a system is developed. Consider for example the following list of factors:

1) Experience A developer experienced in program verification will most likely be able to prove the desired property much faster than someone completely new to the field.
2) Time Since program verification is a very time-consuming effort, verification may not be possible due to time constraints.
3) Computational power Several methods for program verification rely on some kind of solvers which can be very computationally expensive.
4) Type of system Some types of software, like drivers, are easier to verify than complex ones such as, e.g., neural networks.
5) Field of application Depending on the field where the system is deployed, different safety/security requirements may apply.

The above list is by no means complete. It just serves to show that "justifiable" depends on many aspects that may also influence each other. Note, however, that despite the elusiveness of "justifiable", these factors only influence the effort made to verify a system. The result, namely that the system is indeed verified, remains the same once a proof is found. Another perspective is to consider "verifiable" as a parameterized class where the amount of effort is the parameter which may vary depending on the factors mentioned above.

Motivated by the realisation that not every system is verifiable, we define a second class of reliability:

Definition 2. *A system is* fully monitorable *iff it is feasible to implement a decision procedure which decides in reasonable time at runtime whether the component's current behaviour satisfies φ or not.*

Intuitively, this requires a monitor that is able to tell at any point in time whether our system is still working correctly or not. This differs from the first class in two main points: First, we allow the system to violate the property (but we have to be able to detect that) and, second, the performance of the decision procedure plays a vital role. Again we intentionally do not further specify "reasonable time". This is because the definition of "reasonable", as the definition of "justifiable", depends heavily on the context of the application. In some cases, it might be acceptable to realize "something went wrong an hour ago". Oftentimes, however, it is crucial to detect errors relatively fast so as to quickly react to the problem. For many systems, constructing a monitor is more feasible than to full verification. However, there are systems where even constructing a monitor for all possible system states is very hard. Our next class of reliability tries to capture that idea.

Definition 3. *A system is* partially monitorable *iff it is feasible to implement a decision procedure which decides in reasonable time at runtime whether the component's current behavior satisfies φ or not or reaches an inconclusive verdict.*

Here we relax the requirements for the monitor in that we allow it to return an inconclusive verdict. Even though we decided to not require a certain percentage for which the monitor must come to a conclusive verdict, it is reasonable to do so in practice. Note that failing to come to a verdict may either be due to the nature of the property itself or due to the monitor. Also, failing to return a conclusive verdict may or may not be caused by a time-out.

For the sake of completeness, we introduce a last class of reliability which we call "non-monitorable":

Definition 4. *A system is* non-monitorable *iff it is neither partially monitorable nor verifiable.*

This is the worst-case where we are not able to give any assurances for the system, neither at design-time nor at runtime. However, this may still not be a lost cause as testing is still an option. If it is possible to rigorously test the system to ensure some quality criterion, that may, depending on the context, suffice.

The order in which we presented the classes corresponds to a natural total order them in terms of the strength of achievable assurances. While a verifiable component exhibits "perfect" behavior, a fully monitorable system is at least able to recognize whether or not it made a mistake. This ability is limited for partially monitorable systems and is completely lacking for non-monitorable ones.

A system may be verifiable but not monitorable. Consider a system where a huge effort is justifiable to ensure it runs as expected but which has very challenging real-time requirements. In that case, the system would be verifiable but not monitorable.

Note also that the actual dependability of a system and the dependability assurances that can be made in practice may differ. A system may be perfectly dependable, but it may be very hard to prove that. Our classes however prevent the opposite case: If a system is verifiable, strong dependability assurances can be made.

3.2 State-space Complexity

The second factor of the analytic capacity of a system (besides analysability) is state-space complexity which refers to the set of states that have to be analysed. Recall the environmental states E and the architectural configurations Γ from Section 2.2. Consider the utility function $\mathcal{U} : E \times \Gamma \rightarrow \mathbb{R}^n$ that maps a configuration and an environmental state to a real-valued vector, in which each dimension quantifies a specific property. Let us assume that $\mathcal{U}$ is monitorable. At runtime, a reconfiguration strategy is implemented so that a reconfiguration is triggered if $\mathcal{U}(\varepsilon, \gamma) \nvdash q_\varphi$, in which $q_\varphi \in \mathbb{R}^n$ represents a vector of properties that must be satisfied, e.g., quality objectives. The notation $\vdash$ is commonly used in requirements engineering and should be understood as "satisfies" [37]. At design-time, $\mathcal{U}$ has to be estimated to conduct design-time analysis which can be achieved by abstracting E and Γ by environmental and architectural models. We advocate the use of abstraction in terms of models and metamodels to overcome the state-space explosion problem induced by $E \times \Gamma$. Abstraction, however, is at the expense of accuracy and may lead to unrepresentative analysis results. More formally, let $M_E \in \mathcal{I}(MM_E), M_\Gamma \in \mathcal{I}(MM_\Gamma)$ in which M_E denotes a model instance of metamodel MM_E and $\mathcal{I}(MM_E)$ the set of all model instances of MM_E and $M_\Gamma \in \mathcal{I}(MM_\Gamma)$ w.r.t. MM_Γ respectively. To obtain a sufficient approximation of $\mathcal{U}$, we have to find a sufficient abstraction of E and Γ, so that:

$$\forall(\varepsilon, \gamma) \in (E \times \Gamma) : \mathcal{U}'(M_E, M_\Gamma) \sim \mathcal{U}(\varepsilon, \gamma) \tag{1}$$

In non-adaptive systems, software architects have to design an architecture that satisfies the quality objectives w.r.t. the environmental dynamics. In this case, the concept of architectural configurations is known as *Design Space* and the reconfiguration space as *Design Options* [22], assuming: $\exists \gamma^* \in \Gamma : \forall \varepsilon \in E : \mathcal{U}(\gamma^*, \varepsilon) \vdash q_\varphi$. Thus, only the design space must be explored and the state-space complexity is $|S| = |E \times \Gamma|$.

Design space exploration, however, is not sufficient in the context of SAS, since it neglects the temporal aspect. According to Esfahani and Malek [11], the development of SAS is associated with a number of uncertainties. One of these uncertainties is called "Parameter over time" and refers to the temporal aspect of SAS, in which the future behaviour of environmental and system variables has to be taken into account in order to select optimal reconfigurations. Therefore, we argue that the uncertainty "Parameter over time" can be captured by considering the challenge of engineering the reconfiguration strategy π as MDP, as described in Section 2.2. Compared to non-adaptive systems, however, this is a major challenge, since the state-space is associated with a temporal aspect. Thus, it is not sufficient to take into account the system state-space S but rather the *trajectory space* induced by the MDP. The trajectory space contains all possible state paths that must be taken into account to validate whether or not π satisfies the quality objectives. We define the state-space complexity of SAS w.r.t. the trajectory space as follows: $|S_1 \times S_2 \times \cdots \times S_T|$.

In summary, the state-space complexity of SAS corresponds to the cardinality of the trajectory space which is dependent on the degree of abstraction to model E and Γ in order to estimate $\mathcal{U}$.

3.3 Characterization of Analytic Capacity

The analytic capacity is determined by the two factors presented in the last two subsections, namely state-space complexity and analysability of the components. These two factors affect the analytic capacity in that a more complex state-space, as well as a less analysable component, contribute negatively to the analytic capacity of a system; whereas, in contrast, a low complexity of the state-space, as well as a high analysability, lead to a higher analytic capacity. A high complexity of the state-space leads to a decreased analytic capacity in two ways: First, through the increased complexity it becomes harder to observe whether the system at some point satisfies the property it is requested to satisfy. Second, purely statistically it is less probable that the system indeed does satisfy its requested property.

In absence of a better metric, it is also possible to interpret the complexity of the state-space as a metric for the level of abstraction of the used model. A higher complexity correlates with a fine-grained model whereas lower complexity indicates a higher level of abstraction. The level of abstraction is interesting in the sense that all assurances (and thus the categorization into one of the classes) depend on the model used. Consider a system that is considered verifiable. In that case, a proof must exist which shows that the system satisfies its specification. However, this proof is conducted w.r.t. a model which may assume perfect hardware behavior. Errors occurring at hardware level are not modeled and thus not captured by the proof. The level of abstraction directly influences the assurances made for the system.

Furthermore, it is worth noting that the two factors are completely independent of each other. It is theoretically possible to have a very small state-space (even a non-adaptive system) which is still non-monitorable and, dually, it is possible to have a huge state-space for which a system is still fully verifiable. This is, however, a mostly theoretical perspective since most of the time a system is harder to verify/monitor as the state-space it operates on is growing.

4 CLASSIFICATION OF ML SOFTWARE

In this section, we present the preliminary classification scheme based on four classes of dependability and corresponding classification dimensions.

4.1 Classes of Architectural Dependability Assurance

We begin to define the four classes of architectural dependability assurance, each of which has its own implications.

I Static Analysability Static analysability is the ability to make assurances at design-time. More specifically, we say that an ML-based system and its operating environment are statically analysable if

$\mathcal{U}$ is sufficiently approximated w.r.t. formula (1), so that all states can be analysed with an acceptable degree of accuracy.

II Monitor Analysability Monitor analysability refers to the ability to make assurances at design-time. We say that an ML-based system and its operating environment are monitor-analysable if $\mathcal{U}$ is sufficiently approximated w.r.t. formula (1) so that subsets of states can be analysed with an acceptable degree of accuracy. For example, consider [3] as an example of statically analysable systems in the performance domain for SAS and [10] for ML-based but non-adaptive systems.

We distinguish between *Input-* and *Output Monitorability*. Input monitorability refers to a class of algorithms which enable to assess input values, such as variational inference [4], verification of safe input regions [35] or sampled feature spaces validation [14]. What they all have in common is that they try to detect "bad" input data and differ in the way they achieve this. Finally, output monitorability refers to a class of algorithms that assess the outputs w.r.t. to the input data, e.g., by using methods of outcome-explainability like introduced in Section 2.1. We argue that input monitorability is preferable, since it is possible to proactively intervene in situations in which monitored inputs can lead to false predictions. In contrast, output monitorability can only intervene reactively after the prediction was made.

III A-Posteriori Analysability A-posteriori analysability is defined by the inability to conduct design-time analysis. It is assumed that specific properties can only be evaluated at runtime. In case of a system crash, the analysis has to be conducted *a-posteriori* based on logged runtime data of the system. We say that an ML-based system and its operating environment is a-posteriori analysable if $\mathcal{U}$ cannot be sufficiently approximated w.r.t. formula (1).

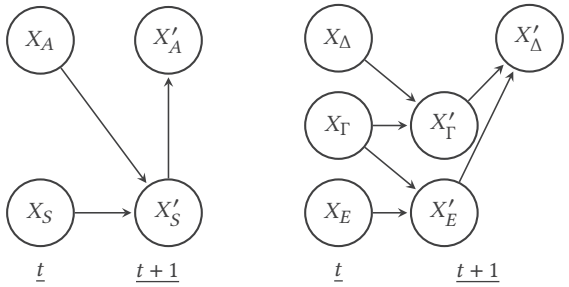
IV Non-Analysability Non-analysability is the worst-case scenario and refers to the inability to analyse a system and its environment at all. We say that an ML-based system and its operating environment is non-analysable if it is not even a-posteriori analysable.

4.2 Classification Dimensions

The first dimension refers to the analytic capacity. The classes are strongly oriented towards the analytic capacity, e.g., systems with high analytic capacity are probably classified to be in class **I**.

The second dimension concerns the question of whether or not there is a fail-safe mode. Intuitively, the possibility to transition to fail-safe at any time contributes in no way to the extend to which the SAS can be analyzed. Nevertheless, even if it is not possible to make strong assurances about certain properties, a fail-safe mode is always an option that can be taken in states which cannot be accurately assessed.

Finally, recall that we consider an SAS as an MDP in which $\mathcal{T}_\delta(s_i, s_j)$ represents an instantiated version of $\mathcal{T}_a(s_i, s_j)$. MDP's generate trajectories through the state space that are induced by $\mathcal{T}_\delta(s_i, s_j)$ and the policy π (see Figure 1a) [31]. As π is a fixed strategy, only $\mathcal{T}_\delta(s_i, s_j)$ is to be estimated and determines whether or not we are able to probabilistically simulate the SAS at design- or runtime. Now, consider the MDP of SAS's as depicted in Figure 1. Without loss of generality, each MDP can be described as a *Dynamic Bayesian Network* (DBN) [32]. For the sake of simplicity, we



(a) DBN of MDP **(b) DBN of MDP in the context of SAS**

$$P(X'_S, X'_A | X_S, X_A) = P(X'_S | X_S, X_A) \cdot P(X'_A | X'_S) \quad \text{(a)}$$

$$= \underbrace{P(X'_\Gamma | X_\Gamma, X_\Delta) \cdot P(X'_E | X_\Gamma, X_E)}_{P(X'_S | X_S, X_A) = \mathcal{T}_\delta(s_i, s_j)} \cdot \underbrace{P(X'_\Delta | X'_\Gamma, X'_E)}_{P(X'_A | X'_S) = \pi} \quad \text{(b)}$$

Figure 1: DBN representation adopted from [32].

omitted the rewards. We use DBNs, as it provides a good representation of how the problem factorizes into various probability distributions that are to be estimated. We note that no detailed DBN knowledge is required; it is sufficient to know that the distribution to transition to the random variables $X'_\Gamma, X'_E, X'_\Delta$ at time instant $t+1$ factorizes to a product of distributions w.r.t. the parents of X_Γ, X_E, X_Δ at t depicted by an arrow [21]. Broadly speaking, it indicates the stochastic evolution of configurations (Γ), reconfigurations (Δ) and environment (E) over time, represented as random variables. Among other things, the distribution decomposes to $P(X'_\Delta | X'_\Gamma, X'_E)$. Note that the policy π is constructed w.r.t. $P(X'_\Delta | X'_\Gamma, X'_E)$ and thus substitutes the distribution. The product of the remaining distributions forms $\mathcal{T}_a(s_i | s_j)$. More precisely, the function is factorized in one part, which describes the reconfiguration process ϕ, and another part, which describes the environmental dynamics. Recall the reconfiguration process described by the function ϕ, as ϕ is deterministic the distribution $P(X'_\Gamma | X_\Gamma, X_\Delta)$ can be substituted by the indicator function $\mathbb{1}_{\phi(\gamma,\delta)=\gamma'}$ with $\gamma, \gamma' \in \Gamma, \delta \in \Delta$ which can be easily estimated. Note that $\mathbb{1}_b$ evaluates to 1 if condition b holds true and 0 otherwise. Thus, only the distribution $P(X'_E | X_\Gamma, X_E)$ is to be estimated to simulate $\mathcal{T}_\delta(s_i | s_j)$. This can be achieved by making assumptions, e.g., X'_E depends solely on the last environmental state X_E so that $P(X'_E | X_\Gamma, X_E) = P(X'_E | X_E)$. $P(X'_E | X_E)$ can now be estimated by probabilistic models, e.g. DBN's [21]. Moreover, $P(X'_E | X_\Gamma, X_E)$ could be replaced by a simulator which simulates the environment, its dynamics and the interaction with the SAS like in [12, 18]. Independently of whether $P(E' | A, E)$ is approximated by mathematical models or simulators, it determines whether or not it is possible to conduct design-time analysis and constitutes another classification dimension.

4.3 Classification Scheme

In this section, we present a preliminary classification scheme; see Table 1. The columns include a dimension not introduced in Section 4.2, namely $\mathbf{F^{-1}}$, which refers to the ability to assess systems input or output values, i.e., $\mathbf{F^{-1}}$ serves to distinguish between input and output monitorability.

We argue that the most critical factor to make assurances at design-time is determined by the approximation of $P(X'_E | X_\Gamma, X_E)$

Table 1: Classification scheme

Class №	Dimensions			
	AC	$\sim \mathrm{P}(\mathrm{X}'_{\mathrm{E}}\|\mathrm{X}_{\Gamma}, \mathrm{X}_{\mathrm{E}})$	**Fail-Safe**	F^{-1}
I	*High*	✓	–	–
IIa	*Medium*	✓	–	✓
IIb	*Medium*	✓	–	✗
III	*High*\/*Medium*	✗	✓/–	–
	Low	–	✓/–	–
IV	*High*\/*Medium*	✗	✗	–
	Low	–	✗	–

✓: Applies, ✗: Does not apply, – : Irrelevant

and the main reason for a classification into classes **I** and **II**. If the latter is given, the next criterion is the analytic capacity of the system and its environment. We categorized the analytic capacity (**AC**) into *High*, *Medium*, and *Low*. It follows that domains with *High* analytic capacity are classified into class **I** and domains with *Medium* analytic capacity into class **II** and so forth.

In terms of class **I**, we argue that whether or not we have a fail-safe mode is irrelevant. Class **I** implies that $\mathcal{U}$ is sufficiently approximated regarding the accuracy of q_φ. Moreover, *High* analytic capacity means that all states are at least fully monitorable according to Definition 2. Therefore, if the state space can be explored in reasonable time and q_φ is sufficiently approximated (e.g. perfect models of system and environment) to conduct verification, then it is possible to prove whether q_φ holds true for each state. In this case, no fail-safe mode is required, as the level of assurance regarding q_φ is provable. Otherwise, the state space can be either too large so that only statistical statements can be given or q_φ is not sufficiently accurate w.r.t. $\mathcal{U}$ for verification. Either way, the assurance level is not as strong as in the former case so that it is only possible to give assurance regarding the validity of q_φ, e.g., model-based testing [3]. Thus, it is also irrelevant whether a fail-safe mode is available, since the claim regarding the assurances of q_φ and the intention at design-time is different.

In terms of class **II**, the same reasoning regarding the existence of a fail-safe mode applies as in class **I**. It is, however, of greater importance regarding the level of assurance. *Medium* analytic capacity means that a subset of states are monitorable according to Definition 3. Therefore, it is possible to prove that q_φ applies for this subset of states; for the remaining states there is the option to transition to fail-safe mode. If there is no fail-safe option, only the validity of q_φ can be assured.

Finally, class **III** gives no assurances at design-time so that q_φ can be only assessed at runtime. This mainly depends on the inability to approximate $P(X'_E|X_\Gamma, X_E)$ or due to the low analytic capacity. Only the fact that there is a fail-safe option or the application is not safety-critical allows to conduct a-posteriori analysis based on the recorded runtime data. Class **IV** is only different to class **III** in that there is in both cases no fail-safe option.

5 RELATED WORK

Addressing dependability at algorithmic level in ML-based systems is not a new research field. For instance, [17] and [16] suggest methods for deep neural networks to address resilience and fault-tolerance due to hardware failures. Varshney and Alemzadeh [33] enumerate several strategies to achieve safety in ML, e.g. *reject options*. Moreover, Abdelzad et al. [1] as well as Gu and Easwaran [14] present techniques that allow to reason about whether specific input data can lead to false predictions. Additionally, there has been a lot of effort to verify the robustness of networks (their resistance against adversarial examples), e.g. [19, 20, 35]. There are also approaches which apply monitoring techniques to ML systems. Alshiek et al. present an approach for safe reinforcement learning by constructing shields (monitors) which guarantee safe behaviour while being as permissive as possible. Another approach by Cheng et al. [7] tries to predict whether a neural network's prediction is reliable based on monitoring the activation pattern of the neurons.

There is, however, very little work that addresses dependability at architectural level. Serbian [28] discusses the idea of using architectural patterns in ML-based systems to deal with safety issues. In the context of autonomous driving, the work of [30] describes several approaches to deal with uncertainty issues of ML. The approaches describe different components, each with its own responsibility (e.g., fail-safe, input checking, etc.) that can be described as architectural patterns.

Finally, in terms of self-adaptive systems, there is also little work. Only de Lemos and Grzes [8] envision the idea of using transparent AI to address uncertainties.

6 CONCLUSION

In this paper, we presented our preliminary work on four classes of architectural dependability, with implications about the level of assurances that can be given at design-time. For this purpose, we identified several dimensions which are responsible for classifying a system and its environment into one of those classes. The dimensions were primarily elaborated to classify SAS which monitor ML-based software. We stress, however, that the classes and dimensions are not limited to the ML context but apply to all systems that undergo an adaptation process at runtime.

In future work, we plan to refine the classification scheme. Moreover, we intend to implement an SAS in a *Human-Robot-Interaction* use case scenario in the context of the CyberProtect[1] project. In the context of this project, the SAS is supposed to safeguard and check potential malicious ML predictions to maintain a specific level of safety. Monitors will be implemented w.r.t. the concepts of explainable ML; we argue that explainable ML is a promising domain to monitor properties of ML-based software. As we approach dependability properties from an architectural level, we want to investigate to what extent assurances can be given by model-based approaches. Finally, we plan to explore a wide range of ML-based software applications and classify them according to our classification scheme in order to argue about the validity of the scheme.

[1] https://www.cyberprotect-bw.de/

ACKNOWLEDGMENTS

This research has been funded by the Ministry of Economic Affairs Baden-Württemberg.

REFERENCES

[1] Vahdat Abdelzad, Krzysztof Czarnecki, Rick Salay, Taylor Denounden, Sachin Vernekar, and Buu Phan. 2019. Detecting Out-of-Distribution Inputs in Deep Neural Networks Using an Early-Layer Output. *arXiv preprint arXiv:1910.10307* (2019).

[2] Mehdi Amoui, Mazeiar Salehie, Siavash Mirarab, and Ladan Tahvildari. 2008. Adaptive action selection in autonomic software using reinforcement learning. In *Fourth International Conference on Autonomic and Autonomous Systems (ICAS'08)*. IEEE, 175–181.

[3] Matthias Becker, Markus Luckey, and Steffen Becker. 2013. Performance analysis of self-adaptive systems for requirements validation at design-time. In *Proceedings of the 9th international ACM Sigsoft conference on Quality of software architectures*. ACM, 43–52.

[4] David M Blei, Alp Kucukelbir, and Jon D McAuliffe. 2017. Variational inference: A review for statisticians. *Journal of the American statistical Association* 112, 518 (2017), 859–877.

[5] Javier Cámara and Rogério de Lemos. 2012. Evaluation of resilience in self-adaptive systems using probabilistic model-checking. In *Proceedings of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. IEEE Press, 53–62.

[6] Chenyi Chen, Ari Seff, Alain Kornhauser, and Jianxiong Xiao. 2015. Deepdriving: Learning affordance for direct perception in autonomous driving. In *Proceedings of the IEEE International Conference on Computer Vision*. 2722–2730.

[7] Chih-Hong Cheng, Georg Nührenberg, and Hirotoshi Yasuoka. 2019. Runtime monitoring neuron activation patterns. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 300–303.

[8] Rogerio de Lemos and Marek Grzes. 2019. Self-adaptive Artificial Intelligence. (2019).

[9] Finale Doshi-Velez and Been Kim. 2017. Towards a rigorous science of interpretable machine learning. (2017).

[10] Tommaso Dreossi, Alexandre Donzé, and Sanjit A Seshia. 2019. Compositional falsification of cyber-physical systems with machine learning components. *Journal of Automated Reasoning* 63, 4 (2019), 1031–1053.

[11] Naeem Esfahani and Sam Malek. 2013. Uncertainty in self-adaptive software systems. In *Software Engineering for Self-Adaptive Systems II*. Springer, 214–238.

[12] Simos Gerasimou, Radu Calinescu, Stepan Shevtsov, and Danny Weyns. 2017. Undersea: an exemplar for engineering self-adaptive unmanned underwater vehicles. In *2017 IEEE/ACM 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, 83–89.

[13] Shixiang Gu, Ethan Holly, Timothy Lillicrap, and Sergey Levine. 2017. Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates. In *2017 IEEE international conference on robotics and automation (ICRA)*. IEEE, 3389–3396.

[14] Xiaozhe Gu and Arvind Easwaran. 2019. Towards safe machine learning for CPS: infer uncertainty from training data. In *Proceedings of the 10th ACM/IEEE International Conference on Cyber-Physical Systems*. ACM, 249–258.

[15] Riccardo Guidotti, Anna Monreale, Salvatore Ruggieri, Franco Turini, Fosca Giannotti, and Dino Pedreschi. 2018. A survey of methods for explaining black box models. *ACM computing surveys (CSUR)* 51, 5 (2018), 93.

[16] Muhammad Abdullah Hanif, Faiq Khalid, Rachmad Vidya Wicaksana Putra, Semeen Rehman, and Muhammad Shafique. 2018. Robust machine learning systems: Reliability and security for deep neural networks. In *2018 IEEE 24th International Symposium on On-Line Testing And Robust System Design (IOLTS)*. IEEE, 257–260.

[17] Le-Ha Hoang, Muhammad Abdullah Hanif, and Muhammad Shafique. 2019. FT-ClipAct: Resilience Analysis of Deep Neural Networks and Improving their Fault Tolerance using Clipped Activation. *arXiv preprint arXiv:1912.00941* (2019).

[18] M Usman Iftikhar, Gowri Sankar Ramachandran, Pablo Bollansée, Danny Weyns, and Danny Hughes. 2017. DeltaIoT: A self-adaptive Internet of Things exemplar. In *Proceedings of the 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. IEEE Press, 76–82.

[19] Guy Katz, Clark Barrett, David L Dill, Kyle Julian, and Mykel J Kochenderfer. 2017. Reluplex: An efficient SMT solver for verifying deep neural networks. In *International Conference on Computer Aided Verification*. Springer, 97–117.

[20] Guy Katz, Derek A Huang, Duligur Ibeling, Kyle Julian, Christopher Lazarus, Rachel Lim, Parth Shah, Shantanu Thakoor, Haoze Wu, Aleksandar Zeljić, et al. 2019. The marabou framework for verification and analysis of deep neural networks. In *International Conference on Computer Aided Verification*. Springer, 443–452.

[21] Daphne Koller and Nir Friedman. 2009. *Probabilistic graphical models: principles and techniques*. MIT press.

[22] Anne Martens, Heiko Koziolek, Steffen Becker, and Ralf Reussner. 2010. Automatically improve software architecture models for performance, reliability, and cost using evolutionary algorithms. In *Proceedings of the first joint WOSP/SIPEW international conference on Performance engineering*. ACM, 105–116.

[23] Gabriel A Moreno, Javier Cámara, David Garlan, and Bradley Schmerl. 2015. Proactive self-adaptation under uncertainty: a probabilistic model checking approach. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 1–12.

[24] Christian Murphy and Gail E Kaiser. 2008. Improving the dependability of machine learning applications. (2008).

[25] Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z Berkay Celik, and Ananthram Swami. 2017. Practical black-box attacks against machine learning. In *Proceedings of the 2017 ACM on Asia conference on computer and communications security*. ACM, 506–519.

[26] Georg Püschel, Christian Piechnick, Sebastian Götz, Christoph Seidl, Sebastian Richly, Thomas Schlegel, and Uwe Aßmann. 2014. A combined simulation and test case generation strategy for self-adaptive systems. *Journal On Advances in Software* 7, 3&4 (2014), 686–696.

[27] Markus Schumacher, Eduardo Fernandez-Buglioni, Duane Hybertson, FrankBuschmann, and Peter Sommerlad. 2006. *Security Patterns - Integrating Security and Systems Engineering*.

[28] Alexandru Constantin Serban. 2019. Designing Safety Critical Software Systems to Manage Inherent Uncertainty. In *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*. IEEE, 246–249.

[29] Sanjit A Seshia, Dorsa Sadigh, and S Shankar Sastry. 2016. Towards verified artificial intelligence. *arXiv preprint arXiv:1606.08514* (2016).

[30] Sina Shafaei, Stefan Kugele, Mohd Hafeez Osman, and Alois Knoll. 2018. Uncertainty in machine learning: A safety perspective on autonomous driving. In *International Conference on Computer Safety, Reliability, and Security*. Springer, 458–464.

[31] Richard S Sutton and Andrew G Barto. 2018. *Reinforcement learning: An introduction*. MIT press.

[32] Marc Toussaint, Amos Storkey, and Stefan Harmeling. 2010. Expectation-Maximization methods for solving (PO) MDPs and optimal control problems. *Inference and Learning in Dynamic Models* (2010).

[33] Kush R Varshney and Homa Alemzadeh. 2017. On the safety of machine learning: Cyber-physical systems, decision sciences, and data products. *Big data* 5, 3 (2017), 246–255.

[34] David Verstraete, Andrés Ferrada, Enrique López Droguett, Viviana Meruane, and Mohammad Modarres. 2017. Deep learning enabled fault diagnosis using time-frequency image analysis of rolling element bearings. *Shock and Vibration* 2017 (2017).

[35] Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. 2018. Efficient formal safety analysis of neural networks. In *Advances in Neural Information Processing Systems*. 6367–6377.

[36] Danny Weyns. 2012. Towards an integrated approach for validating qualities of self-adaptive systems. In *Proceedings of the Ninth International Workshop on Dynamic Analysis*. ACM, 24–29.

[37] Pamela Zave and Michael Jackson. 1997. Four dark corners of requirements engineering. *ACM transactions on Software Engineering and Methodology (TOSEM)* 6, 1 (1997), 1–30.

[38] Jie M Zhang, Earl T Barr, Benjamin Guedj, Mark Harman, and John Shawe-Taylor. 2019. Perturbed Model Validation: A New Framework to Validate Model Relevance. *arXiv preprint arXiv:1905.10201* (2019).