Generating adaptation rule-specific neural networks

Tomáš Bureš[1] · Petr Hnětynka[1] · Martin Kruliš[1] · František Plášil[1] · Danylo Khalyeyev[1] · Sebastian Hahner[2] · Stephan Seifermann[2] · Maximilian Walter[2] · Robert Heinrich[2]

Abstract
There have been a number of approaches to employ neural networks in self-adaptive systems; in many cases, generic neural networks and deep learning are utilized for this purpose. When this approach is to be applied to improve an adaptation process initially driven by logical adaptation rules, the problem is that (1) these rules represent a significant and tested body of domain knowledge, which may be lost if they are replaced by a neural network, and (2) the learning process is inherently demanding given the black-box nature and the number of weights in generic neural networks to be trained. In this paper, we introduce the rule-specific neural network method that makes it possible to transform the guard of an adaptation rule into a rule-specific neural network, the composition of which is driven by the structure of the logical predicates in the guard. Our experiments confirmed that the black box effect is eliminated, the number of weights is significantly reduced, and much faster learning is achieved whilst the accuracy is preserved. This text is an extended version of the paper presented at the ISOLA 2022 conference (Bureš et al. in Proceedings of ISOLA 2022, Rhodes, Greece, pp. 215–230, 2022).

Keywords Self-adaptive systems · Adaptation rules · Machine learning · Neural networks

1 Introduction

The recent advances in neural networks and machine learning [2] led to their proliferation in various disciplines, and the field of self-adaptive systems is no exception [3]. In particular, they have found usage in approaches to control how systems of cooperating agents are formed and reconfigured at runtime [4, 5].

These approaches employ neural networks to implement the self-adaptation loop, also known as the MAPE-K loop, which controls the runtime decisions in the system (e.g., to which service to route a particular request) and the runtime architectural changes (e.g., which services to deploy/undeploy or reconfigure).

In typical cases, a neural network is used for the analysis and planning stages of the MAPE-K loop, replacing the traditional means of analyzing the system state and deciding on adaptation actions. These traditional adaptation mechanisms are often specified in some form of logical rules (e.g., if–then rules or a state machine with guards and actions) [4, 6, 7].

Using a neural network to make decisions on adaptation actions naturally means training the network for the situations the self-adaptive system is supposed to handle. Such training typically requires a large number of system behavior examples—training data in the form of observed inputs and expected adaptation actions. This approach is significantly different from the logical rules that have been traditionally used to describe adaptation actions. Due to this substantial conceptual gap between the two approaches, it is difficult to evolve an existing self-adaptive system based on some form

✉ T. Bureš
bures@d3s.mff.cuni.cz

P. Hnětynka
hnetynka@d3s.mff.cuni.cz

M. Kruliš
krulis@d3s.mff.cuni.cz

F. Plášil
plasil@d3s.mff.cuni.cz

D. Khalyeyev
khalyeyev@d3s.mff.cuni.cz

S. Hahner
sebastian.hahner@kit.edu

S. Seifermann
stephan.seifermann@kit.edu

M. Walter
maximilian.walter@kit.edu

R. Heinrich
robert.heinrich@kit.edu

[1] Charles University, Prague, Czech Republic

[2] Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany

of logical rules into a new system that uses a neural network to make adaptation decisions. Seemingly, the typical design choice is to recreate the analysis and planning stages of the MAPE-K loop from scratch.

The existing logical rules represent a significant body of domain knowledge, especially if the system has been well-functioning and tuned to its task. Thus, when replacing the logical rules with a neural network, this body of domain knowledge is often lost, which leads to severe regress. On the other hand, applying the neural network may be advantageous since it can dynamically learn completely unanticipated relationships of stochastic character. Thus, it makes the self-adaption refined to take advantage of the specific features otherwise hidden in the system and not captured by the inherently static logical rules.

Nevertheless, if logical rules are used to determine the expected actions in training data, it is not easy to train the neural network to reliably yield actions corresponding to the existing rule-based self-adaptive system in question. The main culprit is that the neural network is often built as a black box composed of generic layers (such as a combination of recurrent and dense layers). Thus, the structure of such a generic neural network does not reflect the relationships characteristic of the domain in which the self-adaptive system resides. In other words, the neural network is built as a generic one, not exploiting the existing domain knowledge about the self-adaptive system whose adaptation actions it controls.

While this genericity is inherently advantageous in empowering the neural network to "discover" ultimately unanticipated relationships, it may also hinder the ability to adequately learn because it makes the neural network relatively complex, which potentially increases adaptation uncertainty.

Therefore, replacing a rule-based adaptation entirely with a generic neural network-based one might be an overly drastic change that may potentially degrade the reliability of the system. Moreover, it may raise legitimate concerns since generic neural networks are much less comprehensible and predictable given their black-box nature and the typically large number of weights to be trained—there is always a danger of overfitting.

In this paper, we aim to answer the following research questions: (1) how to endow an existing rule-based self-adaptation system with the ability to learn via neural networks while benefiting from the domain knowledge encoded in the logical rules; and (2) how to scale the learning ability in a way that would allow the transition from logical rules to a neural network to be done on a step-by-step basis.

We address these research questions by introducing a rule-specific Neural Network (*rsNN*) method, which allows the transformation of an adaption rule to the corresponding *rsNN* to be done systematically. The key feature is that an *rsNN* is composable—its architecture is driven by the structure of the logical predicates in the adaption rule in question. Moreover, prior to the composition process, the predicates can be refined by predefined atomic "attunable" predicates, each having a direct equivalent in a primitive element of *rsNN* ("seed" of *rsNN*).

This paper is an updated and substantially extended version of [1]. In particular, another example has been added, the evaluation has been extended as well as the discussed related work.

The rest of the paper is organized as follows. Section 2 presents two examples that are used to motivate and illustrate *rsNN*. Section 3 is devoted to the key contribution of the paper—it describes the concepts and ideas of *rsNN*, furthermore Sect. 4 discusses the methodology, results, and limitations of experimental evaluation. Section 5 discusses other approaches focused on employing neural networks in self-adaptation, and the concluding Sect. 6 summarizes the contribution.

2 Motivating examples

As particular motivating examples, we utilize two realistic yet straightforward use cases. The first one is taken from our former project focused on security in Industry 4.0 settings [8]. The second one focuses on a scheduling problem in the ReCodEx system,[1] a real application for the evaluation of coding assignments used at our institution.

2.1 Industry 4.0 example

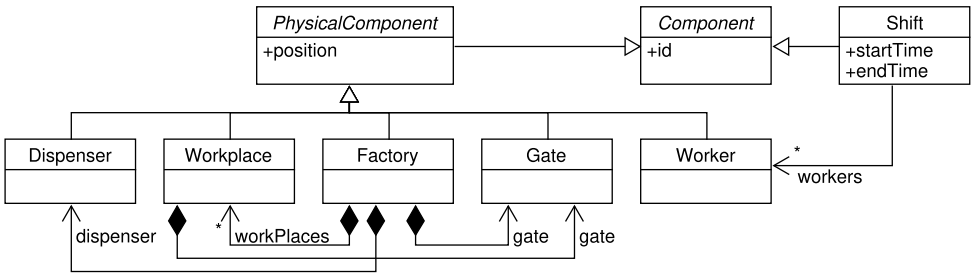
The example employs the MAPE-K loop principle to dynamically reconfigure the software architecture of agents operating jointly on a common task. The agents in our example are workers in a factory. The architecture defines groups of workers that collaborate and operate on a given task. Each group provides access policies that allow the workers to perform their tasks. Since these tasks are subject to change, the access control is intertwined with dynamic, runtime modification of the software architecture.

Implementation-wise, the MAPE-K controller dynamically re-establishes the groups of workers to deal with situations in the environment—e.g., when a machine breaks down, the MAPE-K controller establishes a group of workers that communicate and collaborate to fix the machine (so that the software architecture of components is dynamically reconfigured). It also gives these workers the necessary access rights, e.g., to access the machine's logs and physically enter the room (workplace) where the machine is located.

In the example, we pick up a particular adaptation rule from the larger use-case in the project mentioned above,

[1] https://github.com/ReCodEx.

Fig. 1 Components of the example




```
1  rule AccessToWorkplace(worker) {
2      shift = shifts.filter(worker in shift.workers)
3      guard {
4        duringShift(shift) &&
5          atWorkplaceGate(worker, shift.workplace) &&
6            hasHeadgear(worker)
7      }
8      action {
9        allow(worker, ENTER, shift.workplace)
10     }
11 }
```

Listing 1 Access to workplace rule

and later in Sect. 3, we will employ it to demonstrate a step-by-step transition from this static adaptation rule to the corresponding rsNN neural network.

Let us consider a factory with several workplaces, where production is organized in shifts, each determined by its starting and ending time, during which worker groups perform their tasks. Each group is assigned exactly one workplace, and access to this workplace needs to be managed. The workers are allowed to enter the factory only at a time close to a particular shift's start and must leave soon after the shift ends. After entering, they have to pick up headgear (protective equipment) from a dispenser as a necessary condition for being permitted to enter the assigned workplace. Similarly, they are allowed to enter solely the assigned workplace and only at the time close to the shift start (and have to leave shortly after the shift end). Figure 1 summarizes all the components of the example and their relations.

As expected in the Industry 4.0 domain, the assignment of workers to particular shifts is not static, but can frequently change, and the roles of individual workers within the shift can also alternate rapidly. This leads to changes to the software architecture at the runtime. Consequently, the access control system of the factory cannot assign access rights statically only, thus supporting dynamic, situation-based access control.

To perform access right adaptation, the MAPE-k controller uses adaptation rules in the form of guard-actions, where the action is adding/revoking or allowing access.

Listing 1 shows an example of an adaptation rule, which dynamically determines a group of workers, formed for the duration of a shift, having access rights to the assigned workplace. In particular, the adaptation rule specifies whether a

```
1  pred duringShift(shift) {
2        shift.startTime − 1200 < NOW && shift.endTime
             + 1200 > NOW
3  }
4
5  pred atWorkplaceGate(worker, workplace) {
6      sqrt((workplace.gate.posX − worker.posX) ^ 2 +
7        (workplace.gate.posY − worker.posY) ^ 2) < 10
8  }
9
10 pred hasHeadgear(worker) {
11     worker.events
12       .filter(event −> event.type == (TAKE_HGEAR ||
             RET_HGER))
13       .sortDesc(event −> event.time)
14       .first().type == TAKE_HGEAR
15 }
```

Listing 2 Predicates from Listing 1

specific worker belongs to the group, and if so, it gives the worker access to the workplace assigned for the shift.

The structure of the adaptation rule has three parts. First, there are declared data fields (in this particular case, only a single field initialized to the shift of the given worker—line 2). Second, there is a guard, which defines the condition when the rule is applied. This particular guard reads: To allow a worker to enter the assigned workplace, the worker needs to be already at the appropriate workplace gate (line 5), needs to have a headgear ready (line 6), and needs to be there at the right time (i.e., during the shift or close to its start or end—line 4). Finally, there is an action determining what has to be executed—in this case, the assignment of the allow access rights to the assigned workplace to the worker (line 9).

The guard predicates duringShift, atWorkplaceGate, and hasHeadgear are declared in Listing 2.

The predicate duringShift tests whether the current time is between 20 minutes (i.e., 1200 seconds) before the start of the shift and 20 minutes after the end of the shift. The global variable NOW contains the current time.

The atWorkplaceGate predicate mandates that the position of the worker has to be close (in terms of Euclidean distance) to the gate of the workplace assigned to the worker.

The predicate hasHeadgear checks whether the worker retrieved a headgear from the dispenser. To check this, we assume that each worker is associated with a list of related events (the events data field of the worker—line 11 in Listing 2). For instance, retrieving and returning the headgear

are the events registered in the list of events performing the respective actions. Thus, the check of whether a worker has a headgear available is performed by verifying that after filtering the two specific event types from the list (line 12), the latest event is TAKE_HGEAR (the filtered events are sorted in descending order—line 13 and line 14).

2.2 Coding assignments example

The second example was adopted from the ReCodEx, a system for the evaluation of coding assignments being used at our university, which works as follows. The teachers prepare coding assignments that can be evaluated automatically via automated tests. These assignments are given to students, who solve them and submit their solutions as source code. ReCodEx compiles the solutions and executes them in test suites prepared by the teachers.

For this paper, we have selected a particular problem related to workload distribution over available resources. Since executing the tests is often computationally demanding, the ReCodEx system runs multiple evaluation services (workers) in a private cluster. When a solution is submitted by a student, a new evaluation job is created and has to be assigned to one of the workers. The strategy used to distribute jobs affects how long the student has to wait for the test results. Not surprisingly, there are always spikes in utilization just before the assignments' deadlines. At these points, the cluster is overutilized, and the prioritization of jobs becomes important. The goal becomes to prioritize the jobs adequately and route them to the respective workers in the cluster so that the students do not complain too much about the ReCodEx system being slow.

The main issue here is to determine the length of the evaluation job from the subjective perspective, i.e., whether it will be *short* or *long*. If the job is expected to be short, the student is likely to remain in the user session waiting eagerly for the results. If it is expected to take a long time, the user is likely to attend to other duties and return to the system after some time. In other words, the short jobs need to be prioritized to minimize their evaluation latency, while the long jobs may be delayed slightly without the user objecting. Unfortunately, the jobs must be assigned to workers in a non-preemptive manner (i.e., once an evaluation starts, it must be completed), so assigning long-running evaluations to all workers could easily delay many short jobs, which would most likely lead to student complaints.

To reduce the possibility that long-running jobs would clog the system, the workers were divided into two categories. *Regular* workers, who process all kinds of jobs, and *priority* workers, who are dedicated to short jobs only. The most important remaining issue is how to determine whether an evaluation job will be short or long.

```
1 pred isShort(job) {
2                   job.refSolutionDuration < 60 &&
                        job.timeLimit < 300
3 }
```

Listing 3 Predicate whether a job is deemed short

```
1 pred isIdle(worker) {
2                   worker.jobQueue.length == 0
3 }
4
5 pred isBusy(worker) {
6                   worker.jobQueue.length > 2 ||
                        worker.jobQueue.exists(job -> not
                        isShort(job))
7 }
```

Listing 4 Predicates related to worker load

2.2.1 Length estimation

Getting an accurate estimation regarding the length of the evaluation is quite difficult, since it depends on many factors. Even the times of solutions of the same assignment may vary significantly, since each student may have chosen an algorithm with different time complexity, not to mention that some solutions may hold serious bugs like infinite loops.

To get an estimate of how long it could take to test the solution of a particular assignment, we use two important values:

1. the time limits set for the tests of the particular assignment
2. and evaluation time of the reference solution provided by the teacher.

This gives us the following predicate for deciding the type of job.

Besides that, we do not know the actual length of the evaluation, there is another variable in the equation. The objective is to prevent student complaints, but each student has a subjective time threshold regarding the expectation of how long his/her evaluation could actually take. Thus the explicit thresholds (60 and 300 seconds) in Listing 3 are simply an educated guess.

2.2.2 Job scheduling

The assignment of a job to workers is also affected by the current workload of the workers. The predicates in Listing 4 help us distinguish *idle* workers which have nothing to do and *busy* workers which are somewhat overloaded—in our case, we defined busy as having either more than two jobs or at least one job that was assumed to have been in the queue for a long time. Workers that are not idle or busy are assumed to have a regular workload. The scheduler should prefer idle workers and avoid busy workers when possible. Also, note that the definition of a busy worker depends on internal parameters that can be derived from the total number of workers as well

as the average workload the system handles. The constants embedded in the predicate below were selected empirically based on the experience with the actual system.

The job scheduling algorithm can be summarized in the following simple rules:

1. Make an evaluation length estimation based on the parameters of the submission and associated assignment.
2. If there is an idle worker who would accept the job (i.e., a regular worker in case of a long job or any worker in case of a short job), assign the job to him.
3. If the job is short and there is a priority worker who is not busy, assign the job to him.
4. If there is a regular worker who is not busy, assign the job to him.
5. If the job is short, assign it to a random priority worker. Otherwise, assign it to a random regular worker.

The complete algorithm would be slightly more complex as it needs to handle job or worker failures as well; however, these issues are beyond the context of this paper.

3 Refining adaptation rules

The problem with the adaptation rules we presented in Sect. 1 is that their guards are too static, and thus they do not capture the domain-specific stochastic character of the data they act upon. As already mentioned in Sect. 1, we aim to employ a dedicated *rule-specific Neural Network* to benefit from its ability to learn from the domain characteristic data being handled. To this end, in this section, we outline the method that allows us to refine an original adaption rule to make its guard predicates "attunable" and convert the guard into an *rsNN*. In a sense, our method of employing a dedicated *rsNN* for this purpose can be viewed as paving a middle ground between the adaptation rules with static guards and the adaptation rules driven by (typically complex) generic neural networks such as in [4, 9].

The main idea of our method unfolds in three stages:

1. An adaptation rule is *refined* by manually rewriting (transforming) its selected guard predicates into their attunable form—they become *attunable predicates*. This is done by applying predefined atomic attunable predicates (*aa-predicates*) listed in Sect. 3.1. These aa-predicates serve as *rsNN* seeds in the second stage. Nevertheless, not all the guard predicates have to be transformed this way—those remain *static predicates* (their selection is application-specific).
2. We apply an automated step that generates an *rsNN* that reflects the guard of the refined adaptation rule, containing, in particular, the trainable parameters of aa-predicates as trainable weights.
3. We employ traditional neural network training using stochastic gradient descent to pre-train the trainable weights.

The result is an *rsNN* being a custom neural network, the composition of which is driven by the structure of the guard formula with aa-predicates. This neural network is pre-trained to match the outputs of the original guard formula of the adaptation rule. Nevertheless, being a neural network, it can be further trained by running additional examples.

As to pre-training data, we assume there are sample traces of input data to the system, obtained either from historical data, simulation, or random sampling. We use the logical formulas of the original guard predicates over the input data to provide the ground truth (i.e., expected inputs) employed in the supervised learning of the *rsNN*.

Further, the developer has the ability to specify the learning capacity in many aa-predicates, which in turn determines how many neurons are used for their implementation in the *rsNN*.

3.1 Atomic attunable predicates as *rsNN* seeds

This section provides an overview of the aa-predicates defined in the *rsNN* method. The key idea is that these aa-predicates serve as elementary building blocks for attunable predicates forming an adaptation rule, and at the same time, each of them is easily transformable into a building block of the corresponding *rsNN*—serving as an *rsNN* seed as defined in Sect. 3.3.

Each aa-predicate operates on a single n-dimensional input value (i.e., a fixed-size vector). Since each aa-predicate yields a true/false value, its corresponding *rsNN* seed solves a classification task, yielding likewise true/false.

Following the type of input value domain, we distinguish between aa-predicates that operate on domains with a metric (i.e., with the ability to measure the distance between quantities) and categorical quantities, where no such metric exists:

1. **Metric Quantity:** There are two types of aa-predicates defined over a given metric:
   (a) Quantity lies in a one-sided interval

   $$isAboveThreshold\_nD(x, min, max)$$

   $$isBelowThreshold\_nD(x, min, max)$$

   Here x is a value in an n-dimensional space that is compared to a learned threshold (above or below) by the corresponding *rsNN* seed. In order to control the uncertainty that is potentially induced by learning, the *min* and *max* parameters impose the limits for the learned threshold.
   (b) Quantity lies in a two-sided interval

   $$hasRightValue\_nD(x, min, max, c)$$

   Here it is verified whether the parameter x lies inside the learned interval of an n-dimensional space. The

parameters *min* and *max* have the same meaning as in the case of the aa-predicates for a one-sided interval, while the parameter *c* states the learning capacity of the corresponding *rsNN* seed; technically, this is, e.g., the highest number of the neurons in a hidden layer of the *rsNN* seed.

2. **Categorical quantity:** For this type of input domain, we define an aa-predicate that decides whether a categorical quantity has the right value:

$$hasRightCategories\_nD(x,m,c)$$

Here x is an n-dimensional vector of categorical values from the same domain of the size m (the number of categories). The corresponding *rsNN* learns which combinations of categorical values in the input vector satisfy this aa-predicate. The learning capacity is determined by c.

3.2 Making guard predicates attunable

In this section, we demonstrate the first stage of the *rsNN* method (i.e., the manual rewriting of guard predicates) on the Industry 4.0 example presented in Sect. 2. We show two alternatives for such rewriting to demonstrate that the designer may choose several ways to make a predicate attunable, depending on what quantities are to be the subject of future learning.

We start with the guard predicates shown in Listing 2. At first, we assume that the designer would like to rewrite duringShift to make it attunable, with the goal to learn the permitted time interval in which the access is allowed. For example, security reasons may require learning the typical behavior patterns of workers induced by the public transportation schedule. (On the contrary, in Listing 2, the interval is firmly set from 20 minutes before the shift starts to 20 minutes after the shift is over.)

We rewrite the duringShift guard predicate as shown in Listing 5: The comparison of NOW with a particular threshold is replaced by the aa-predicates isAboveThreshold and isBelowThreshold, respectively. Each of them represents a comparison with a learned threshold.

The aa-predicates isAboveThreshold and isBelowThreshold have three parameters: (1) the value to test against the learned threshold, (2) the minimum value of the threshold, (3) the maximum value of the threshold.

Since this threshold should not depend on the actual time of the shift, the times are given relative to its start and end. By assuming a worker cannot arrive earlier than one hour before the shift starts (+3600 seconds in line 2), the relative time 0 corresponds to that point in time (as computed by NOW + 3600 - shift.end). Similarly, by assuming a worker cannot leave later than one hour after the shift ends (-3600 seconds

```
1  pred duringShift(shift) {
2    isAboveThreshold_1D(NOW + 3600 − shift.start,min=0,
         max=36000) &&
3    isBelowThreshold_1D(NOW − 3600− shift.end,
         min=−36000, max=0)
4  }
5
6  pred atWorkplaceGate(worker, workplace) {
7    sqrt((workplace.gate.posX − worker.posX) ^ 2 +
         (workplace.gate.posY − worker.posY) ^ 2) < 10
8  }
9
10 pred hasHeadgear(worker) {
11   worker.events.filter(event −> event.type ==
         (TAKE_HGEAR || RET_HGER))
12                 .sortDesc(event −> event.time).first().type
                       == TAKE_HGEAR
13 }
```

Listing 5 Guard predicates with refined duringShift by aa-predicates—one-sided intervals

in line 3), the relative time 0 corresponds to that point in time (as computed by NOW - 3600 - shift.end). The minimum and maximum values of the threshold correspond to the interval of 10 hours (i.e., 36,000 seconds).

The other predicates atWorkplaceGate and hasHeadgear) stay the same, as does their conjunction in the AccessToWorkplace rule.

Note that we combine static predicates with an attunable predicate. This shows that only a part of a rule can be endowed with the ability to learn, while the rest can stay unchanged. At the same time, we put strict limits on how far the learning can go. In the example, these limits are expressed by the interval of 10 hours, which spans from one hour before the shift to one hour after the shift (assuming the shift takes 8 hours). In other words, the value in the attunable predicate gained in the process of learning cannot exceed these bounds. This is useful if learning is to be combined with strict assurances with respect to uncertainty control.

As another alternative of the rule refinement, we assume the time of entry, place of entry, and the relation to the last event concerning the headgear is to be learned. Also, contrary to the variant of duringShiftin Listing 5, we assume the time of entry is not just a single interval, but it can include multiple intervals (e.g., to reflect the fact that workers usually access the gate only at some time before and after the shift due to the public transportation schedule).

To capture this, we rewrite the predicates duringShift, atWorkplaceGate, and hasHeadGear as shown in Listing 6.

The guard predicate duringShift is realized using the aa-predicate hasRightValue_1D, which represents a learnable set of intervals. It has four parameters. In addition to the first three, which have the same meaning as before (i.e., value to be tested on whether it belongs to any of the learned intervals, the minimum and the maximum values for the intervals), there is the fourth parameter capacity, which expresses learning capacity. The higher it is, the finer intervals the predicate is able to learn. Since it works relative to the

```
pred duringShift(shift) {
    hasRightValue_1D(NOW − shift.start, min=−3600,
        max=36000, capacity=20)
}

pred atWorkplaceGate(worker) {
    hasRightValue_2D[worker.workplace.id](worker.pos,
            min=(0,0), max=(316.43506,177.88289),
                capacity=20)
}

pred hasHeadGear(worker) {
    hasRightCategories_1D(
      worker.events.filter(event −> event.type ==
          (TAKE_HGEAR || RET_HGER))
        .sortDesc(event −> event.time).take(1),
            categories=2, capacity=1
    )
}
```

Listing 6 Guard predicates expressed by a two-sided interval and categorical quantity aa-predicates

```
pred isShort(job) {
    isBelowThreshold_1D(job.refSolutionDuration,
                        min=10, max=120) &&
    isBelowThreshold_1D(job.timeLimit,
                        min=10, max=600)
}

pred isBusy(job) {
    hasRightValue_1D(worker.jobQueue.length,
                     min=1, max=10, capacity=10) ||
    hasRightValue_1D(worker.jobQueue
        .filter(job −> not isShort(job)).length,
        min=1, max=10, capacity=10
    )
}
```

Listing 7 Predicates isShort and isBusy in atunable form

min/max parameters, it is unitless. Technically, the learning capacity determines the number of neurons used for training. The exact meaning of the capacity parameter is given further in Sect. 3.3.

The guard predicate atWorkplaceGate is rewritten similarly. However, as the position is a two-dimensional vector, a 2D version of the hasRightValue aa-predicate is used. The meaning of its argument is the same as in the 1D version applied for duringShift. A special feature of atWorkplaceGate is that it is specific to the workplace assigned to the worker. (There are several workplaces where the work is conducted during a shift. Each worker is assigned to a particular workplace, and their access permission is thus limited only to that workplace.) Thus, the hasRightValue_2D aa-predicate has to be trained separately for each workplace. The square brackets express this after the hasRightValue_2D aa-predicate, which signifies that its training is qualified by workplace ID. Since the example assumes that there are three workplaces in a shift, there are three aa-predicates to be trained.

The hasHeadGear guard predicate is rewritten using the hasRightCategories_1D aa-predicate, which assumes 1-dimensional vector of categorical values (i.e., a single value in this case) from the domain of size 2. In this simple case, the learning capacity is set to 1.

In a similar vein, looking at the coding assignments example (Sect. 2.2), we can refine the predicates isShort (Listing 3) and isBusy (Listing 4) using aa-predicates as in Listing 7.

3.3 Construction of *rsNN*

In this section, we formalize the second stage of the *rsNN* method, i.e., the automated construction of an *rsNN* that reflects the guard of a refined adaptation rule. First, we show how to transform a logical formula into an elementary *rsNN* (*rsNN seed*) in general, and how to combine *rsNN* seeds into larger units (and how to combine these larger units as well) via transformed logical connectives. Then, we describe how the elementary logical formulas in the guard (i.e., static predicates and aa-predicates) are transformed into *rsNN* seeds.

Transforming a logical formula and connectives.

A logical formula $L(x_1,\ldots,x_m)$ is transformed to a continuous function

$N(x_1,\ldots,x_m,w_1,\ldots,w_n) \to [0,1]$ (i.e., a neural network), where $x_1,\ldots,x_m$ are the inputs to the logical formula (e.g., the current time, position of the worker in an aa-predicate) and $w_1,\ldots,w_n$ are trainable weights. The goal is to construct the function N and train its weights such that $L(x_1,\ldots,x_m) \Leftrightarrow N(x_1,\ldots,x_m,w_1,\ldots,w_n) > 0.5$ for as many inputs $x_1,\ldots,x_m$ as possible. By convention, we interpret $N(\ldots) > 0.5$ as $true$, while if this relation does not hold, it is interpreted as $false$. Also, we use the symbol $\mathcal{T}$ to denote the transformation from the logical formula L to the continuous function N—i.e., $N(\ldots) = \mathcal{T}(L(\ldots))$.

As to logical connectives, we deviate from the traditional notion in which conjunction is defined as a product and disjunction is derived using De Morgan's laws. This is because our experiments showed that the conjunctions of multiple operands almost exclude training (very likely due to the vanishing gradient problem [10]). Therefore, we transform conjunction and disjunction as follows (similarly to in [11]):

$$\mathcal{T}(L_1 \& \ldots \& L_k) = S((\mathcal{T}(L_1) + \cdots + \mathcal{T}(L_k) - k + 0.5) * p)$$

$$\mathcal{T}(L_1 \vee \cdots \vee L_k) = S((\mathcal{T}(L_1) + \cdots + \mathcal{T}(L_k) - 0.5) * p)$$

$$\mathcal{T}(\neg L) = 1 - \mathcal{T}(L),$$

where $S(x)$ is the sigmoid activation function defined as $S(x) = \frac{1}{1+e^{-x}}$, and $p > 1$ is an adjustable strength of the conjunction/disjunction operator. The bigger it is, the stricter the results are. However, too high values have the potential to harm training due to the vanishing gradient problem.

Transformation of a static predicate. A static predicate is transformed simply into a function that returns 0 or 1 depending on the result of the static predicate. Formally, we

transform a static predicate $L_S(x_1,\ldots,x_m)$ to the function $N_S(x_1,\ldots,x_m)$ as follows:

$$\mathcal{T}(L_S) = \begin{cases} 0 & \text{if not } L_S(x_1,\ldots,x_m) \\ 1 & \text{if } L_S(x_1,\ldots,x_m) \end{cases}$$

Transformation of one-sided interval aa-predicates. We transform an aa-predicate $isAboveThreshold(x,min,max)$ to the function $N_>(x,w_t)$ and an aa-predicate $isBelowThreshold(x,min,max)$ to the function $N_<(x,w_t)$ as follows.

$$\mathcal{T}(isAboveThreshold) = S\left(\left(\frac{x-min}{max-min} - w_t\right) * p\right),$$

$$\mathcal{T}(isBelowThreshold) = S\left(\left(w_t - \frac{x-min}{max-min}\right) * p\right),$$

where w_t is a trainable weight.

Transformation of two-sided interval aa-predicates. We base these aa-predicates on radial basis function (RBF) networks [12]. We apply one hidden layer of Gaussian functions and then construct a linear combination of their outputs. The weights in the linear combination are trainable. The training capacity c in the aa-predicate determines the number of neurons (i.e., points for which the Gaussian function is to be evaluated) in the hidden layer.

We set the means μ_i of the Gaussian function to a set of points over the area delimited by *min* and *max* parameters of the aa-predicate (e.g., forming a grid or being randomly sampled from a uniform distribution). We choose the σ parameter of the Gaussian function to be of the scale of the mean distance between neighboring points. The exact choice of σ seems not to be very important. Our experiments have shown that it has no significant effect and what matters is only its scale, not the exact value. The trainable linear combination after the RBF layer automatically adjusts to the chosen values of μ_i and σ.

For the sake of clarity, we show the transformation of

$$hasRightValue\_nD(x,min,max,c)$$

for $n = 1$ and for arbitrary n. In the 1-D case, we transform an aa-predicate $hasRightValue\_1D(x,min,max,c)$ to the function $N^1_{\gtrsim}(x,w_{a_1},\ldots,w_{a_c},w_b)$ as follows:

$$\mathcal{T}(hasRightValue\_1D) = S\left(w_b + \sum_{i=1}^{c} w_{a_i} e^{-\frac{(\mu_i - x)^2}{2\sigma^2}}\right),$$

where c is the capacity parameter of the predicate, $\mu_i \in [min,max]$ and σ are set as explained above, and $w_{a_1},\ldots,w_{a_c},w_b$ are trainable weights.

This is generalized to the n-D case as follows:

$$\mathcal{T}(hasRightValue\_nD) = S\left(w_b + \sum_{i_1=1}^{c}\cdots\sum_{i_n=1}^{c} w_{a_{i_1,\ldots,i_n}} e^{-\frac{|\mu_{i_1,\ldots,i_n} - x|^2}{2\sigma^2}}\right),$$

where $\mu_{i,j} \in [min_1,max_1]\times\cdots\times[min_n,max_n]$ and σ are set as explained above, x is an n-D vector, $|\cdot|$ stands for vector norm, and $w_{a_{1,\ldots,1}},\ldots,w_{a_{c,\ldots,c}},w_b$ are trainable weights.

Transformation of a categorical quantity aa-predicate. We base this aa-predicate on a multi-layer perceptron with one hidden layer, which has the number of units equal to the capacity parameter c of the aa-predicate and is activated by the ReLU activation function.

The transformation of an aa-predicate

$$hasRightCategories\_nD(x,m,c)$$

to the function

$$N_{\doteq}(x,w^h_{a_{1,1}},..,w^h_{a_{c,m}},w^h_{b_1},..,w^h_{b_c},w^o_{a_1},..,w^o_{a_c},w^o_b)$$

is defined as follows:

$$\mathcal{T}(hasRightCategories\_nD) = S\left(w^o_b + \sum_{i=1}^{c} w^o_{a_i}\,\mathrm{ReLU}\left(w^h_{b_i} + \sum_{j=1}^{n}\sum_{k=1}^{m} w^h_{a_{i,j,k}}\delta_{x_j,k}\right)\right),$$

where $x \in \{1,\ldots,m\}^n$ is the n-dimensional input vector of categorical values from the same domain of size m, c is the capacity, $w^h_{i,j,k}$, w^h_b are trainable weights of the hidden layer, $w^o_{a_i}$, w^o_b are trainable weights of the output layer, $\delta_{i,j}$ is the Kronecker delta, i.e., $\delta_{i,j} = 1$ if $i = j$ and $\delta_{i,j} = 0$ otherwise. The ReLU function is defined as $\mathrm{ReLU}(x) = \max(0,x)$. Note that the Kronecker delta in the formula stands for one-hot encoding of the categorical input values.

3.4 Training an *rsNN*

The N-function defined as the result of the transformations (Sect. 3.3) contains trainable weights. We train these weights using supervised learning and employing the traditional stochastic gradient descent optimization.

The samples for training are taken from existing logs obtained from the system runtime or a simulation. In the case of the Industry 4.0 example, each sample contains the current time, the worker's id, his position, and the history of events associated with the worker. To obtain accurate outputs for supervised learning, we exploit the fact that we have the original logical formula of the guard with static predicates available. Thus, we use it as an oracle for generating the

ground truth for training inputs. The exact training procedure is described in [13].

After this training step, the function N can be used as a drop-in replacement for the corresponding adaptation rule. Moreover, being a neural network, it is able to digest additional samples generated at runtime, e.g., to learn from situations when the outputs of the system have been manually corrected/overridden.

3.5 Advanced use: employing atunable predicates as safeguards

In this section, we show that an rsNN can serve as a safeguard in the decision process of an externally provided generic (black-box) artificial neural network (ANN) when the safety of a decision process is an issue. Integration with logical decisions would be straightforward, so we have chosen a more complex classification problem.

The presented case is based on the Coding Assignments Example (Sect. 2.2), where the job scheduling algorithm can be replaced by a classification ANN that decides to which worker (class) a job should be assigned. Such a network can take all the information from the submission, related assignment, or even current states of the workers as the input, so it could make more astute decisions than a static algorithm, even if we make the predicates like *isShort* or *isBusy* attunable. However, using a black-box ANN may lead to unexpected behavior, thus creating instability in the system. In the following, we will show how to combine the ANN with logical formulas converted in *rsNN* that will act as a safeguard. The ANN will be still in charge of selecting a worker for a given job, but the safeguard will prevent assigning a long-running job to a priority worker.

A typical classification neural network has an output layer that contains one neuron for each class and uses *softmax* activation function. To integrate the *isShort* predicate with the ANN, we use the following formula:

$$\begin{aligned}\mathcal{T}(ANN,isShort) = \sigma(&\\ &\mathcal{T}_{/\sigma}(ANN)[r_1] + 1 - \mathcal{T}(isShort), \ldots,\\ &\mathcal{T}_{/\sigma}(ANN)[p_1] \cdot \mathcal{T}(isShort), \ldots)\end{aligned}$$

The σ is the softmax function, $\mathcal{T}_{/\sigma}$ is the output of ANN without the last softmax layer, r_i denotes output nodes that correspond to regular workers, and p_i refers to priority worker outputs. Let us clarify that the same modification is performed for all regular and priority outputs, respectively.

The formula performs basically the following. The probabilities of classes that correspond to the priority workers are zeroed if the job is assessed as long. Similarly, the probabilities of the regular workers are boosted by 1 if the job is long to avoid pathological situations such as when the network assigns 0 to all regular workers.

Naturally, the attunable version of a complex predicate can be used with an *rsNN* employed in parallel with a black-box ANN and then applied to adjust the outputs.

4 Evaluation

We evaluated our approach by comparing the training results of *rsNNs* created by the method proposed in Sect. 3 with generic NNs comprising one and two dense layers. The complete set of necessary code and data for replicating the evaluation, as well as the experiments, detailed evaluation of results, graphs, and discussion that did not fit this paper, are available in the replication package [13].

Please note that we intentionally do not compare *rsNNs* (that use supervised learning) with methods like deep reinforcement learning (DRL) or DQN, as they target a different problem and thus cannot be straightforwardly compared.

4.1 Industry 4.0 example

4.1.1 Methodology and datasets

For the Industry 4.0 example, we created two datasets: (a) *random* sampled dataset, which was obtained by randomly generating inputs and using the original logical formula of the guard as an oracle; (b) *combined* dataset, which combines data from a simulation and the random dataset. Both datasets have about 500,000 of data points.

The datasets were balanced in such a manner that half of the samples correspond to *true* and half to the *false* evaluation of the guard of AccessToWorkplace. Additionally, to obtain more representative results for evaluation, the false cases were balanced so that each combination of the top-level conjunctions outcomes (i.e., duringShift & atWorkplaceGate & hasHeadGear) has the same probability.

The combined dataset combines false cases from random sampling and true cases from a simulation. The simulation was performed by a simulator[2] we developed in the frame of an applied research project (Trust 4.0). The reason for combining these two sources is to get better coverage of all possible cases when the guard of the adaptation rule evaluates to *false*.

For the code assignments example, we used anonymized job logs from the ReCodEx system. These logs were combined in a Cartesian product with all possible states of worker job queues (to express how much each worker is busy), and the training outputs were computed using a rule-based algorithm that also takes into account the supposed feedback from the users.

[2] https://github.com/smartarch/trust4.0-demo.

As the baseline generic NNs, we selected dense neural networks. Given our experiments and consultation with an expert outside our team (a researcher from another department who specializes in practical applications of neural networks), this architecture suits the problem at hand the best. Our setup comprises networks with one and two dense layers of 128 to 1024 nodes (in the case of two layers, both of them have the same amount of nodes). The dense layers use ReLU activation and the final layer uses sigmoid. The greatest accuracy was observed when two 256-node dense layers were used; thus, this configuration was selected as the *baseline*.

Three versions of rsNNs representing our approach were built corresponding to different levels of refinement. The first two models refined only the time condition: one used the isAboveThreshold and isBelowThreshold variant (as in Listing 5)—denoted as *time (A&B)*, the other used hasRightValue aa-predicate (similar to Listing 5, but with hasRightValue instead of the combination of isAboveThreshold and isBelowThreshold)—denoted as *time (right)*. The last model refined all involved inputs (time, place, and headgear events) as outlined in Listing 6 — denoted as *all*. To verify the properness of logical connectives redefinition (Sect. 3.3), we built a TensorFlow[3] model with no trainable weights (i.e., just rewriting the static predicates using their transformation described in Sect. 3.3). By setting $p = 10$, we achieved 100% accuracy (this value of p was then used in all other experiments).

All NN models were implemented in the TensorFlow machine learning framework and trained on our local GPU cluster. Each dataset was divided into the training part (90%) and the validation part (10%).[4] We measured accuracy on the validation part only. All experiments were repeated 5×; in this evaluation, we present the mean values. For the sake of brevity, we do not report on standard deviations in detail since they were very low (less than 0.1% of accuracy in the worst case).

All training sessions used the batch size of 100 and 100 epochs. We have used the Adam optimizer with cosine decay of the learning rate. In the case of the generic NNs, we have used label smoothing to prevent overfitting.[5] The *rsNNs* did not suffer from overfitting even without label smoothing; thus, we did not apply the label smoothing in this case.

4.1.2 Results

Figure 2 compares the accuracy of tested generic NN configurations after 100 epochs. The greatest accuracy was observed when two 256-node layers were used. Thus, we have selected this configuration as the baseline representative for further comparisons (we further refer to this configuration simply as the "baseline"). Smaller configurations suffer from lower capacity, while larger ones have a stronger inclination to overfitting (especially in the case of the combined dataset). It is also worth mentioning that we as well experimented with other sizes/configurations without any improvement.

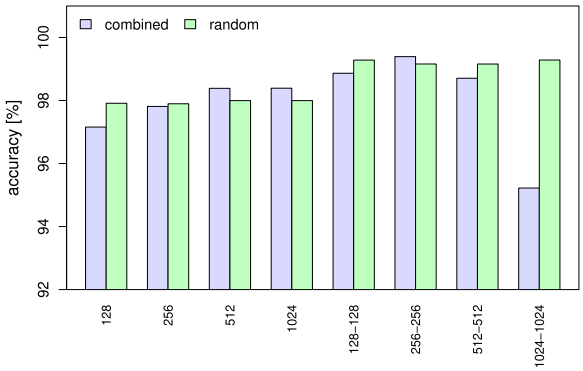


Fig. 2 Accuracy of the baseline solution (dense networks with one or two layers). The column labels denote the width of dense layers

Table 1 presents the measured accuracies on the testing set[6] of both datasets (random and combined) after 100 training epochs, comparing *rsNNs* resulting from different refinements with the baseline. The last two models outperform the baseline in terms of accuracy. The *number of Weights* line refers to the number of trainable weights in each model. While the baseline has many weights (as it features two dense layers), our *rsNNs* have significantly fewer weights, since their composition benefits from the domain knowledge ingrained in the adaptation rules.

The lower number of trainable parameters positively impacts the performance, as it makes the models train and evaluates significantly faster whilst achieving comparable accuracy levels. We did not perform a thorough performance analysis since it heavily depends on many configuration parameters (e.g., batch size) and the actual hardware (especially whether CPU or GPU is used for the training). However, in our configurations, the proposed model was trained roughly several times (up to an order of magnitude) faster than the baseline, as shown in Fig. 3. The generic NNs are much slower in the learning process since they comprise many more trainable weights. Our *rsNNs* reach their peak accuracy within 10 epochs. In all cases, we observed that 100 epochs was sufficient for all models, so we have used this as the limit.

[3] https://www.tensorflow.org/ (version 2.4).

[4] The ratio was selected with respect to the size of the data (we needed a sufficient training set) and given the fact the validation is performed merely to verify the feasibility of the approach.

[5] We have been experimenting with Dropout layers as well, but they were causing significant underfitting.

[6] We divide the data only to the training and testing set (testing set holds 10% of data). We do not need a validation set since we do not perform any hyper-parameter training.

Table 1 Comparison of accuracies of individual methods

	baseline	time (A&B)	time (right)	all
Accuracy (random)	99.159%	98.878%	99.999%	99.978%
Accuracy (combined)	99.393%	92.867%	99.993%	99.575%
number of weights	68,353	2	21	1227

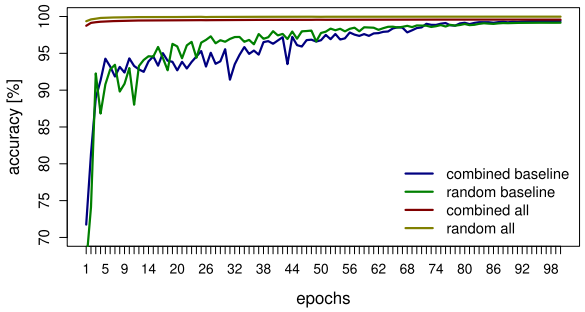


Fig. 3 Training speed demonstrated as validation set accuracy measured after every epoch

4.2 Coding assignments results

The results obtained from the coding assignment example (Sect. 2.2) were similar to the Industry 4.0 example (Sect. 4.1.2). The ANN enhanced with *rsNN* as a safeguard exhibited very similar accuracy to the original (generic) ANN. Therefore, we claim that the safeguard did not hinder the classification process, but provided guarantees for behavior that could not be extracted from a black-box generic ANN. A detailed description of the experiments targeting the coding assignments example is provided in the replication package [13].

The overall accuracy is lower than in the case of the Industry 4.0 example (about 85%), which is caused by the fact this example uses real data from an existing system. Real data contain much more noise than the data from a simulated example.

4.3 Limitations and discussions

Here we discuss the potential limitations of *rsNNs*.

An obvious question is the scalability of *rsNNs*. Since we train the neural networks for each predicate separately, *rsNNs* do not suffer from an exponential blow-up of complexity when the system goes large (in the number of involved components), and the complexity linearly scales with the number of used predicates. Also, since the predicates are trained independently, the training can be done in parallel and thus scaled horizontally, e.g., in a cloud.

Another natural question is how the method deals with possible dynamic changes in a (real-world) environment (which is a common problem). Here, the strong advantage of our method is that it combines logical rules with the ability to adjust the results based on training on samples collected from the running system. This allows regular retraining of *rsNNs* that can integrate new samples collected from the real-world system and thus adjust themselves to potential dynamic changes in the system.

Similarly to the real-world environment dynamicity, there is the danger that real-world data used for training might be noisy and limited. However, compared with other methods based on machine learning, our approach generates small NNs that have far fewer parameters compared to regular NNs (e.g., dense networks) that are normally used to achieve the same accuracy in decisions. This makes *rsNNs* generalize much better than regular NNs. The ability to better generalize also leads to greater robustness towards noise and other disruptions in the input data.

4.4 Threats to validity

Though we did our best, given the limited scope of the paper, we are aware of several threats to the validity of our evaluation. Nevertheless, we believe the evaluation still provides valuable insight into the potential of the *rsNN* method.

The threats to validity are presented below based on the schema in [14], where the validity classes are defined as follows: (i) construct validity, (ii) internal validity, (iii) external validity, and (iv) reliability.

4.4.1 Construct validity

There is a danger that we devised our evaluation wrongly. As a metric, we consider the accuracy achieved by *rsNN* and compare it with a generic neural network, which we chose as the baseline. This way, we show that *rsNN* does not underperform, even though it uses substantially fewer training weights (being thus more robust) and trains faster. We believe this metric is adequate for supporting the argument on the benefit of *rsNN* we articulate in the conclusion of the paper.

4.4.2 Internal validity

There is a possible danger that the employed generic neural network was trained as a black box. Therefore, the reported improvements exhibited by *rsNNs* might not result from the

proposed *rsNN* method, but rather from a hidden factor that we are not aware of. Nevertheless, we aim to mitigate it by using exactly the same inputs and training procedure when training and evaluating the baseline generic NNs and *rsNNs*.

4.4.3 External validity

Given the limited scope of the paper, we demonstrated rsNN on two adaptation rules only (i.e., AccessToWorkplace and isShort). We chose these rules because they showcase the combination of multiple concerns (time, position, event history, and user preference). However, we are aware of the fact that a larger case study would be needed to identify a richer set of the types of aa-predicates. Also, the limited complexity of the predicate does not allow to fully test the limits of what can be learned by an *rsNN*. We tried to mitigate this problem to a certain extent by creating additional experiments with synthetically built guards (featuring, e.g., a conjunction of more than three aa-predicates). To illustrate the potential fields of application of the *rsNN* method, we also evaluated it on two use cases from significantly different domains. It is worth noting that the coding assignment example stems from an existing system used in our Computer Science departments on a daily basis. Thus, the data for evaluation were taken from a "production environment".

4.4.4 Reliability

Another threat to validity is that we constructed the baseline generic NNs on our own. To help reduce the potential bias here, we consulted an expert outside our team to select the best possible generic NN architecture fitting the nature of the data. Moreover, we performed an automated hyperparameter tuning to help us identify the best generic NN architecture, which was eventually used as the baseline.

5 Related work

Using neural networks for the representation of logical formulas and evaluation of fuzzy logical systems is not a new idea. A kind of calculus for the evaluation of propositional logic formulas can be found in [15] together with a proposal for the evaluation of fuzzy logical systems. However, the activation function of neurons in the proposed approach is a simple threshold function. More recent approaches (e.g., [16, 17]) use the sigmoid function or linear interpolation. We take a similar approach and push this further to practical application aligned with refining self-adaptive systems. Also, our approach features a practical approach to logical connectives that are easier to train.

In the domain of adaptive and cyber-physical systems, neural networks and machine learning are used in a number of areas. Not closely related, but rather a large area is the employment of neural networks for anomaly detection—primarily to identify attacks on a system. The paper [18] provides an overview of anomaly detection techniques and machine learning, and neural networks cover most of them. A detailed survey of learning approaches for anomaly detection is in [19]; particular approaches are, e.g., in [20–22].

There are also a number of closely related approaches that employ neural networks in adaptive systems in their analysis phase of the adaptation cycle. Typically, these approaches utilize neural networks to predict the best adaptation strategies. Namely, the approaches are as follows. In [7], neural networks are applied during the analysis and planning phase to reduce large adaptation space when the system has multiple adaptation goals and possible optimization strategies. In our method, we apply neural networks during the same phases, but our goal is to refine static conditions and thus allow more flexible adaptation.

In [6], a whole software engineering framework for adaptive systems is proposed. Neural networks are applied during the restriction of the adaptation space to achieve a meaningful system after adaptation. The approach in [4] is slightly different, as neural networks are employed on the boundary of monitoring and analysis phases of the adaptation loop. They are used to forecast future values of QoS parameters of a monitored system and thus allow for the progressive selection of the best adaptation strategy. A similar prediction is used in [23] to predict values in sensor networks and proactively perform adaptation. Multiple machine learning algorithms, including also neural networks, are employed in [24] to create a dynamic, self-adaptive, and online QoS modeling approach for cloud-based services. The approach is again used to predict QoS values and thus allows the optimization of cloud resource utilization.

The approaches above target either reducing the adaptation space or adapting a system proactively. They differ from our method since we use neural networks to refine static conditions in an adaptive system and thus to learn new unforeseen conditions. A conceptually similar approach is [25], where machine learning approaches are utilized for training a model for rule-based adaptation. Instead of NNs, approaches like random forest, gradient boosting regression models, and extreme boosting trees are used. Similarly, paper [26] proposes a proactive learner; however, the infrastructure is mainly discussed, and details about the used machine learning techniques are omitted. In [27], the authors propose an approach to dynamic learning of knowledge in self-adaptive and self-improving systems using supervised and reinforcement learning techniques.

In [28], machine learning is used to deal with uncertainty in an adaptive system (namely, in a cloud controller). Here, the proposed approach allows the users to specify potentially imprecise control rules expressed with the help of fuzzy logic, and machine learning techniques are used to learn

precise rules. The approach is the complete opposite of ours, where we start with precise rules and use machine learning to reach attunable ones. A similar approach is in [29], where reinforcement learning is also employed for generating and evolving the adaptation rules.

6 Conclusion

In this paper, we introduced the rule-specific Neural Network method that allows for transforming the guard of an adaptation rule into a custom neural network, the composition of which is driven by the structure of the logical predicates in the guard. An essential aspect of *rsNN* is that by having the ability to combine the original static predicates with attunable ones (and, in addition, to set the training capacity of the corresponding part of *rsNN* network), one can step-by-step proceed from a static, non-trainable adaptation rule to fully trainable one. This aspect allows for a gradual transition from the original self-adaptive system to its trainable counterpart, while still controlling the inherent uncertainty of introducing machine learning into the system.

The aspect of being able to control the uncertainty inherent in machine learning is a distinguishing factor of the *rsNN* method. This stems primarily from two facts: (1) The structure of the *rsNN* generated from an adaption rule directly relates to the composition of its predicates, and the static predicates can be combined with attunable ones. (2) An *rsNNs* is a neural network with almost two orders of magnitude fewer neurons than a generic neural network (e.g., a multilayer perceptron network with several hidden dense layers) solving the same task. This makes the *rsNN* less prone to overfitting, which, in general, may lead to unexpected results in real environments. Moreover, given the significant difference in the number of neurons and thus trainable weights, rsNN networks train much faster, as showcased in the results of the experiments.

The process described in this paper relies on the manual transformation of the predicates to rsNN, as our focus here is on introducing the rsNNs and defining their semantics. An important help in working with rsNNs would be some support for the semi-automated transformation of the static predicates into attunable ones. This is however a research problem on its own and thus its out of the scope of this paper. We leave this as future work.

Further, in our future work, we aim to extend the set of the predefined aa-predicates to provide a tool for applications also featuring other than metric and categorical quantities.

Funding This work has been funded by the DFG (German Research Foundation) – project number 432576552, HE8596/1-1 (FluidTrust), supported by the Czech Science Foundation project 20-24814J, partially supported by Charles University institutional funding SVV 260698/2023 and funding from the topic Engineering Secure Systems of the Helmholtz Association (HGF) and by KASTEL Security Research Labs, and partially supported by the Charles University Grant Agency project 408622.

References

1. Bureš, T., Hnětynka, P., Kruliš, M., Plášil, F., Khalyeyev, D., Hahner, S., Seifermann, S., Walter, M., Heinrich, R.: Attuning adaptation rules via a rule-specific neural network. In: Proceedings of ISOLA 2022, Rhodes, Greece, pp. 215–230 (2022)
2. Goodfellow, I., Bengio, Y., Courville, A.: Deep Learning. MIT Press, Cambridge (2016). http://www.deeplearningbook.org
3. Saputri, T.R.D., Lee, S.-W.: The application of machine learning in self-adaptive systems: a systematic literature review. IEEE Access **8**, 205948–205967 (2020)
4. Muccini, H., Vaidhyanathan, K.: A machine learning-driven approach for proactive decision making in adaptive architectures. In: Companion Proc. of ICSA 2019, Hamburg, Germany (2019)
5. Bureš, T., Gerostathopoulos, I., Hnětynka, P., Pacovský, J.: Forming ensembles at runtime: a machine learning approach. In: Proc. of ISOLA 2020, Rhodes, Greece (2020)
6. Gabor, T., Sedlmeier, A., Phan, T., Ritz, F., Kiermeier, M., Belzner, L., Kempter, B., Klein, C., Sauer, H., Schmid, R., Wieghardt, J., Zeller, M., Linnhoff-Popien, C.: The scenario coevolution paradigm: adaptive quality assurance for adaptive systems. Int. J. Softw. Tools Technol. Transf. **22**(4) (2020)
7. Van Der Donckt, J., Weyns, D., Quin, F., Van Der Donckt, J., Michiels, S.: Applying deep learning to reduce large adaptation spaces of self-adaptive systems with multiple types of goals. In: Proc. of SEAMS 2020, Seoul, Korea (2020)
8. Al-Ali, R., Bures, T., Hartmann, B.-O., Havlik, J., Heinrich, R., Hnetynka, P., Juan-Verdejo, A., Parizek, P., Seifermann, S., Walter, M.: Use Cases in Dataflow-Based Privacy and Trust Modeling and Analysis in Industry 4.0 Systems. Technical Report 1000085169, Karlsruher Institut für Technologie (2018) https://publikationen.bibliothek.kit.edu/1000085169
9. Weyns, D., et al.: Towards better adaptive systems by combining MAPE, control theory, and machine learning. In: Proc. of SEAMS 2021, Madrid, Spain (2021)
10. Hochreiter, S., Bengio, Y., Frasconi, P., Schmidhuber, J.: Gradient flow in recurrent nets: the difficulty of learning long-term dependencies. In: A Field Guide to Dynamical Recurrent Neural Networks (2001)
11. Mańdziuk, J., Macukow, B.: A neural network performing Boolean logic operations. Opt. Mem. Neural Netw. **2**(1), 17–35 (1993)
12. Schwenker, F., Kestler, H.A., Palm, G.: Three learning phases for radial-basis-function networks. Neural Netw. **14**(4) (2001)
13. Paper results replicaton package. https://github.com/smartarch/attuning-adaptation-rules-replication-package
14. Runeson, P., Höst, M.: Guidelines for conducting and reporting case study research in software engineering. Empir. Softw. Eng. **14**(2), 131–164 (2009)
15. Li, H., Qin, K., Xu, Y.: Dynamic neural networks for logic formula. In: Proc. of the 8th International Conference on Information Processing (2001)
16. Riegel, R., Gray, A., Luus, F., Khan, N., Makondo, N., Akhalwaya, I.Y., Qian, H., Fagin, R., Barahona, F., Sharma, U., Ikbal, S., Karanam, H., Neelam, S., Likhyani, A., Srivastava, S.: Logical Neural Networks (2020). arXiv:2006.13155 [cs]
17. Shi, S., Chen, H., Zhang, M., Zhang, Y.: Neural Logic Networks. (2019). arXiv:1910.08629 [cs, stat]
18. Mohammadi Rouzbahani, H., Karimipour, H., Rahimnejad, A., Dehghantanha, A., Srivastava, G.: Anomaly detection in cyber-physical systems using machine learning. In: Handbook of Big Data Privacy (2020)

19. Luo, Y., Xiao, Y., Cheng, L., Peng, G., Yao, D.D.: Deep learning-based anomaly detection in cyber-physical systems: progress and opportunities (2021). arXiv:2003.13213 [cs, eess]
20. Goh, J., Adepu, S., Tan, M., Lee, Z.S.: Anomaly detection in cyber physical systems using recurrent neural networks. In: Proc. of HASE 2017, Singapore (2017)
21. Kravchik, M., Shabtai, A.: Detecting cyber attacks in industrial control systems using convolutional neural networks. In: Proc. of CPS-SPC '18, Toronto, Canada (2018)
22. Su, Y., Zhao, Y., Niu, C., Liu, R., Sun, W., Pei, D.: Robust anomaly detection for multivariate time series through stochastic recurrent neural network. In: Proc. of KDD'19, East Lansing, MI, USA (2019)
23. Anaya, I.D.P., Simko, V., Bourcier, J., Plouzeau, N., Jézéquel, J.-M.: A prediction-driven adaptation approach for self-adaptive sensor networks. In: Proc. of SEAMS 2014, Hyderabad, India (2014)
24. Chen, T., Bahsoon, R.: Self-adaptive and online QoS modeling for cloud-based software services. IEEE Trans. Softw. Eng. **43**(5) (2017)
25. Ghahremani, S., Adriano, C.M., Giese, H.: Training prediction models for rule-based self-adaptive systems. In: Proc. of ICAC 2018, Trento, Italy (2018)
26. Bierzynski, K., Lutskov, P., Assmann, U.: Supporting the self-learning of systems at the network edge with microservices. In: Smart Systems Integration; 13th Int. Conf. and Exhibition on Integration Issues of Miniaturized Systems (2019)
27. Stein, A., Tomforde, S., Diaconescu, A., Hähner, J., Müller-Schloer, C.: A concept for proactive knowledge construction in self-learning autonomous systems. In: Proc. of FAS*W 2018, Trento, Italy (2018)
28. Jamshidi, P., Pahl, C., Mendonça, N.C.: Managing uncertainty in autonomic cloud elasticity controllers. IEEE Cloud Comput. **3**(3) (2016)
29. Zhao, T., Zhang, W., Zhao, H., Jin, Z.: A reinforcement learning-based framework for the generation and evolution of adaptation rules. In: Proc. of ICAC 2017, Columbus, OH, USA (2017)