Augmenting Deep Neural Networks with Scenario-Based Guard Rules

Guy Katz(✉)

The Hebrew University of Jerusalem, Jerusalem, Israel
guykatz@cs.huji.ac.il

Abstract. Deep neural networks (DNNs) are becoming widespread, and can often outperform manually-created systems. However, these networks are typically opaque to humans, and may demonstrate undesirable behavior in corner cases that were not encountered previously. In order to mitigate this risk, one approach calls for augmenting DNNs with hand-crafted *override rules*. These override rules serve to prevent the DNN from making certain decisions, when certain criteria are met. Here, we build on this approach and propose to bring together DNNs and the well-studied *scenario-based modeling* paradigm, by encoding override rules as simple and intuitive scenarios. We demonstrate that the scenario-based paradigm can render override rules more comprehensible to humans, while keeping them sufficiently powerful and expressive to increase the overall safety of the model. We propose a method for applying scenario-based modeling to this new setting, and apply it to multiple DNN models. (This paper substantially extends the paper titled "Guarded Deep Learning using Scenario-Based Modeling", published in Modelsward 2020 [47]. Most notably, it includes an additional case study, extends the approach to recurrent neural networks, and discusses various aspects of the proposed paradigm more thoroughly).

Keywords: Scenario-based modeling · Behavioral programming · Machine learning · Deep neural networks

1 Introduction

Deep learning technology [20] is bringing about dramatic changes in the world, by allowing engineers to use automated learning algorithms to create complex models [21]. Deep learning algorithms can generalize examples of how a desired system should behave into an artifact called a *deep neural network* (*DNN*). The DNN is then capable of correctly handling new inputs—including inputs that it had not encountered previously. In many cases, DNNs have been shown to *significantly outperform* manually-crafted software. Notable examples include the AlphaGO Go player [64], which defeated some of the world's strongest human Go players; systems for image recognition with DNN components that achieve superhuman precision [65]; and systems in various other domains, including recommender systems [16], natural language processing [12], and bioinformatics [10].

S. Hammoudi et al. (Eds.): MODELSWARD 2020, CCIS 1361, pp. 147–172, 2021.
https://doi.org/10.1007/978-3-030-67445-8_7

As DNNs are becoming more accurate and easier to create than manually-crafted systems, their use is expected to grow and intensify in the coming decades. Recently it has even been proposed to incorporate DNNs in *highly critical systems*, such as autonomous cars and unmanned aircraft [7,45].

Although DNNs have been demonstrating extraordinary performance, their use poses new challenges [1]. A notable difficulty is the extreme *opacity* of DNNs: because DNNs are generated by computers and not by humans, we can empirically determine that they perform well, but fully understanding their internal decision making is highly difficult. As a result, it is nearly impossible for humans to manually reason about the correctness of DNNs. For example, in many state-of-the-art systems for image recognition, which appeared highly accurate, it has been observed that slight input perturbations could cause DNN to make problematic misclassifications [69]. This phenomenon raises serious concerns about these networks' reliability and safety. In recent years initial attempts have been made to automatically reason about DNNs using formal methods (e.g., [19,41,48,50,54,71]), but these approaches currently afford only limited scalability. Moreover, DNN verification approaches typically focus on detecting erroneous DNN behaviors, but do not specify how to correct such behaviors after their discovery—which is also a difficult task.

As an illustrative example, consider the DeepRM system [57]. The goal of DeepRM is to perform resource allocation: the system has certain available resources (e.g., memory and CPUs), and also a queue of pending jobs. In each time step, the system needs to either schedule a pending job and allocate some of the available resources to this job; or perform a "pass" action, which means that no new jobs are assigned resources while the system waits for executing jobs to terminate and free up resources. DeepRM's goal is to perform scheduling in a way that maximizes job throughput. In order to achieve this goal, the system maintains a model that contains information about resource allocation and pending jobs, and uses a pre-trained DNN to choose which action to perform in each step. When compared to manually created state-of-the-art solutions that tackle the same problem, DeepRM has been shown to perform very well [57].

In spite of its overall satisfactory performance, it has been observed that the DeepRM system may sometimes behave in undesirable ways. For example, DeepRM's creators reported that its DNN controller might sometimes request that a job x be allocated resources, even though no job x exists in the job queue. In order to address this issue, *override rule* were added to DeepRM's implementation [58]. An override rule is a small piece of code that can examine the current state of the system, and then overrides the decision of the DNN controller when certain conditions are detected. In the case described above, the override rule will change the controller's selection to "pass" whenever the DNN requests to allocate resources to a job that is non-existent. There are additional override rules included within the DeepRM implementation [58], and also in implementations of other systems that use DNN controllers (e.g., the Pensieve system [59]). Further, additional undesirable behaviors have been discovered

in DeepRM since its release [52]; and removing these behaviors might require augmenting the system with yet additional override rules.

These cases, and others like them, demonstrate the integral role that override rules are beginning to play in DNN-based models. Because erroneous behaviors may be discovered after such models are deployed, override rules may need to be added, extended, refactored and enhanced at many points throughout the system's lifetime. In this paper, we argue that this situation calls for leveraging suitable modeling techniques, in order to facilitate the creation and maintenance of override rules—in a way that would increase the system's overall reliability.

As part of this work we advocate using the *scenario-based modeling* (*SBM*) framework [13,40] in creating override rules. In SBM, the individual behaviors of a system are modeled as independent scenarios, which are then automatically interwoven when the model is executed—in a way that produces cohesive system behavior. SBM has been shown to afford multiple benefits in the design and automated maintenance of systems. In addition, it is particularly suited for *incremental development*, which is a desirable trait when dealing with override rules. We propose here an approach and a method for applying SBM to systems with DNN components, in a way that allows engineers to specify override rules as SBM scenarios. We discuss the benefits that this approach affords (for example, through the amenability of SBM to automated analysis [38]), and demonstrate its applicability on three recently proposed systems. Although we focus here on systems with DNN components, our proposed approach could be adjusted to also accommodate systems with additional kinds of opaque components.

The rest of this paper is organized as follows. In Sect. 2 we provide the necessary background on SBM, DNNs and override rules. Next, in Sect. 3 we present our method for applying SBM to systems with DNN components. In Sect. 4 we describe how the proposed approach is applied to three case-studies. Next, in Sect. 5 we extend the proposed technique to recurrent neural networks. A discussion of related work appears in Sect. 6, and we conclude in Sect. 7.

2 Background

2.1 Scenario-Based Modeling

Scenario-based modeling [40] is a modeling approach for creating complex reactive systems. The basic notion at the core of this approach is that of a *scenario object*: an object that describes a single behavior, either desirable or undesirable, of the system being modeled. Each scenario object is created separately and independently of other scenarios, and does not directly interact with them; instead, it only interacts with a global execution mechanism. This global execution mechanism is the component in charge of managing the execution of a set of scenario objects, in a way that produces cohesive system behavior.

Several flavors of scenario-based modeling have been proposed, which differ from each other primarily in the idioms that a scenario object uses to interact with the execution mechanism, and thus to affect the overall system execution.

Our work here focuses on the most commonly used idioms, namely the *requesting*, *waiting-for* and *blocking* of events [40]. When the system is executed, each scenario object may declare that it has reached a *synchronization point*, in which the global execution mechanism must trigger an event. The object then specifies three sets of events: events that it *requests* be triggered; events that it *blocks* from being triggered; and events that it does not actively request, but should be notified in case they are triggered by the global execution mechanism (*waited-for events*). The execution mechanism waits for all the individual scenario objects to synchronize (or, alternatively, just for a subset thereof—depending on the semantics in use [27]). Then, it selects an event e that is requested and not blocked for triggering, and informs any relevant scenario object that e has been triggered.

An example of a small, scenario-based model appears in Fig. 1. This model belongs to a system that controls the water level in a tank that has hot and cold water taps. Each of the model's scenario objects is depicted as a transition system, whose nodes represent the (predetermined) synchronization points. The AddHotWater scenario object repeatedly waits for WaterLow events, and requests three times the event AddHot; and the AddColdWater scenario object performs a symmetrical operation with cold water. When a model that includes only the AddHotWater and AddColdWater objects is executed, three AddHot events and three AddCold events may be triggered in any order. If an additional requirement is added that the water temperature in the tank be kept stable, the scenario object Stability may be used to enforce the interleaving of AddHot and AddCold events through the use of event blocking. The execution trace that is generated by the resulting model appears in the event log.

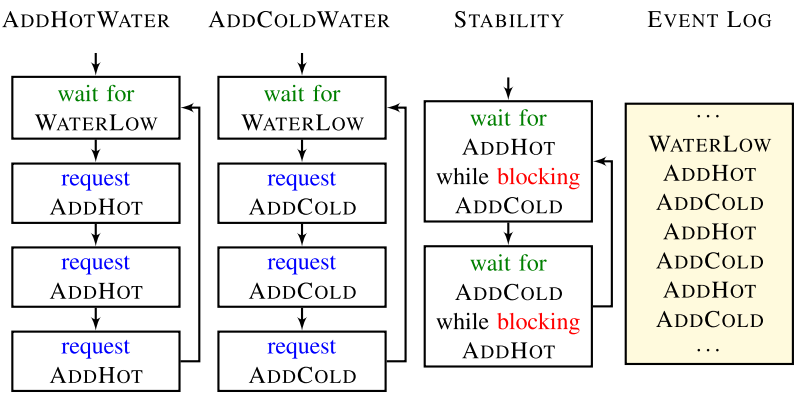

Fig. 1. (From [37]) A scenario-based model of a system that controls the water level in a tank with hot and cold water taps.

Scenario-based modeling has been implemented on top of a variety of programming languages, such as JavsScript [4], Java [39], ScenarioTools [22] and

C++ [30]. The SBM methodology has been successfully applied to model complex systems, such as robotic controllers [25], cache coherence protocols [32] and web-servers [30]. For simplicity, in the rest of this paper, we often describe scenario objects in terms of transitions systems.

We take after the definitions given in [46], and formalize the SBM framework as follows. A scenario object O over event set E is defined as tuple $O = \langle Q, \delta, q_0, R, B \rangle$, which is comprised of the following components:

- Q is a set of states, each representing one of the predetermined synchronization points;
- q_0 is the initial state;
- $R : Q \rightarrow 2^E$ and $B : Q \rightarrow 2^E$ map states to the sets of events requested and blocked at these states (respectively); and
- $\delta : Q \times E \rightarrow 2^Q$ is a transition function, indicating how the object reacts when an event is triggered.

Two scenario objects can be composed into a single, combined scenario object, as follows. For objects $O^1 = \langle Q^1, \delta^1, q_0^1, R^1, B^1 \rangle$ and $O^2 = \langle Q^2, \delta^2, q_0^2, R^2, B^2 \rangle$ over a common event set E, we define the composite scenario object $O^1 \parallel O^2$ as $O^1 \parallel O^2 = \langle Q^1 \times Q^2, \delta, \langle q_0^1, q_0^2 \rangle, R^1 \cup R^2, B^1 \cup B^2 \rangle$, where:

- $\langle \tilde{q}^1, \tilde{q}^2 \rangle \in \delta(\langle q^1, q^2 \rangle, e)$ if and only if $\tilde{q}^1 \in \delta^1(q^1, e)$ and $\tilde{q}^2 \in \delta^2(q^2, e)$; and
- The union of the labeling functions is defined in the natural way; e.g. $e \in (R^1 \cup R^2)(\langle q^1, q^2 \rangle)$ if and only if $e \in R^1(q^1) \cup R^2(q^2)$, and $e \in (B^1 \cup B^2)(\langle q^1, q^2 \rangle)$ if and only if $e \in B^1(q^1) \cup B^2(q^2)$.

The composition operator can be applied repeatedly to compose any number of scenario objects into a single scenario object.

We define a *behavioral model* M to be a collection of scenario objects, $O^1, O^2, \ldots, O^n$. The executions of M are defined to be the executions of the composite scenario object, $O = O^1 \parallel O^2 \parallel \ldots \parallel O^n$. Each execution of M starts from the initial state of O; and in each state q visited throughout the execution an enabled event e is chosen for triggering, if such an event exists (i.e., $e \in R(q) - B(q)$). Then, the execution proceeds to a state $\tilde{q} \in \delta(q, e)$, and so on.

Several extensions have been proposed for the basic variant of SBM described above. In one such extension, which will be particularly useful in our context, events are treated as *typed variables* [51]. For example, an event e can be declared to be of type integer, allowing a scenario object to *request* $e \geq 5$. Another scenario object might block $e \geq 7$. In this setting, the execution framework employs a *constraint solver*, such as an SMT solver [5], in order to resolve the various constraints and find a value assignment for e. In this case, the event $e = 6$ might be triggered. We omit here the formal definition of this extension, which is straightforward; the interested reader is referred to [51].

2.2 Deep Neural Networks and Override Rules

Deep, feed-forward neural networks (DNNs) are directed, weighted graphs, in which the nodes (also known as *neurons*) are organized into layers. The first

and last layers are the input and output layers, respectively, and the multiple remaining layers are referred to as hidden layers. Each neuron in the network (except for input neurons) is connected to neurons from the preceding layer, and each edge is assigned a predetermined weight value (an illustration appears in Fig. 2). The selection of appropriate weight values is key; this is performed when the DNN is created during the *training* phase, which goes beyond the scope of this paper (for additional details, see [20]). In order to evaluate the DNN, values are first assigned to its input neurons, and then propagated forward through the network in an iterative process. In each iteration, values for another layer are computed using the values assigned to neurons in its predecessor. Eventually, the values of the output neurons are computed, and these values constitute the outputs of the DNN which are returned to the user. It is typical for DNNs to be used as controllers or classifiers, in which case the user usually cares about which output neuron received the highest value—as this neuron represents the action, or classification, that the DNN has selected among the possible options.

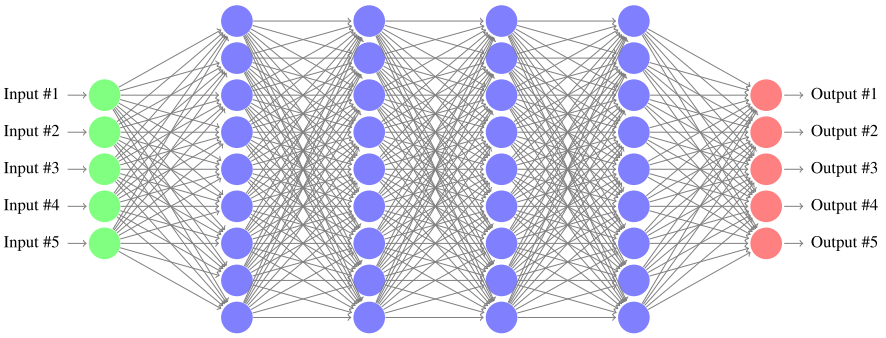

Fig. 2. (From [47]) A fully connected DNN with 5 input nodes (in green), 5 output nodes (in red), and 4 hidden layers containing a total of 36 hidden nodes (in blue). (Color figure online)

For our purpose here, it is usually sufficient to treat DNNs as black boxes, that transform an input into an output in some unknown way. However, for completeness, we briefly describe the evaluation procedure of a DNN. After the input neurons are assigned values, the value of each hidden node is computed in two steps: first, we compute a weighted sum of the node values from the previous layer, according to the predetermined edge weights. Then, we apply a non-linear *activation function* to this weighted sum [20], and the output of this activation function becomes the value of the node being computed. One common activation function is the Rectified Linear Unit (ReLU) [61], computed by $\text{ReLU}(x) = \max(0, x)$. Thus, when a neuron's value is computed using the ReLU activation function, it is taken to be the maximum between the linear combination of node values from the previous layer and 0.

Figure 3 depicts a small DNN (with 7 neurons in total), which will serve as a running example. This DNN acts as a controller: it takes two inputs, x_1 and x_2,

computes values for its three hidden neurons v_1, v_2 and v_3, and then computes its output values y_1 and y_2, which represent scores for two possible actions. The hidden nodes v_1, v_2 and v_3 all use the ReLU activation function. We slightly abuse notation here, and use y_1 and y_2 to denote both the neurons and the actions/classes represented by these neurons. The action that is assigned the higher score is the one selected by the DNN. For example, the input assignment $x_1 = 1, x_2 = 0$ results in output values $y_1 = 1, y_2 = 0$, which mean that action y_1 is selected. In contrast, the input assignment $x_1 = 0, x_2 = 1$ leads to $y_1 = 0, y_2 = 3$, and so action y_2 is selected.

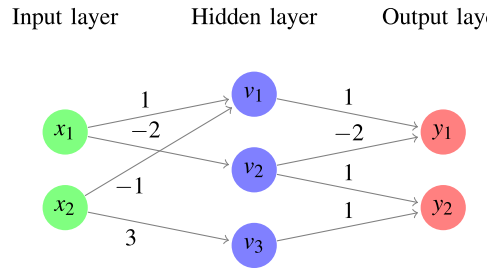

Fig. 3. (From [47]) A small neural network with a single hidden layer.

We formalize the notion of an *override rule* as a triple $\langle P, Q, \alpha \rangle$, where: (i) P is a predicate over the inputs of the network; (ii) Q is a predicate over the outputs of the network; and (iii) α is an override action. The semantics of an override rule $\langle P, Q, \alpha \rangle$ is that whenever P and Q hold for a network's evaluation, then output action α should be the one selected, regardless of the actual output of the DNN. For example, consider the following override rule

$$\langle x_1 > 0 \wedge x_2 < x_1, true, y_2 \rangle.$$

As we saw previously, for input values $x_1 = 1, x_2 = 0$ the DNN normally selects y_1; but, with this override rule, the selection would be changed instead to y_2. Note that this is so because this particular input satisfies the input condition, i.e. it holds that $x_1 > 0$ and $x_2 < x_1$. Our choice of setting Q to true means that this override rule only examines the DNN's inputs, and does not depend on its outputs. If we were to set Q, e.g., to $y_2 > 10$, then the override rule would not be triggered for $x_1 = 1, x_2 = 0$. By adjusting P and Q as needed, this definition is sufficient for expressing many common override rules, such as those in the DeepRM example described in Sect. 1.

3 Modeling Override Scenarios

In the case of DeepRM, engineers have added override rules as unrestricted Python code that resides within the code module that invokes the DNN controller, and then processes its result [58]. Thus, while the DNN component itself

is clearly structured and well defined, the more recently added override rules are expressed as arbitrary pieces of code. This coding convention could lead to several undesirable issues: (i) if the number of override rules was to increase significantly, they could become convoluted and difficult to comprehend, maintain and extend; (ii) the semantics of existing override rules might, in time, become unclear. For example, does a more recent override rule supersede a previous rule if both can be applied? Is there a particular order in which override rules should be checked? Can multiple rules interact with one another? etc; and (iii) the conditions encoded within override rules might, in time, become more complex than originally intended, thus hiding away some of the model's logic where engineers might not know to look for it.

Here, we advocate the modeling of override rules using SBM, in a way that is designed to mitigate the aforementioned difficulties. SBM is particularly geared towards incremental modeling, which is a likely scenario when DNNs are involved: because DNNs are opaque, some of their undesirable behaviors are likely to be detected only post-deployment, thus requiring that new override rules be added. Moreover, SBM's simple semantics serve to guarantee that all interactions between the override rules are well defined. Finally, a substantial amount of work has been carried out on automatically analyzing, verifying and optimizing SBM models; and building on top of this work could prove useful in simplifying override rules and in detecting conflicts between them, as their numbers increase.

3.1 Modeling DNNs and Override Rules in SBM

We now propose a method for creating SBM models, in a way that combines scenario objects with a DNN controller. The core idea is to use a dedicated scenario object, O_{DNN}, to abstractly represent the DNN within the scenario-based model. This O_{DNN} is a non-deterministic scenario that models the DNN controller, and allows it to interact with other scenario objects which are present in the system. For the sake of simplicity, for now we assume that the set of possible inputs to the DNN, denoted $\mathbb{I}$, is finite (we relax this limitation later). Let $\mathbb{O}$ denote the set of possible actions from which the DNN chooses. We add the following new events to our event set E: an *input event* e_i for every $i \in \mathbb{I}$, and an *output event* e_o for every $o \in \mathbb{O}$. We introduce the convention that our new scenario object O_{DNN} repeatedly waits for all input events e_i, and then request all output events e_o. This behavior represents the black-box nature of the neural network component, at least as far as the rest of the model is concerned: engineers only know that after an input event is triggered, one of the output events will be selected, without knowing which. However, when the model is deployed, the execution infrastructure resolves the non-determinism of O_{DNN} by invoking the actual DNN and triggering precisely the output event that corresponds to the DNN's selection. For example, assuming there are only precisely two possible inputs, e.g. $i_1 = \langle 1, 0 \rangle$ and $i_2 = \langle 0, 1 \rangle$, the DNN depicted in Fig. 3 would be represented by the O_{DNN} scenario object that appears in Fig. 4.

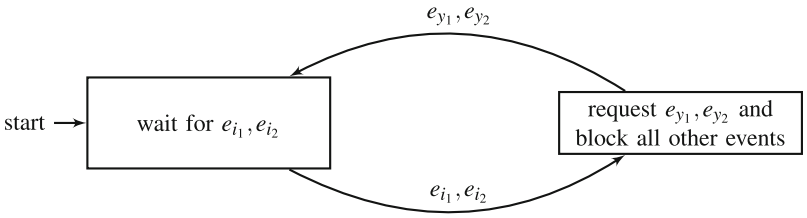

Fig. 4. (From [47]) A scenario O_{DNN} for the neural network in Fig. 3. Events e_{i_1} and e_{i_2} represent the inputs to the neural network, and events e_{y_1} and e_{y_2} represent its outputs.

In order to render the resulting model compatible with the actual DNN, we introduce a convention that states that other scenario objects in the system may not block any input event e_i. These scenario objects may, however, wait-for these events. A single dedicated scenario object, called a *sensor*, is responsible for requesting an input event when it is time for the DNN needs to be evaluated (e.g., following some user action). By another convention that we introduce, no scenario object besides O_{DNN} may request any output event e_o; however, other scenario objects in the system may wait-for, or block, these events. During system execution, it is possible for the neural network to assign the highest output score to an event that is currently blocked by another scenario object. When this happens, we resolve the non-determinism of O_{DNN} by selecting another output event, which represents the output with the next-to-highest score, etc. If no output events are left unblocked, then the system is deadlocked—and the execution terminates.

The motivation that underlies our definitions is to allow various scenario objects to monitor the inputs and outputs of the neural network controller, by waiting for the input and output events associated with them; and then to interfere with the DNN's recommendation by blocking certain output events. This is precisely the use-case of a typical override rule. We note that a scenario object may force the neural network to produce some specific output, by blocking all other possible outputs; alternatively, it may interfere more subtly, by blocking some events while allowing the DNN to choose among the remaining, unblocked events.

Recall our earlier assumption that the sets $\mathbb{I}$ and $\mathbb{O}$ of possible DNN inputs and outputs, respectively, are finite. In practice, this assumption might become a limiting factor: for example, considering the override rule described in Sect. 2.2, the triggering of the override rule was affected by the values assigned to x_1 and x_2, and so it is desirable to express these exact values in our model. Of course, in this case the number of possible value assignments is infinite. To overcome this limitation we again turn to an extension of the SBM semantics [51], which allows engineers to treat events as typed variables. We adjust our formulation slightly: we allow scenario objects in the system to wait-for a single, composite event, whose triggering indicates that values have been assigned to (all of) the

neural network's inputs or outputs. Scenario objects may then access the fields of this composite event, which indicate the individual values assigned to each input or output neuron, and act according to these values.

With this extension in place, the override rule from Sect. 2.2 can be expressed as the scenario object in Fig. 5. This scenario enforces the following override rule: whenever $x_1 > 0$ and $x_2 < x_1$, output event y_2 (and not y_1) should be triggered. Here, the tuple $\langle e_{x_1}, e_{x_2} \rangle$ represents a single event, whose triggering indicates that values have been assigned to the neural network's inputs. This is a composite event that contains two real values, x_1 and x_2, that the override scenario can access and use in order to determine its next state. Output event e_{y_1} indicates, as before, that the override scenario forbids the neural network from selecting y_1 as its output action.

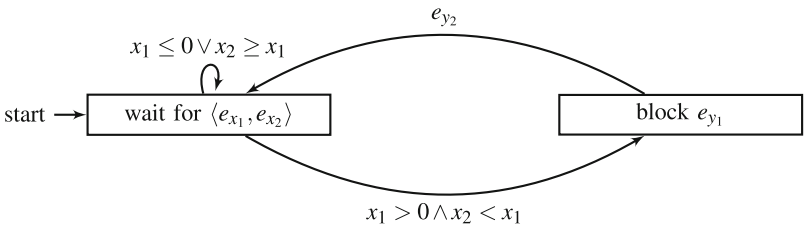

Fig. 5. (From [47]) A scenario object enforcing the override rule that whenever $x_1 > 0$ and $x_2 < x_1$, output event y_2 should be triggered.

3.2 Liveness Properties

Override rules are most often used to enforce *safety properties*. These properties state that "bad things never happen". However, sometimes there is a need to enforce also *liveness* properties, which state that "good things eventually happen". Specifically, this need can arise in the context of online reinforcement learning [68], in which the DNN controller changes over time. In this context we may wish to ensure, for example, that the DNN controller eventually tries out new output actions. If these output actions prove beneficial, the online RL mechanism will ensure that the neural network controller repeats them in the future. Liveness properties are relevant also when there are *fairness constraints*; for example, we may wish to ensure that in a resource management system, every pending job is eventually scheduled.

An example in which we wish to enforce liveness properties appears in [52], where the authors describe the *Custard* system: a congestion control system, which uses a neural network controller. Custard monitors the conditions of a computer network, and then select a bitrate for sending information across this network—with the goal of minimizing congestion while maintaining high throughput [43]. In [52], the authors examine Custard in order to determine whether there exist cases in which the DNN controller chooses a *sub-optimal*

sending rate, i.e. a sending rate that does not utilize all available bandwidth, and never attempts to increase this bitrate. Clearly, such behavior constitutes a liveness violation, which can be corrected using an override rule.

Using SBM, we can encode the fact that one (or multiple) DNN output actions should eventually be blocked. Because blocking some actions forces the DNN controller to pick a different action, it can be used to enforce a liveness property. In practice, this blocking can be performed by having a scenario object wait for a sequence of n consecutive rounds in which a particular output event is triggered, and then block it in round $n+1$. An example for $n=3$ appears in Fig. 6: the scenario therein looks for 3 consecutive DNN evaluations where event y_2 is triggered, after which it blocks y_2 once, forcing the neural network to select another output action. An alternative approach is to have the override scenario block a particular output event with a very low probability [37], thus eventually blocking that event with probability 1.

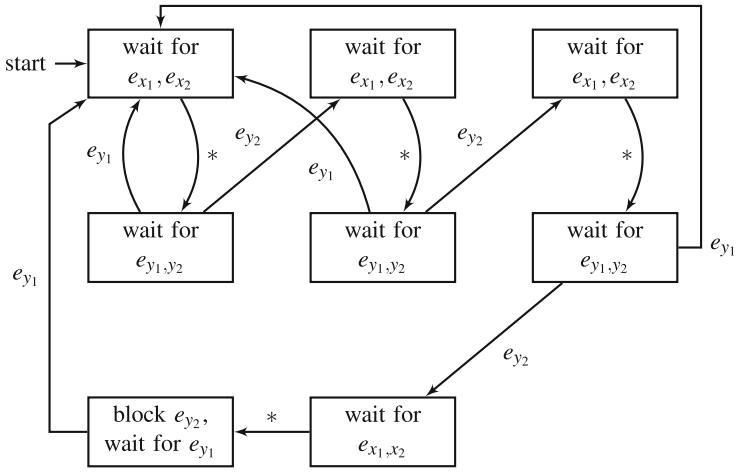

Fig. 6. (From [47]) A scenario object that enforces a liveness property for the network from Fig. 4.

3.3 Automated Analysis

Various studies indicate that using scenario-based modeling may serve to facilitate automated formal analysis (e.g., [38]). More specifically, the simple synchronization constructs employed by SBM scenario objects to communicate with each other render tasks such as automated repair [46], compositional verification [28] and model checking [49] simpler than they would be for less restricted models. We argue that the amenability of SBM to formal analysis adds to its attractiveness as a formalism for expressing override rules.

We illustrate this claim through one particular use case that involves deadlock freedom. As a DNN-based system is deployed and additional override rules

are slowly added to it, potentially by different engineers, we run the risk that a certain sequence of inputs to the DNN controller could cause a deadlock. A simple illustrative example appears in Fig. 5: this figure depicts an override rule that enforces that whenever $x_1 > 0$ and $x_2 < x_1$, output y_2 should be selected. Suppose now that at some later point in time, a different engineer is concerned about the possibility that the DNN might always advise y_2. This engineer then creates a new override rule, depicted in Fig. 6, to the effect that after 3 consecutive y_2 events, a different event must be triggered. When run simultaneously, these two override rules could produce a deadlock: for example, if the neural network is given the inputs $x_1 = 2, x_2 = 1$ three consecutive times, both override rules would be triggered, causing output events e_{y_1} and e_{y_2} to be simultaneously blocked.

The absence of such deadlocks can be guaranteed through the use of formal verification. The verification process can be carried out, e.g., after the addition of each new override rule, or on a periodic basis. Whenever a deadlock is detected, the counter-example provided by the verification tool could help the modeler in identifying and altering the conflicting override rules—after which verification can be run again, in order to ensure that the system is now indeed deadlock free. Clearly, additional system-specific properties, beyond deadlock freedom, could also be formally verified.

4 Three Case-Studies

In order to evaluate our approach, we implemented it on top of the BPC framework for scenario-based modeling in C++ [30] (other SBM frameworks could, of course, be used instead). The BPC package allows engineers to leverage many of the useful and expressive constructs of C++, while enforcing that they adhere to the SBM principles: i.e., each scenario is modeled using a separate object, and inter-scenario interactions are performed strictly through the global execution mechanism provided by BPC. Here, we used BPC to model override rules for the DeepRM system for resource management [57], the Pensieve system for adaptive bitrate selection [59], and the Custard system for congestion control [43].

4.1 Override Rules for DeepRM

The DeepRM system [57] (mentioned in Sect. 1) is a resource allocation system: it assigns available resources to pending jobs, in order to maximize job throughput. In order to evaluate our approach we implemented an override rule that prevents DeepRM's DNN controller from attempting to assign resources to non-existent jobs, which is undesirable system behavior that occurs in practice [58].

The BPC code for an override rule that addresses this situation, implemented as a scenario object, appears in Fig. 7. We assume here that the queue of pending jobs is of length 5, and we use $y_0, y_1, \ldots, y_5$ to denote the DNN's output actions. Output actions $y_1, \ldots, y_5$ indicate that the job in slot i of the queue should be selected for resource allocation, whereas the special action y_0 is the "pass"

action—which indicates that no job should be allocated resources at this time. We denote by x an event that indicates that the neural network needs to be evaluated on certain input values, which are available as parameters of x. The job queue's state is included in the input to the DNN controller. Specifically, we use $x[1], \ldots, x[5]$ to denote Boolean values that indicate whether or not there is currently a pending job in the corresponding slot of the queue.

Our override scenario object is implemented as a single class, which inherits from BPC's special BThread class. The override scenario object uses the special bSync() method in order to indicate that it has reached a synchronization point, and wishes to synchronize with the other scenarios in the model (including O_{DNN}, the special scenario object that models the DNN controller). The bSync() method takes as input three Event vectors—the first with the set of requested events, the second with the set of waited-for events, and the third with the set of blocked events. The bSync() call then suspends the object's execution, until the BPC execution mechanism has selected and triggered an event. Then, if the event that was triggered was requested or waited-for by a scenario object, that scenario is woken up and resumes its execution. In that case, the scenario object can also retrieve the triggered event using the lastEvent() method.

Our override scenario object runs in an infinite loop. In each iteration it synchronizes and waits for the input event x to be triggered; and once that triggering has occurred, the scenario examines x to determine which slots of the job queue are occupied. Finally, it synchronizes once again, in order to block event y_i for any unoccupied slots. Note that this scenario object can never cause a deadlock, because it never blocks the special "pass" event, y_0.

4.2 Override Rules for Pensieve

In online video streaming, a client wishes to download a video from a server and play it. The video is typically available in multiple levels of quality, known as *definitions*, that the client can choose from. The typical client will attempt to choose the highest definition that is reasonable for the current bandwidth conditions—i.e., the highest definition for which the video can be viewed without pauses for *rebuffering*, which are known to be detrimental to the viewer's experience. Further, bandwidth conditions might change while the video is being downloaded and played (e.g., if additional users start using the same physical link), in which case the choice of definition might need to be adjusted. An algorithm for selecting the definition rates in which a video is to be downloaded is called an *adaptive bitrate* (*ABR*) algorithm. Recently, DNN-based ABR algorithms have been shown to perform exceedingly well when compared to manually designed solutions [59].

The Pensieve system [59] is one such DNN-based ABR system. The system's goal is straightforward: given previous bitrate choices and statistics about how successful they were (i.e., how quickly parts of the video, called *chunks*, have previously been downloaded), the system selects the bitrate in which the next chunk is to be downloaded. Internally, Pensieve employs a DNN controller that takes as input: (i) a list of past bitrate selections; (ii) a list of past throughput

```
class EnsureJobExists : public BThread {
  void entryPoint() {
    Vector<Event> emptySet = {};
    Vector<Event> allInputs = { x };
    Vector<Event> allOutputs = { y0,...,y5 };

    while ( true ) {
      bSync( emptySet, allInputs, emptySet );
      lastInput = lastEvent();
      Vector<Event> blocked = {};

      for ( int i = 1; i <= 5; ++i ) {
        if ( !lastInput[i] )
          blocked.append( yi )
      }

      bSync( emptySet, allOutputs, blocked )
    }
  }
}
```

Fig. 7. (From [47]) A scenario object for preventing the DeepRM DNN controller from assigning resources to non-existing jobs.

rates (indicating how quickly past chunks were downloaded); (iii) the number of remaining video chunks to be downloaded; and (iv) the current *buffer size*, which indicates how many seconds of already-downloaded content are available for playing, before rebuffering occurs. The DNN controller has a fixed number of outputs (6, in our case study), each corresponding to a possible definition in which the next chunk can be downloaded; and the definition associated with the output to which the DNN assigns the highest score is the one selected for the next video chunk.

Despite Pensieve's overall excellent performance [59], formal verification of this system has recently revealed many corner cases in which it makes undesirable bitrate selections [52]. For example, consider the following properties:

- When there is a single video chunk left to download, the client's buffer is quite full, and all recently downloaded video chunks were downloaded in the highest definition available (HD), the last chunk should be downloaded in HD.
- When there is a single video chunk left to download, the client's buffer is nearly empty, and all recently downloaded video chunks were downloaded in the lowest definition available (SD), the last chunk should be downloaded in SD.

Both properties describe extreme cases, in which the correct choice of bitrate is clear: either conditions are excellent and so the best definition should be used,

or conditions are so poor that the worst definition should be used. However, even for these simple properties, dozens of violations (i.e., cases where the DNN selects some other definition) have been discovered [52].

As part of our evaluation, we use scenario-based override rules to enforce correct system behavior in both of these cases. To this end, we introduce scenario objects that wait until there are n chunks left in the video; and then monitor whether they are all downloaded in a fixed definition d. Then, if all chunks except for the very last one have been downloaded in definition d, the blocking idiom is applied to enforce that definition d is selected also for the last chunk. See Fig. 8 for an illustration. Of course, this override rule may be enhanced to include additional criteria (e.g., constraints on the client's buffer size) before the blocking is applied.

4.3 Override Rules for Custard

As we briefly mentioned in Sect. 3.2, Custard is a DNN-based system for congestion control. Custard's DNN controller receives as input various readings about the current, and previous, state of the computer network (e.g., loss rates, throughputs and latency readings). Then, it selects the next sending bit rate. Custard is a reactive system, in the sense that it was designed to run continuously and use the results of its past decisions (as they are reflected in past network readings) in order to make its next choice of bitrate.

Due to the opacity of Custard's DNN controller, one concern is that it might make selections that are overly *conservative*. Specifically, we typically wish to avoid a situation in which the state of the computer network is completely steady, and yet Custard's DNN controller never tries to increase the sending bitrate—and consequently never finds out whether some of the available bandwidth is currently unused.

Figure 9 depicts a scenario object that prevents the situation described above. This scenario attempts to identify situations in which the DNN's inputs and outputs have been completely steady for the last $n = 10$ rounds. Once this situation is detected, the scenario object blocks the previous output action from being triggered again, forcing the DNN to try an alternative. Note that event x represents here an input assignment (which is comprised of multiple input values) on which the neural network has been evaluated; whereas event y represents the DNN's output selection. For simplicity, we do not examine here the actual values of x, and instead only look for steady, repeating assignments (however, in practice we may wish to apply this override rule only if the computer network's conditions are both *steady* and *good*, which serves to indicate that there may be additional, unused bandwidth).

5 Recurrent Neural Networks

5.1 Memory Units

So far, we have focused on models that incorporate *feed-forward* neural networks. These networks, described in Sect. 2.2, are designed so that each of their

```
const int n = 10;

class EnsureLastChunkDefinition : public BThread {
  void entryPoint() {
    Vector<Event> empty;
    Vector<Event> allInputs = { x };
    Vector<Event> allOutputs = { y };

    Event lastInput;
    Event lastOutput;

    bool steadyState = false;

    int d;

    while ( true ) {
      bSync( empty, allInputs, empty );
      lastInput = lastEvent();

      if ( lastInput.remainingChunks > n )
        continue;

      else if ( lastInput.remainingChunks == n ) {
        steadyState = true;
        bSync( empty, allOutputs, empty );
        lastOutput = lastOutput();
        d = lastOutput.definition;
        continue;
      }

      else if ( lastInput.remainingChunks == 1 ) {
        if ( steadyState ) {
          bSync( empty, empty, allOutputs.erase( d ) );
          steadyState = false;
        }
        continue;
      }

      else {
        bSync( empty, allOutputs, empty );
        lastOutput = lastOutput();
        if ( steadyState && lastOutput.definition != d )
          steadyState = false;
      }
    }
  }
}
```

Fig. 8. A scenario object for forcing the Pensieve DNN to maintain the same definition for the last chunk.

```
const int n = 10;

class PreventSteadyState : public BThread {
  void entryPoint() {
    Vector<Event> empty;
    Vector<Event> allInputs = { x };
    Vector<Event> allOutputs = { y };

    Event lastInput;
    Event lastOutput;

    while ( true ) {
      bSync( empty, allInputs, empty );
      lastInput = lastEvent();

      bSync( empty, allOutputs, empty );
      lastOutput = lastOutput();

      bool steadyState = true;
      int i = 1;
      while ( i < n && steadyState ) {
        bSync( empty, allInputs, empty );
        if ( lastInput != lastEvent() )
          steadyState = false;

        bSync( empty, allOutputs, empty );
        if ( lastOutput != lastEvent() )
          steadyState = false;

        ++i;
      }

      if ( steadyState ) {
        bSync( empty, allInputs, empty );
        bSync( empty, allOutputs, lastOuptut );
      }
    }
  }
}
```

Fig. 9. (From [47]) A scenario object for enforcing the Custard DNN to choose a different action if the state has been steady for $n = 10$ iterations.

evaluations is independent of previous evaluations. This is suitable, for example, in image recognition: each image is classified independently, regardless of how images encountered previously were classified. However, this kind of neural network might be ill-suited for certain tasks that require *context*. Consider, for example, a DNN designed to interpret words that form a sentence, which are

passed to the DNN one word at a time. As the DNN reads a word, it must consider the previous words in the sentence in order to properly interpret its meaning.

To address this need, the machine learning community has designed a variant of deep neural networks called *recurrent neural networks* (*RNNs*) [20]. Much like its feed-forward counterpart, an RNN is evaluated each time on a set of input values and produces a set of output values. However, it also maintains, using internal *memory units*, some aggregated information from the previous evaluations. This stored information affects the future evaluations of the RNN. RNNs have proven remarkably useful for a variety of tasks that involve context, such as machine translation [15], health applications [56], and speaker recognition [70].

We demonstrate the concept of an RNN through a simple example, depicted in Fig. 10. This network has two input nodes, x_1 and x_2, two output nodes, y_1 and y_2, and a single hidden node v. The new construct is the memory unit, $\tilde{v}$, which is connected to v. When the network is evaluated on input $\langle x_1, x_2 \rangle$, it computes the output $\langle y_1, y_2 \rangle$ using weighted sums and activation functions, same as before. However, the value stored in the memory unit also participates in this computation; and once the evaluation is performed, the value computed for node v is stored in $\tilde{v}$, to be used in the next evaluation. By convention, we assume that the memory unit is first initialized to 0. Suppose the network is initially evaluated on input $\langle x_1, x_2 \rangle = \langle 1, 0 \rangle$; for this input, $v = 1$ and $\langle y_1, y_2 \rangle = \langle 1, 2 \rangle$. The value $v = 1$ will now be stored in $\tilde{v}$ for the next evaluation. Next, if the network is again evaluated on $\langle 1, 0 \rangle$, the new value computed for v will be 2, and now this value will be stored in $\tilde{v}$; and the network's outputs will be $\langle 2, 4 \rangle$. It is straightforward to show that the memory unit in this particular RNNs computes the sum of the ReLUs of all previously received x_1 values.

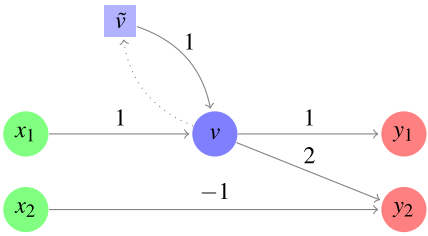

Fig. 10. A recurrent neural network.

5.2 Undesirable Behaviors in RNNs

Much like with feed-forward neural networks, various models that incorporate RNN components have been shown to demonstrate undesirable behavior. One common example is that of *adversarial inputs*—inputs that the network classifies correctly, but which, when they are slightly perturbed in subtle ways, cause the network to make severe misclassification errors [69]. Adversarial inputs are

mostly known to plague feed-forward neural networks that perform image recognition tasks [55,69], but recently they have also been shown to exist in RNNs; for example, slight perturbations to audio files, which are inaudible to the human ear, were shown to cause RNN misclassification [11].

These errors, and others, indicate that RNN-based models suffer from the same intrinsic drawbacks of feed-forward networks: although they perform well in general, they may behave in undesirable ways in some cases; and because they are completely opaque to the human eye, manually maintaining, extending and correcting them is impractical. The verification community has also observed this and has begun devising techniques for RNN verification [42,74]. However, just like in the feed-forward case, these techniques can detect a bug but do not provide a framework for removing bugs after they are detected. It is thus highly likely that as RNN-based models continue to be deployed in various systems, override rules will need to be added to these models.

5.3 Override Rules for RNNs

We extend our previous notion of an override rule to the RNN setting, as follows. We define an *RNN override rule* as the quadruple $\langle P, M, Q, \alpha \rangle$, where: (i) P is a predicate over the inputs of the network; (ii) M is a predicate over the memory units of the network; (iii) Q is a predicate over the outputs of the network; and (iv) α is an override action. The definitions of P, Q and α are as before, but we now include a fourth element, the predicate M, which can render the activation of the override rule conditional on the state of the RNN's memory units. The semantics of an override rule $\langle P, M, Q, \alpha \rangle$ is that whenever P, M and Q hold for a network's evaluation, then output action α should be the one selected, regardless of the actual output of the RNN.

We demonstrate with an example, Consider again the RNN depicted in Fig. 10, and the following override rule:

$$\langle x_1 > 0, \tilde{v} > 0, true, y_1 \rangle.$$

As we saw previously, for input values $x_1 = 1, x_2 = 0$ the RNN outputs $y_1 = 1, y_2 = 2$, and so y_2 is selected. At this point, the override rule is not triggered: although $x_1 > 0$, the predicate $M = (\tilde{v} > 0)$ does not initially hold, because $\tilde{v} = 0$. If the network is again evaluated on $x_1 = 1, x_2 = 0$, it would normally compute $y_1 = 2, y_2 = 4$ and select y_2; however, now $\tilde{v} = 1$, the predicate M is satisfied, and so the override rule is triggered and the network is forced to select y_1 instead.

5.4 Modeling RNN Override Rules in SBM

Similarly to the feed-forward case, we propose SBM as an attractive paradigm for modeling RNN override rules. We achieve this by again representing the RNN using a dedicated, non-deterministic scenario, O_{RNN}. This scenario repeatedly waits for a composite event that represents an assignment to the RNN's *inputs*

and also to its *memory units*; and then it requests all possible composite events, each of which represents a possible evaluation of the RNN's outputs. The intention is, once more, to simulate the black-box nature of the RNN: we do not allow the rest of the model to affect (i.e., block) the values of the RNN's inputs or memory units, but we allow it to observe (wait for) these values and affect the RNN's output values. When the system is deployed, the non-determinism of O_{RNN} is resolved using the actual input values that the RNN is given, and the actual values stored in its memory units at that time.

Using this formulation, override rules for the RNN case can again be expressed as scenario objects. We demonstrate this for the override rule discussed before, namely

$$\langle x_1 > 0, \tilde{v} > 0, true, y_1 \rangle,$$

whose corresponding override scenario is depicted in Fig. 11. The tuple $\langle e_{x_1}, e_{x_2}, e_{\tilde{v}} \rangle$ represents a single composite event, whose triggering indicates that values have been assigned to the neural network's inputs and memory unit. This composite event contains three real values, x_1, x_2 and $\tilde{v}$, that the override scenario can access and use in order to determine its next state. As before, the blocking of output event e_{y_2} indicates that the override scenario forbids the selection of y_2 as the RNN's output action.

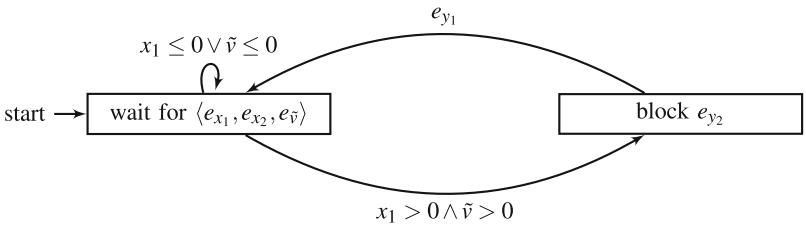

Fig. 11. An override rule for an RNN.

The same desirable properties that we discussed for the feed-forward case carry over to RNNs; i.e., (i) RNN override scenario can be used to encode both safety and liveness override rules; and (ii) automated SBM analysis can be used to ensure the consistency of override rules.

6 Related Work

Override rules, which are sometimes also referred to as *shields*, have been applied ad-hoc in various DNN-enabled systems. Some examples, which we have already mentioned, include DeepRM [57] and Pensieve [59]. Override rules, and related forms of runtime monitors, are found also in drones [14], control systems for robots [62], and in various other formalisms which are not directed particularly at deep learning [18,26,44,63,73]. In recent years, the formal methods community

has started studying override rules for systems with DNNs: for example, recent papers have proposed techniques for synthesizing override rules that affect the controller in minimal ways [3,72].

SBM and its various aspects, especially those pertaining to the formal analysis of scenario-based models, have been thoroughly studied over the last decade. These aspects include the automatic verification [31], repair [36], optimization [24,29,35,66,67] and synthesis [23] of scenario-based models. SBM has also played a key role in the Wise Computing initiative [33,34,60], which seeks to make the computer a proactive team member, capable of developing complex models hand-in-hand with human engineers.

In this work we focused on SBM as a possible formalism for expressing override rules. There exists other, related modeling schemes, which could also be used for similar purposes. For example, the publish-subscribe framework for parallel composition shares many traits with SBM [17], and could be applied in a similar way. Aspect oriented programming [53] is another formalism, which allows developers to specify and execute cross-cutting program instructions on top of a base application. Both publish-subscribe and aspect oriented programming, however, do not directly support the blocking idiom, which appears quite useful for specifying override rules. Other behavior- and scenario-based models, such as LEGO Mindstorms leJOS [2], Branicky's behavioral programming [8], and Brooks's subsumption architecture [9], all suggest constructing systems from individual behaviors. One advantage that the scenario-based approach affords compared to these formalisms is that it is language-independent, and has been implemented on top of multiple platforms. It can thus extend, in a variety of ways, the arbitration and coordination mechanisms in use by these architectures.

Another related formalism is the BIP formalism (behavior, interaction, priority) [6]. BIP uses the notion of *glue* for assembling components into cohesive systems. The goals that BIP pursues are similar to those of SBM, although BIP focuses mostly on correct-by-construction systems. SBM, in contrast, is more geared towards executing intuitively-specified scenarios, and resolving the constraints that they pose at run-time.

7 Discussion and Next Steps

As the use of DNNs is becoming widespread in multiple and varied systems, ensuring the safety of these systems is quickly turning into an urgent need—specifically by using override rules. We argue here that by using modeling schemes that model together the DNN and its override rules, progress can be made towards this important goal. We propose to use a scenario-based modeling approach for this purpose, explain how a basic scenario-based scheme can be adjusted to incorporate DNNs, and demonstrate the approach on multiple, recently-proposed DNNs.

Moving forward, we believe that applying a more structured methodology for modeling override rules raises the following key question: as the number of override rules increases and as they become more complex, could they fully capture

the DNN's logic and eventually replace it? We believe that the answer is negative, because override rules typically forbid some specified behavior, but rely on the DNN controller to prioritize among the remaining possible options. We thus believe that a more realistic approach is to combine a DNN controller together with appropriately crafted override rules, in a way that allows engineers to maintain, enhance and extend both components throughout the system's lifetime.

We consider our work to date a first step, which we intend to extend. Specifically, we plan to work on (i) leveraging the other advantages of scenario-based modeling, specifically its amenability to automated analysis and verification, in proving the overall correctness of DNN-based models; and (ii) customizing the idioms of scenario-based modeling, or similar techniques, to better suit integration with deep neural networks, and guard them in more subtle ways. In the longer run, we envision that work in this direction will eventually lead to the creation of DNN-enabled systems that are more robust, reliable, and easier to maintain and extend.

Acknowledgements. We thank Yafim (Fima) Kazak for his contributions to this project. This work was partially supported by grants from the Binational Science Foundation (2017662) and the Israel Science Foundation (683/18).

References

1. Amodei, D., Olah, C., Steinhardt, J., Christiano, P., Schulman, J., Mané, D.: Concrete Problems in AI Safety (2016). Technical report. https://arxiv.org/abs/1606.06565
2. Arkin, R.C.: Behavior-Based Robotics. MIT Press, Cambridge (1998)
3. Avni, G., Bloem, R., Chatterjee, K., Henzinger, T.A., Könighofer, B., Pranger, S.: Run-time optimization for learned controllers through quantitative games. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11561, pp. 630–649. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4_36
4. Bar-Sinai, M., Weiss, G., Shmuel, R.: BPjs: an extensible, open infrastructure for behavioral programming research. In: Proceedings 21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS), pp. 59–60 (2018)
5. Barrett, C., Tinelli, C.: Satisfiability modulo theories. In: Clarke, E., Henzinger, T., Veith, H., Bloem, R. (eds.) Handbook of Model Checking, pp. 305–343. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-10575-8_11
6. Bliudze, S., Sifakis, J.: A notion of glue expressiveness for component-based systems. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 508–522. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-85361-9_39
7. Bojarski, M., et al.: End to End Learning for Self-Driving Cars (2016). Technical report. http://arxiv.org/abs/1604.07316
8. Branicky, M.: Behavioral programming. In: Working Notes AAAI Spring Symposium on Hybrid Systems and AI (1999)
9. Brooks, R.: A robust layered control system for a mobile robot. Robot. Autom. **2**(1), 14–23 (1986)

10. Chicco, D., Sadowski, P., Baldi, P.: Deep autoencoder neural networks for gene ontology annotation predictions. In: Proceedings 5th ACM Conference on Bioinformatics, Computational Biology, and Health Informatics (BCB), pp. 533–540 (2014)
11. Cisse, M., Adi, Y., Neverova, N., Keshet, J.: Houdini: fooling deep structured prediction models. In: Proceedings of the 31st Conference on Neural Information Processing Systems (NeurIPS) (2017)
12. Collobert, R., Weston, J., Bottou, L., Karlen, M., Kavukcuoglu, K., Kuksa, P.: Natural language processing (almost) from scratch. J. Mach. Learn. Res. (JMLR) **12**, 2493–2537 (2011)
13. Damm, W., Harel, D.: LSCs: breathing life into message sequence charts. J. Formal Methods Syst. Des. (FMSD) **19**(1), 45–80 (2001)
14. Desai, A., Ghosh, S., Seshia, S., Shankar, N., Tiwari, A.: SOTER: Programming Safe Robotics System using Runtime Assurance (2018). Technical report. https://arxiv.org/abs/1808.07921
15. Devlin, J., Chang, M.W., Lee, K., Toutanova, K.: BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding (2018). Technical Report. http://arxiv.org/abs/1810.04805
16. Elkahky, A., Song, Y., He, X.: A multi-view deep learning approach for cross domain user modeling in recommendation systems. In: Proceedings of the 24th International Conference on World Wide Web (WWW), pp. 278–288 (2015)
17. Eugster, P., Felber, P., Guerraoui, R., Kermarrec, A.: The many faces of publish/-subscribe. ACM Comput. Surv. (CSUR) **35**(2), 114–131 (2003)
18. Falcone, Y., Mounier, L., Fernandez, J., Richier, J.: Runtime enforcement monitors: composition, synthesis, and enforcement abilities. J. Formal Methods Syst. Des. (FMSD) **38**(3), 223–262 (2011)
19. Gehr, T., Mirman, M., Drachsler-Cohen, D., Tsankov, E., Chaudhuri, S., Vechev, M.: AI2: safety and robustness certification of neural networks with abstract interpretation. In: Proceedings of the 39th IEEE Symposium on Security and Privacy (S&P) (2018)
20. Goodfellow, I., Bengio, Y., Courville, A.: Deep Learning. MIT Press, Cambridge (2016)
21. Gottschlich, J., et al.: The three pillars of machine programming. In: Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL), pp. 69–80 (2018)
22. Greenyer, J., et al.: ScenarioTools – a tool suite for the scenario-based modeling and analysis of reactive systems. J. Sci. Comput. Program. (J. SCP) **149**, 15–27 (2017)
23. Greenyer, J., Gritzner, D., Katz, G., Marron, A.: Scenario-based modeling and synthesis for reactive systems with dynamic system structure in ScenarioTools. In: Proceedings of the 19th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS), pp. 16–23 (2016)
24. Greenyer, J., et al.: Distributed execution of scenario-based specifications of structurally dynamic cyber-physical systems. In: Proceedings of the 3rd International Conference on System-Integrated Intelligence: New Challenges for Product and Production Engineering (SYSINT), pp. 552–559 (2016)
25. Gritzner, D., Greenyer, J.: Synthesizing executable PLC code for robots from scenario-based GR(1) specifications. In: Seidl, M., Zschaler, S. (eds.) STAF 2017. LNCS, vol. 10748, pp. 247–262. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-74730-9_23

26. Hamlen, K., Morrisett, G., Schneider, F.: Computability classes for enforcement mechanisms. ACM Trans. Program. Lang. Syst. (TOPLAS) **28**(1), 175–205 (2006)
27. Harel, D., Kantor, A., Katz, G.: Relaxing synchronization constraints in behavioral programs. In: McMillan, K., Middeldorp, A., Voronkov, A. (eds.) LPAR 2013. LNCS, vol. 8312, pp. 355–372. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-45221-5_25
28. Harel, D., Kantor, A., Katz, G., Marron, A., Mizrahi, L., Weiss, G.: On composing and proving the correctness of reactive behavior. In: Proceedings of the 13th International Conference on Embedded Software (EMSOFT), pp. 1–10 (2013)
29. Harel, D., Kantor, A., Katz, G., Marron, A., Weiss, G., Wiener, G.: Towards behavioral programming in distributed architectures. J. Sci. Comput. Program. (J. SCP) **98**, 233–267 (2015)
30. Harel, D., Katz, G.: Scaling-up behavioral programming: steps from basic principles to application architectures. In: Proceedings of the 4th SPLASH Workshop on Programming based on Actors, Agents and Decentralized Control (AGERE!), pp. 95–108 (2014)
31. Harel, D., Katz, G., Lampert, R., Marron, A., Weiss, G.: On the succinctness of idioms for concurrent programming. In: Proceedings of the 26th International Conference on Concurrency Theory (CONCUR), pp. 85–99 (2015)
32. Harel, D., Katz, G., Marelly, R., Marron, A.: An initial wise development environment for behavioral models. In: Proceedings of the 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD), pp. 600–612 (2016)
33. Harel, D., Katz, G., Marelly, R., Marron, A.: First steps towards a wise development environment for behavioral models. Int. J. Inf. Syst. Model. Des. (IJISMD) **7**(3), 1–22 (2016)
34. Harel, D., Katz, G., Marelly, R., Marron, A.: Wise computing: toward endowing system development with proactive wisdom. IEEE Comput. **51**(2), 14–26 (2018)
35. Harel, D., Katz, G., Marron, A., Sadon, A., Weiss, G.: Executing scenario-based specification with dynamic generation of rich events. In: Hammoudi, S., Pires, L.F., Selić, B. (eds.) MODELSWARD 2019. CCIS, vol. 1161, pp. 246–274. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-37873-8_11
36. Harel, D., Katz, G., Marron, A., Weiss, G.: Non-intrusive repair of reactive programs. In: Proceedings of the 17th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS), pp. 3–12 (2012)
37. Harel, D., Katz, G., Marron, A., Weiss, G.: Non-intrusive repair of safety and liveness violations in reactive programs. Trans. Comput. Collect. Intell. (TCCI) **16**, 1–33 (2014)
38. Harel, D., Katz, G., Marron, A., Weiss, G.: The effect of concurrent programming idioms on verification. In: Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development (MODELSWARD), pp. 363–369 (2015)
39. Harel, D., Marron, A., Weiss, G.: Programming coordinated behavior in Java. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 250–274. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14107-2_12
40. Harel, D., Marron, A., Weiss, G.: Behavioral programming. Commun. ACM (CACM) **55**(7), 90–100 (2012)
41. Huang, X., Kwiatkowska, M., Wang, S., Wu, M.: Safety verification of deep neural networks. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10426, pp. 3–29. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_1

42. Jacoby, Y., Barrett, C., Katz, G.: Verifying recurrent neural networks using invariant inference. In: Proceedings of the 18th International Symposium on Automated Technology for Verification and Analysis (ATVA) (2020)
43. Jay, N., Rotman, N., Brighten Godfrey, P., Schapira, M., Tamar, A.: Internet congestion control via deep reinforcement learning. In: Proceedings of the 32nd Conference on Neural Information Processing Systems (NeurIPS) (2018)
44. Ji, Y., Lafortune, S.: Enforcing opacity by publicly known edit functions. In: Proceedings of the 56th IEEE Annual Conference on Decision and Control (CDC), pp. 12–15 (2017)
45. Julian, K., Lopez, J., Brush, J., Owen, M., Kochenderfer, M.: Policy compression for aircraft collision avoidance systems. In: Proceedings of the 35th Digital Avionics Systems Conference (DASC), pp. 1–10 (2016)
46. Katz, G.: On module-based abstraction and repair of behavioral programs. In: McMillan, K., Middeldorp, A., Voronkov, A. (eds.) LPAR 2013. LNCS, vol. 8312, pp. 518–535. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-45221-5_35
47. Katz, G.: Guarded deep learning using scenario-based modeling. In: Proceedings of the 8th International Conference on Model-Driven Engineering and Software Development (MODELSWARD), pp. 126–136 (2020)
48. Katz, G., Barrett, C., Dill, D.L., Julian, K., Kochenderfer, M.J.: Reluplex: an efficient SMT solver for verifying deep neural networks. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10426, pp. 97–117. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_5
49. Katz, G., Barrett, C., Harel, D.: Theory-aided model checking of concurrent transition systems. In: Proceedings of the 15th International Conference on Formal Methods in Computer-Aided Design (FMCAD), pp. 81–88 (2015)
50. Katz, G., et al.: The marabou framework for verification and analysis of deep neural networks. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11561, pp. 443–452. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4_26
51. Katz, G., Marron, A., Sadon, A., Weiss, G.: On-the-fly construction of composite events in scenario-based modeling using constraint solvers. In: Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development (MODELSWARD), pp. 143–156 (2019)
52. Kazak, Y., Barrett, C., Katz, G., Schapira, M.: Verifying deep-RL-driven systems. In: Proceedings of the 1st ACM SIGCOMM Workshop on Network Meets AI & ML (NetAI) (2019)
53. Kiczales, G., et al.: Aspect-oriented programming. In: Akşit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997). https://doi.org/10.1007/BFb0053381
54. Kuper, L., Katz, G., Gottschlich, J., Julian, K., Barrett, C., Kochenderfer, M.: Toward Scalable Verification for Safety-Critical Deep Networks (2018). Technical report. http://arxiv.org/abs/1801.05950
55. Kurakin, A., Goodfellow, I., Bengio, S.: Adversarial Examples in the Physical World (2016). Technical report. http://arxiv.org/abs/1607.02533
56. Lipton, Z., Kale, D., Elkan, C., Wetzel, R.: Learning to diagnose with LSTM recurrent neural networks. In: Proceedings of the 4th International Conference on Learning Representations (ICLR) (2016)
57. Mao, H., Alizadeh, M., Menache, I., Kandula, S.: Resource management with deep reinforcement learning. In: Proceedings of the 15th ACM Workshop on Hot Topics in Networks (HotNets), pp. 50–56 (2016)

58. Mao, H., Alizadeh, M., Menache, I., Kandula, S.: Resource Management with Deep Reinforcement Learning: Implementation (2016). https://github.com/hongzimao/deeprm
59. Mao, H., Netravali, R., Alizadeh, M.: Neural adaptive video streaming with pensieve. In: Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM), pp. 197–210 (2017)
60. Marron, A., et al.: Six (im)possible things before breakfast: building-blocks and design-principles for wise computing. In: Proceedings of the 19th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS), pp. 94–100 (2016)
61. Nair, V., Hinton, G.: Rectified linear units improve restricted Boltzmann machines. In: Proceedings of the 27th International Conference on Machine Learning (ICML), pp. 807–814 (2010)
62. Phan, D., Yang, J., Grosu, R., Smolka, S., Stoller, S.: Collision avoidance for mobile robots with limited sensing and limited information about moving obstacles. J. Formal Methods Syst. Des. (FMSD) **51**(1), 62–68 (2017)
63. Schierman, J., et al.: Runtime Assurance Framework Development for Highly Adaptive Flight Control Systems (2015). Technical report. https://apps.dtic.mil/docs/citations/AD1010277
64. Silver, D., et al.: Mastering the game of go with deep neural networks and tree search. Nature **529**(7587), 484–489 (2016)
65. Simonyan, K., Zisserman, A.: Very Deep Convolutional Networks for Large-Scale Image Recognition (2014). Technical report. http://arxiv.org/abs/1409.1556
66. Steinberg, S., Greenyer, J., Gritzner, D., Harel, D., Katz, G., Marron, A.: Distributing scenario-based models: a replicate-and-project approach. In: Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development (MODELSWARD), pp. 182–195 (2017)
67. Steinberg, S., Greenyer, J., Gritzner, D., Harel, D., Katz, G., Marron, A.: Efficient distributed execution of multi-component scenario-based models. Commun. Comput. Inf. Sci. (CCIS) **880**, 449–483 (2018)
68. Sutton, R., Barto, A.: Introduction to Reinforcement Learning. MIT Press, Cambridge (1998)
69. Szegedy, C., et al.: Intriguing Properties of Neural Networks (2013). Technical report. http://arxiv.org/abs/1312.6199
70. Wan, L., Wang, Q., Papir, A., Lopez-Moreno, I.: Generalized End-to-End Loss for Speaker Verification (2017). Technical Report. http://arxiv.org/abs/1710.10467
71. Wang, S., Pei, K., Whitehouse, J., Yang, J., Jana, S.: Formal security analysis of neural networks using symbolic intervals. In: Proceedings of the 27th USENIX Security Symposium (2018)
72. Wu, M., Wang, J., Deshmukh, J., Wang, C.: Shield Synthesis for Real: Enforcing Safety in Cyber-Physical Systems (2019). Technical report. https://arxiv.org/abs/1908.05402
73. Wu, Y., Raman, V., Rawlings, B., Lafortune, S., Seshia, S.: Synthesis of obfuscation policies to ensure privacy and utility. J. Autom. Reason. **60**(1), 107–131 (2018)
74. Zhang, H., Shinn, M., Gupta, A., Gurfinkel, A., Le, N., Narodytska, N.: Verification of recurrent neural networks for cognitive tasks via reachability analysis. In: Proceedings of the 24th Conference on European Conference on Artificial Intelligence (ECAI) (2020)