Assumption Generation for the Verification of Learning-Enabled Autonomous Systems

Corina Păsăreanu[1,2], Ravi Mangal[2], Divya Gopinath[1], and Huafeng Yu[3]
[1]KBR, NASA Ames [2]Carnegie Mellon University [3]Boeing Research and Technology

Providing safety guarantees for autonomous systems is difficult as these systems operate in complex environments that require the use of learning-enabled components, such as deep neural networks (DNNs) for visual perception. DNNs are hard to analyze due to their size (they can have thousands or millions of parameters), lack of formal specifications (DNNs are typically learnt from labeled data, in the absence of any formal or informal requirements), and sensitivity to small changes in the environment. We present an assume-guarantee style compositional approach for the formal verification of system-level safety properties of such autonomous systems. Our insight is that we can analyze the system *in the absence* of the DNN perception components by automatically synthesizing *assumptions* on the DNN behaviour that *guarantee* the satisfaction of the required safety properties. The synthesized assumptions are the *weakest* in the sense that they characterize the output sequences of all the possible DNNs that, plugged into the autonomous system, guarantee the required safety properties. The assumptions can be leveraged as run-time monitors over a deployed DNN to guarantee the safety of the overall system; they can also be mined to extract local specifications for use during training and testing of DNNs. We illustrate our approach on a case study taken from the autonomous airplanes domain that uses a complex DNN for perception.

***Index Terms*—Autonomous systems, Closed-loop safety, Assumptions.**

I. Introduction

Autonomy is increasingly prevalent in many applications, such as recommendation systems, social robots and self-driving vehicles, that require strong safety guarantees. However, this is difficult to achieve, since autonomous systems are meant to operate in uncertain environments that require using machine-learnt components. For instance, deep neural networks (DNNs) can be used in autonomous vehicles to perform complex tasks such as perception from high-dimensional images. DNNs are massive (with thousands, millions or even billions of parameters) and are inherently opaque, as they are trained based on data, typically in the absence of any specifications, thus precluding formal reasoning over their behaviour. Current system-level assurance techniques that are based on formal methods, either do not scale to systems that contain complex DNNs [1, 2], provide no guarantees [3], or provide only probabilistic guarantees [4, 5] for correct operation of the autonomous system. Falsification techniques [6] can be used to find counterexamples to safety properties but they cannot guarantee that the properties hold.

Moreover, it is known that, even for well-trained, highly-accurate DNNs, their performance degrades in the presence of distribution shifts or adversarial and natural perturbations from the environment (e.g., small changes to correctly classified inputs that cause DNNs to mis-classify them) [7]. These phenomena present safety concerns but it is currently unknown how to provide strong assurance guarantees about such behaviours. Despite significant effort in the area, current formal verification and certification techniques for DNNs [8, 9] only scale to modest-sized networks and provide only partial guarantees about input-output DNN behaviour, i.e. they do not cover the whole input space. Furthermore, it is unknown how to relate these (partial) DNN guarantees to strong guarantees about the safety of the overall autonomous system.

We propose a compositional verification approach for learning-enabled autonomous systems to achieve strong assurance guarantees. The inputs to the approach are: the design models of an autonomous system, which contains both conventional components (e.g., controller and plant modeled as labeled transition systems) and the learning-enabled components (e.g., DNN used for perception), and a safety property specifying the desired behaviour of the system.

While the conventional components can be modeled and analyzed using well-established techniques (e.g., using model checking for labeled transition systems, as we do in this paper), the challenge is to reason about the perception components. This includes the complex DNN together with the sensors (e.g., cameras) that generate the high-dimensional DNN inputs (e.g., images), which are subject to random perturbations from the environment (e.g., change in light conditions), all of them difficult, if not impossible, to model precisely. To address this challenge, we take an abductive reasoning approach, where we analyze the system *in the absence* of the DNN and the sensors, deriving conditions on DNN behaviour that guarantee the safety of the overall system. We build on our previous work on automated assume-guarantee compositional verification [10, 11], to automatically generate *assumptions* in the form of labeled transition systems, encoding sequences of DNN predictions that guarantee system-level safety. The assumptions are the weakest in the sense that they characterize the output sequences of all the possible DNNs that, plugged into the autonomous system, satisfy the property. We further propose to mine the assumptions to extract local properties on DNN behavior, which in turn can be used for the separate testing and training of the DNNs.

We envision the approach to be applied at different development phases for the autonomous system. At design time, the approach can be used to uncover problems in the autonomous system *before* deployment. The automatically

generated assumptions and the extracted local properties can be seen as *safety requirements* for the development of neural networks. At run time, the assumptions can be deployed as safety monitors over the DNN outputs to *guarantee the safety behaviour* of the overall system.

Developers can use the assumptions to test and debug their DNNs, e.g., on sequences of images obtained from simulations. The assumptions can also be used for training of the DNNs on inputs that do not require manual labelling. The extracted local specifications can also be used to facilitate training and testing of DNNs, even for data that does not come in sequence.

We summarize our contributions as follows: (1) Analysis with strong safety guarantees for autonomous systems with learning-enabled perception components. The outcome of the analysis is in the form of *assumptions* and *local specifications* over DNN behavior, which can be used for training and testing the DNN and also for run-time monitoring to provide the safety guarantees. (2) Demonstration of the approach on a case study inspired by a realistic scenario of an autonomous taxiing system for airplanes, that uses a complex neural network for perception. (3) Experimental results showing that the extracted assumptions are small and understandable, even if the perception DNN has large output spaces, making them amenable for training and testing of DNNs and also for run-time monitoring. (4) Probabilistic analysis, using empirical probabilities derived from profiling the perception DNN, to measure the probability that the extracted assumptions are violated when deployed as run-time safety monitors. Such an analysis enables developers to estimate how restrictive the safety monitors are in practice.

II. Preliminaries

*a) **Labeled Transition Systems**:* We use finite labelled transition systems (LTSs) to model the behaviour of an autonomous system as communicating processes. A *labeled transition system* (LTS) is a tuple $M = (Q, \Sigma, \delta, q_0)$, where

- Q is a finite set of *states*;
- Σ, the *alphabet* of M, is a set of observable actions;
- $\delta \subseteq Q \times (\Sigma \cup \{\tau\}) \times Q$ is a *transition relation*;
- $q_0 \in Q$ is the initial state.

Here τ denotes a local, unobservable action. We use $\alpha(M)$ to denote the alphabet of M (i.e. $\alpha(M) = \Sigma$). A trace $\sigma \in \Sigma^*$ of an LTS M is a sequence of observable actions that M can perform starting in the initial state. The *language* of M, denoted $L(M)$, is the set of traces of M. Note that our definition allows non-deterministic transitions.

Given two LTSs M_1 and M_2, their *parallel composition* $M_1 \,||\, M_2$ synchronizes shared actions and interleaves the remaining actions. We provide the definition of $||$ (which is commutative and associative) in a process-algebra style. Let $M = (Q, \Sigma, \delta, q_0)$ and $M' = (Q', \Sigma', \delta', q'_0)$ be two LTSs. We say that M *transits* to M' with action a, written as $M \xrightarrow{a} M'$, iff $(q_0, a, q'_0) \in \delta$, $\Sigma = \Sigma'$, and $\delta = \delta'$. Let $M_1 = (Q_1, \Sigma_1, \delta_1, q_{1,0})$ and $M_2 = (Q_2, \Sigma_2, \delta_2, q_{2,0})$. $M_1 \,||\, M_2$ is an LTS $M = (Q, \Sigma, \delta, q_0)$ such that $Q = Q_1 \times Q_2$, $q_0 = (q_{1,0}, q_{2,0})$, $\Sigma = \Sigma_1 \cup \Sigma_2$ and δ is defined as follows:

$$\frac{M_1 \xrightarrow{a} M'_1, a \notin \Sigma_2}{M_1 \,||\, M_2 \xrightarrow{a} M'_1 \,||\, M_2} \qquad \frac{M_1 \xrightarrow{a} M'_1, M_2 \xrightarrow{a} M'_2, a \neq \tau}{M_1 \,||\, M_2 \xrightarrow{a} M'_1 \,||\, M'_2}$$

We also use LTSs to represent safety properties P. P can be synthesized, for example, from a specification in a temporal logic formalism such as (fluent) LTL [12]. The language of P describes the set of allowable behaviours for M; $M \models P$ iff $L(M{\downarrow_{\alpha(P)}}) \subseteq L(P)$ where $\alpha(P)$ is the alphabet of of P. The $\downarrow_\Sigma$ operation hides (i.e., makes unobservable by replacing with τ) all the observable actions of an LTS that are not in Σ. The verification of property P is performed by first building an *error* LTS, P_{err}, which is the complement of P trapping possible violations with an extra error state err, and checking reachability of err in $M \,||\, P_{err}$.

*b) **Weakest Assumption**:* A system M (modeled as an LTS) can be viewed as an *open* system interacting with its environment through an interface $\Sigma_I \subseteq \alpha(M)$. For a property P, the weakest assumption characterizes *all* the *environments* in which M can be guaranteed to satisfy the property. We formalize it here, generalizing from [11].

Definition 1 (Weakest assumption). *Let M be an LTS, P a safety property ($\alpha(P) \subseteq \alpha(M)$) and $\Sigma'_I \subseteq \Sigma_I$ be an alphabet of interface actions. The weakest assumption $A_w^{\Sigma'_I}$ for M with respect to P and Σ'_I is a (deterministic) LTS such that $\alpha(A_w^{\Sigma'_I}) = \Sigma'_I$ and for any component M', $M{\downarrow_{\Sigma'_I}} \,||\, M' \models P$ iff $M' \models A_w^{\Sigma'_I}$.*

While it is natural to think of the weakest assumption in terms of the complete interface alphabet Σ_I between M and its environment, we shall see that it is also useful to compute assumptions with respect to smaller interface alphabets $\Sigma'_I \subseteq \Sigma_I$, as only the subset Σ'_I may be observable in practice. Consequently, only the projection $M{\downarrow_{\Sigma'_I}}$, which 'forces' M to communicate with its environment through the actions that are present in Σ'_I, is relevant when reasoning about the weakest assumption.

*c) **Automatic Generation of Assumptions**:* From prior work [11], we know that the weakest assumption exists for components and safety properties modeled as LTSs. The same work also provides an algorithm to build such assumptions automatically, summarized in Algorithm II.1.

The function `BuildAssumption` has as parameters an LTS model M, a property P, and an interface alphabet Σ'_I. The first step builds $M \,||\, P_{err}$ (P_{err} is the complement of P), which contains *all* the traces that violate P, and applies projection with Σ'_I to obtain the LTS M'. This LTS is further processed with a *determinization step* which performs τ elimination and subset construction (for converting the non-deterministic LTS into a deterministic one). Unlike regular automata algorithms, `Determinize` handles the err state in a special way. During subset construction, a state of the determinisic LTS represents a *set* of states in the nondeterministic LTS M'; if any of these states is err, then the whole set becomes err in the deterministic LTS. Specifically, if performing a sequence of actions from Σ does not *guarantee* that M is safe, it

Algorithm II.1: Computing Weakest Assumption

Inputs: LTS model M, property P, and interface alphabet Σ'_I

Output: Assumption $A_w^{\Sigma'_I}$ for M with respect to P, Σ'_I

```
BuildAssumption(M, P, Σ'_I):
    M' := (M || P_err)↓Σ'_I
    A_err^Σ'_I := Determinize(M')
    Â_err^Σ'_I := CompleteWithSink(A_err^Σ'_I)
    A_w^Σ'_I := RemoveError(Â_err^Σ'_I)
    return A_w^Σ'_I
```

is considered as an error trace. Subsequently, the resulting deterministic LTS $A_{err}^{\Sigma'_I}$ is *completed* such that every state has an outgoing transition for every action. This is done by adding a special $sink$ state and transitions leading to it. The missing transitions in $A_{err}^{\Sigma'_I}$ represent behaviors that are never exercised by M; with this completion, they are made into sink behaviors and no restriction is placed on them. The assumption $A_w^{\Sigma'_I}$ is obtained from the complete LTS by removing the err state and all the transitions to it.

III. Compositional Verification of Learning-enabled Autonomous Systems

We present a compositional approach for verifying the safety of autonomous systems with learning-enabled components. We model our system as a parallel composition of LTSs; however our approach is more general, and can be adapted to reasoning about more complex, possibly unbounded, representations, such as LTSs with countably infinite number of states and hybrid systems, by leveraging previous work on assuming-guarantee reasoning for such systems [13, 14]. We focus on cyber-physical systems that use DNNs for vision perception (or more generally, perception from high-dimensional data). These DNNs are particularly difficult to reason about, due to large sizes, opaque nature, and sensitivity to input perturbations or distribution shifts.

Let us consider an autonomous system consisting of four components; systems with more components can be treated similarly. The system contains a $Perception$ component (i.e., a DNN) which processes images ($img \in Img$) and produces estimates ($s_{est} \in Est$) of the system state, a $Controller$ that sends commands[1] ($c \in Cmd$) to the physical system being controlled in order to maneuver it based on these state estimates, the $Dynamics$ modeling the evolution of the actual physical system states ($s \in Act$) in response to control signals, and the $Sensor$, e.g., a high-definition camera, that captures images representing the current state of the system and its surrounding environment ($e \in Env$). There may be other sensors (radar, LIDAR, GPS) that we abstract away for simplicity.

Suppose that each of these components can be modeled as an LTS. The alphabet of observable actions for each component is as follows: $\alpha(Perception) = Img \cup Est$, $\alpha(Controller) = Est \cup Cmd$, $\alpha(Dynamics) = Act \cup Cmd$, and $\alpha(Sensor) = Act \cup Env \cup Img$. We can write the overall system as $System = Sensor \,||\, Perception \,||\, Controller \,||\, Dynamics$.

Although simple, the type of system we consider resembles (semi-)autonomous mechanisms that are already deployed in practice, such as adaptive cruise controllers and lane-keeping assist systems, which similarly use a DNN for visual perception to provide guidance to the software controlling electrical and mechanical subsystems of modern vehicles. An example of such a system designed for autonomous taxiing of airplanes on taxiways is illustrated in Figure 1. Section IV includes a detailed explanation of this system.

We aim to check that the overall system satisfies a *safety property* P. For the example described in the next section, one such safety property is that the airplane does not go off the taxiway, which can be expressed in terms of constraints on the allowed actual system states. In order to check this property, one could run many simulations, using, e.g., XPlane [15]. However, simulation alone may not be enough to achieve the high degree of confidence in the correctness of the system necessary for deployment in a safety-critical setting (e.g., an airplane in our case). We therefore aim to formally verify the property, i.e., we aim to check that $System \models P$ holds.

Formally verifying $System$ presents serious scalability challenges, even ignoring the learning-enabled aspect, since the conventional components ($Controller$ and $Dynamics$ in our case) can be quite complex; nevertheless they can be tackled with previous techniques, possibly involving abstraction to reduce their state spaces [13]. However, the DNN component makes the scalability problem extremely severe. Further, the perturbations from the external world can not be modeled precisely.

*a) **Assume-guarantee Reasoning**:* To address the above challenges, we propose to use compositional techniques that decompose the verification of $System$ into the separate verification of its components. In particular, we decompose $System$ into two subsystems—the conventional components, i.e., $Controller$ and $Dynamics$, which can be modeled and analyzed using established model-checking techniques, on one side, and the perception components, i.e., the DNN together with the $Sensor$, which are challenging to analyze precisely, on the other side; see illustration in Figure **??**. Thus, we decompose $System$ into $M_1 = Controller \,||\, Dynamics$ and $M_2 = Perception \,||\, Sensor$. We then focus on the analysis of M_1. This analysis can be further decomposed using similar compositional techniques, to increase scalabiliy.

Formally checking that system-level property P holds on M_1 in isolation does not make too much sense, as M_1 is meant to work together with M_2 and will not satisfy P by itself (except in very particular cases). Assume-guarantee reasoning addresses this problem by checking properties using *assumptions* about a component's context (i.e., the rest of the system). The simplest rule for such reasoning checks if a system composed of M_1 and M_2 satisfies a property P by checking that M_1 satisfies P under assumption A and discharging A on 'context' M_2. This rule can be represented as follows:

[1]We use the term "commands" instead of the standard term "actions" since we already use actions to refer to the transition labels of LTSs.

$$\frac{\langle A, M_1, P\rangle \quad \langle true, M_2, A\rangle}{\langle true, M_1 \| M_2, P\rangle}$$

Here $\langle A, M, P\rangle$ denotes an assume-guarantee triple which is true if whenever M is part of a system that satisfies A, the system also guarantees P, i.e., $\forall M', M \mid\mid M' \models A \Rightarrow M \mid\mid M' \models P$. For LTSs, this is equivalent to $A \mid\mid M \models P$.

We then seek to automatically build an assumption A such that $\langle A, M_1, P\rangle$ holds; one such assumption is the weakest assumption described in Section II-0c; i.e., by definition $\langle A_w^{\Sigma_I'}, M_1, P\rangle$ is true. If we can also show that $\langle true, M_2, A\rangle$, then, according to the assume-guarantee rule, it follows that the autonomous system $System = M_1 \| M_2$ satisfies the property.

Formally checking $\langle true, M_2, A\rangle$ (the second premise of our rule) is infeasible due to the complexity of the DNN and difficulty in mathematically modeling the $Sensor$ component as well as the random environment conditions (as explained before). Instead, we show how the assumption can be leveraged for monitoring (at run-time) the outputs of the DNN, to *guarantee* that the overall system satisfies the required property. Furthermore, we show how the automatically generated assumption can be leveraged for extracting local DNN specifications, which in turn can be used for training and testing the DNN.

*b) **Assumption Generation**:* We build the weakest assumption for M_1 with respect to property P using Algorithm II.1, according to an interface alphabet for M_1.

Interface Alphabet. The *interface* between M_1 and M_2 consists of the updates to the actual system states, henceforth called *actuals* (performed by the $Dynamics$ component) and to the estimated system states, henceforth called *estimates* (performed by the $Perception$ component); let us denote it as $\Sigma_I = Act \cup Est$.

By construction, $A_w^{\Sigma_I}$ captures precisely the traces over Σ_I^* that ensure that M_1 does not violate the prescribed safety property, i.e. $A_w^{\Sigma_I} \mid\mid M_1 \models P$ and therefore $\langle A_w^{\Sigma_I}, M_1, P\rangle$. Furthermore, $A_w^{\Sigma_I}$ is the weakest assumption, ensuring that $A_w^{\Sigma_I}$ does not restrict M unnecessarily.

As we shall see, it is also useful to consider a smaller alphabet $\Sigma_I' = Est$ when building the assumptions (as this leads to useful run-time monitors). We note the following property which relates weakest assumptions for different interface alphabets.

Theorem 2. *Given LTS M, property P, and interface alphabet $\Sigma_I \subseteq \alpha(M)$, $\forall \Sigma_I', \Sigma_I''$. $\Sigma_I' \subseteq \Sigma_I'' \subseteq \Sigma_I \Rightarrow L(A_w^{\Sigma_I'}) \subseteq L(A_w^{\Sigma_I''}) \subseteq L(A_w^{\Sigma_I})$.*

Proof. (Sketch) The intuition is that projection with a smaller alphabet results in more behaviours in M leading also to more error behaviours; consequently, the corresponding assumptions are more restrictive (contain less behaviours). Conversely, by adding actions to the interface alphabet, the corresponding assumption becomes weaker, more permissive. □

Thus, $A_w^{\Sigma_I'}$ is only weakest with respect to a particular interface alphabet Σ_I'. Moreover, $L(A_w^{\Sigma_I'})$ can be empty if no assumption with alphabet Σ_I' can ensure that the system M satisfies property P. In section V-0b we will describe an experiment that aims to quantify how permissive the assumption is in the context of a system that uses a realistic DNN for perception.

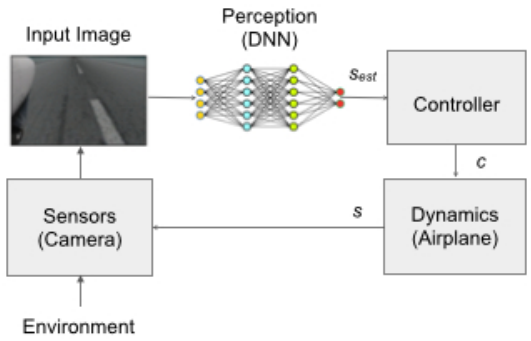

Fig. 1: Autonomous system using a DNN for perception

*c) **Uses of Weakest Assumptions $A_w^{\Sigma_I'}$ and $A_w^{\Sigma_I}$:***

$A_w^{\Sigma_I'}$ for Run-time Monitoring. The generated assumptions can be used as monitors deployed at run-time to ensure that the autonomous system guarantees the desired safety properties. One difficulty is that Σ_I refers to labels in Act which represent the actual values of the system states. However the autonomous system may not have access to the actual system state—the very reason it uses a DNN is to get the estimated values of the system state (which we denoted by Est).

While in some cases, it may be possible to get the actual values through alternative means, e.g., through other sensors, we can also set the alphabet of the assumption to be only in terms of the estimates received from the DNN, i.e., $\Sigma_I' = Est$, and build a run-time monitor solely based on $A_w^{\Sigma_I'}$.

Since $A_w^{\Sigma_I'}$ is modeled only in terms of the Est alphabet, it follows that it can be deployed as a run-time monitor on the outputs of a DNN that is used by the autonomous system.

$A_w^{\Sigma_I'}$ for Testing and Training DNNs. The extracted assumptions over alphabet Est can also be used for testing a candidate DNN to ensure that it follows the behaviour prescribed by the assumption. For many autonomous systems (see e.g., the airplane taxiing application in section IV), the perception DNN is trained and tested based on images obtained from simulations which naturally come in a sequence, and therefore, can be easily checked against the assumption by evaluating if the sequence of DNN predictions represents a trace in $L(A_w^{\Sigma_I'})$. The assumption can also be used during the training of the DNN as a specification of desired output for unlabeled real data, thus reducing the burden of manually labeling the input images. We leave these directions for future work.

$A_w^{\Sigma_I}$ for Synthesizing Local Specifications. We also propose to analyze the weakest assumption (expressed as an LTS) generated over the full interface alphabet $\Sigma_I = Act \cup Est$ to synthesize local, non-temporal specifications for the DNN. These specifications can be used as formal *documentation* of the expected DNN behavior; furthermore, they can be leveraged to train and test the DNN. Unlike the temporal LTS assumptions, evaluating the DNN with respect to local specifications does not require sequential data, making them more natural to use when evaluating DNNs.

Algorithm III.1: Synthesize Local Specifications

Inputs: Assumption with err, $A^{\Sigma_I}_{err} = (Q, \Sigma_I, \delta, q_0)$
Output: Local specifications Φ

```
1  SynthesizeSpec(A^{Σ_I}_{err}):
2      Φ := {}
3      foreach q ∈ Q do
4          if ∃a.(q, a, err) ∈ δ then
5              E := {a | (q, a, err) ∈ δ}
6              E' := Est − E
7              foreach (q', a', q) ∈ δ do
8                  φ := (s = a') ⇒ ⋁_{a∈E'}(s_est = a)
9                  Φ := Φ ∪ φ
10     return Φ
```

Algorithm III.1 describes the procedure for synthesizing such local specifications from the assumption. The input to the algorithm is the assumption with an err state (i.e., the output of line 3 in Algorithm II.1). We first note that the assumption $A^{\Sigma_I}_{err}$ does not restrict the actuals; i.e., only transitions corresponding to estimates (i.e., labeled with elements from Est) can lead to the err state. In other words, transitions in $A^{\Sigma_I}_{err}$ corresponding to actuals (i.e., labeled with elements from Act) are always followed by an Est-labeled transition. Intuitively, this happens because for a safety property P that restricts the allowed actuals, the subsystem $M_1 = Controller \,||\, Dynamics$ can reach the err state only after the $Controller$ reads the estimates and issues a command. If the command does not cause the actuals to evolve to err, then a transition corresponding to an actual is necessarily followed by a transition corresponding to an estimate. Algorithm III.1 exploits this structure in $A^{\Sigma_I}_{err}$ to synthesize local specifications. For each state q in $A^{\Sigma_I}_{err}$ (lines 3-9) that can directly transition to the err state (lines 4-9), the algorithm first collects all the actions a that lead to err (line 5). As described earlier, all these actions necessarily belong to the alphabet Est. Next, for each incoming transition to q (lines 7-9), we construct a local specification ϕ (line 8). Each incoming transition to q necessarily corresponds to an action $a' \in Act$ as described earlier. The synthesized local specification expresses that for an actual system state s with value a', the corresponding estimated system state (s_{est}) should have a value in E' to avoid err. Thus, these local specifications tolerate some error in the estimation, as they do not strictly prescribe that the estimated values (i.e., the outputs of the DNN) should be the same as the actual values (i.e., the ground truth for the DNN).

We can argue that if $M_2 = Perception \,||\, Sensor$ satisfies these local specifications then it also satisfies the assumption (proof by contradiction). Intuitively, this is true because the local specifications place stronger requirements on M_2 compared with the assumption $A^{\Sigma_I}_w$.

Theorem 3. *Given assumption $A^{\Sigma_I}_{err}$ and $M_2 = Perception \,||\, Sensor$, if M_2 satisfies local specifications $\Phi =$ `SynthesizeSpec`$(A^{\Sigma_I}_{err})$, then $\langle true, M_2, A^{\Sigma_I}_w \rangle$.*

Proof. (Sketch) Assume that $\langle true, M_2, A^{\Sigma_I}_w \rangle$ does not hold, i.e., there is a counterexample trace σ of M_2 that violates the assumption. Since $A^{\Sigma_I}_w$ only restricts the state estimates, it must be the case that the last action in this trace is an estimate $s_{est} \in Est$. Let q_i be the state in the assumption that is reached by simulating σ on $A^{\Sigma_I}_w$ up to the last, violating state estimate s_{est}. Since M_2 satisfies the local specification for q_i it means there is no such s_{est}, reaching a contradiction. □

Furthermore, if $\langle true, M_2, A^{\Sigma_I}_w \rangle$ holds, then, according to the assume-guarantee reasoning rule, it follows that the $System = M_1 \,||\, M_2$ satisfies the required properties.

While it may be infeasible to *formally prove* such properties for M_2, we envision that these local specifications can be used instead for testing and even training the DNN. Given an image img labeled with the underlying actual a' (i.e., the $Sensor$ produces img when the actual state is a'), we can test the DNN against the local specification $s = a' \Rightarrow \bigvee_{a \in E'}(s_{est} = a)$, by checking if the DNN prediction on img satisfies the consequent of the specification. Compared with the standard DNN testing objective that checks if the state estimated by the DNN *matches* the underlying actual system state, our local specifications yield a relaxed testing objective. Similarly, these specifications can also be used during training to relax the training objective. Instead of requiring the DNN to predict the actual system state from the image, under the relaxed objective, any prediction that satisfies the corresponding local specification is acceptable. Such a relaxed objective could potentially lead to a better DNN due to the increased flexibility afforded to the training process, but we leave the exploration of this direction for future work.

IV. The TaxiNet System

In this section, we present a case study applying our compositional approach to a realistic autonomous system. In particular, we analyse the design models for an experimental autonomous system for center-line tracking of airplanes on airport taxiways. The system uses a DNN called TaxiNet for perception. TaxiNet is a regression model with 24 layers including five convolution layers, and three dense layers (with 100/50/10 ELU neurons) before the output layer. TaxiNet is designed to take a picture of the taxiway as input and return the plane's position with respect to the center-line on the taxiway. It returns two outputs; cross track error (`cte`), which is the distance in meters of the plane from the center-line and heading error (`he`), which is the angle in degrees of the plane with respect to the center-line. These outputs are fed to a controller which in turn manoeuvres the plane such that it remains close to the center of the taxiway. This forms a closed-loop system where the perception network continuously receives images as the plane moves on the taxiway.

TaxiNet has been trained and tested using the X-Plane simulator [15]. Though center-lines on the pavement of airport taxiways and taxiways have standardized shapes and colors, their visibility maybe poor for a number of reasons, including skid marks on the taxiways, poor lighting conditions, and bad weather. The architecture of the system is the same as in Figure 1. For this application, state s captures the position of the airplane on the surface in terms of `cte` and `he` values.

```
Controller = S0,
S0=(turn → est[cte:CTERange][he:HERange] → S1[cte][he]),
S1[cte:CTERange][he:HERange] =
 if (cte==0 && he==0) then (cmd[0] → S0)
 else if (cte==1 && he==1) then (cmd[2] → S0)
 else if (cte==1 && he==2) then (cmd[1] → S0)
 else if (cte<1 && (he==0 || he==1)) then (cmd[2] → S0)
 else if (cte<1 && he==2) then (cmd[0] → S0)
 else if (cte>1 && (he==0 || he==2)) then (cmd[1] → S0)
 else if (cte>1 && he==1) then (cmd[0] → S0).
```

```
Dynamics = (start[cte:InitCTERange][he:InitHERange] → S0[
     cte][he]),
S0[cte:CTERange][he:HERange] = (turn → cmd[c:CmdRange] → S1
     [cte][he][c]),
S1[cte:CTERange][he:HERange][c:CmdRange] =
 if ((he==1 && c==1)||(he==2 && c==2)) then (err → ERROR)
 else if ((cte==0 && he==0 && c==1)||(cte==2 && he==0 && c
     ==2)) then (err → ERROR)
 else if ((cte==0 && he==1 && c==0)||(cte==2 && he==2 && c
     ==0)) then (err → ERROR)
 else if (he==0 && c==0) then (act[cte][0] → S0[cte][0])
 else if ((he==0 && c==1) || (he==1 && c==0)) then (act[cte
     -1][1] → S0[cte-1][1]) //move left one position
 else if ((he==0 && c==2) || (he==2 && c==0)) then (act[cte
     +1][2] → S0[cte+1][2]) //move right one position
 else if ((he==1 && c==2) || (he==2 && c==1)) then (act[cte
     ][0] → S0[cte][0]).
```

Fig. 2: TaxiNet $Controller$ and $Dynamics$ LTSs in the process-algebra style FSP language for the LTSA tool.

*a) **Safety Properties**:* We aim to check that the system satisfies two safety properties. The properties specify conditions for safe operation in terms of allowed cte and he values for the airplane by using taxiway dimensions. The first property states that the airplane shall never leave the taxiway (i.e., $|\texttt{cte}| \leq 8$ meters). The second property states that the airplane shall never turn more than a prescribed degree (i.e., $|\texttt{he}| \leq 35$ degrees), as it would be difficult to manoeuvre the airplane from that position. Note that, the DNN output values are normalized to be in the safe range; however, this does not preclude the overall system from reaching an error state.

*b) **Component Modeling**:* We build a discrete-state model of $M_1 = Controller \mid\mid Dynamics$ as an LTS. We assume a discrete-event controller and a discrete model of the aircraft dynamics. The $Controller$ and the $Dynamics$ operate over discretized actual and estimated values of the system state. We use a fixed discretization for he and experiment with discretizations at different granularities for cte, as defined by a parameter MaxCTE. For instance, when $\texttt{MaxCTE} = 2$, the discretization divides the cte and he as follows.

$$\underline{\texttt{cte}} = \begin{cases} 0 \text{ if } \texttt{cte} \in [8, -2.7) \\ 1 \text{ if } \texttt{cte} \in [-2.7, 2.7] \\ 2 \text{ if } \texttt{cte} \in (2.7, 8] \end{cases} \qquad \underline{\texttt{he}} = \begin{cases} 1 \text{ if } \texttt{he} \in [-35, -11.67) \\ 0 \text{ if } \texttt{he} \in [-11.67, 11.66] \\ 2 \text{ if } \texttt{he} \in (11.66, 35.0] \end{cases}$$

For simplicity, we use cte and he to denote both the discrete and continuous versions in other parts of the paper (with meaning clear from context).

Figure 2 presents the LTSs for the $Controller$ and $Dynamics$ components. We use act[cte][he] to denote actual states s in the Act alphabet and est[cte][he] to denote the estimated states s_{est} in Est. While we could express the safety properties as property LTSs, for simplicity, we encode them here as the ERROR states in the LTS of the $Dynamics$ component, where an error for either cte or he indicates that the airplane is off the taxiway or turned more than the prescribed angle, respectively. The ERROR states, reached by transitions labeled err, correspond to the property err state described in Section II. Off-the-shelf tools for analyzing LTSs, such as LTSA [16], then check reachability of these error states automatically.

The $Controller$ reads the estimates via est-labeled transitions. The $Controller$ can take three possible actions to steer the airplane—GoStraight, TurnLeft, and TurnRight (denoted by cmd[0], cmd[1], and cmd[2] respectively). The $Dynamics$ updates the system state, via act-labeled transitions. Action turn is meant to synchronize the $Controller$ and the $Dynamics$, to ensure that the estimates happen after each system update.

We analyze $M_1 = Controller \mid\mid Dynamics$ as an *open* system; in M_1 the estimates can take *any* values (see transition labeled est[cte : CTERange][he : HERange] in the $Controller$), irrespective of the values of the actuals. Thus, we implicitly take a *pessimistic view* of the $Perception$ DNN and assume the worst-case—the estimates can be arbitrarily wrong—for its behavior. It may be that a well-trained DNN with high test accuracy may perform much better in practice than this worst-case scenario. However, it is well known that even highly trained, high performant DNNs are vulnerable to adversarial attacks, natural and other perturbations as well as to distribution shifts which may significantly degrade their performance. We seek to derive strong guarantees for the safety of the overall system even in such adversarial conditions, hence our *pessimistic* approach.

Note also that when using an optimistic $Perception$ component, LTSA reports no errors, meaning that the system is safe assuming no errors in the perception; see Appendix **??**.

*c) **Assumption Generation**:* We build an assumption, using the algorithm described in Section II-0c, that *restricts* M_1 in such a way that it satisfies the safety properties. At the same time, the assumption does not restrict M_1 unnecessarily, i.e., it allows M_1 to operate normally, as long as it can be prevented (via parallel composition with the assumption) from entering the error states.

For the assumption alphabet, we consider Σ_I=act[CTERange][HERange] $\cup$ est[CTERange][HERange] which consists of actual and estimated values exchanged between M_1 and M_2. As mentioned in Section III-0c, while the resulting assumption $A_w^{\Sigma_I}$ can be used for synthesizing local specifications, using it as run-time monitor can be difficult since the actual values of the system state may not be available at run-time with the system accessing the external world only through the values that are estimated by the DNN. We therefore define a second alphabet, Σ'_I=est[CTERange][HERange], which consists of only the estimated values, and build a second assumption $A_w^{\Sigma'_I}$. We describe these two assumptions in more detail below.

Assumption $A_w^{\Sigma'_I}$. Figure 3 shows the assumption that was generated for the alphabet Σ'_I consisting of only the estimated values (est[CTERange][HERange]).

In the figure, each circle represents a state. The initial state (0) is shown in red. Let us look at some of the transitions in detail. The initial state has a transition leading back to

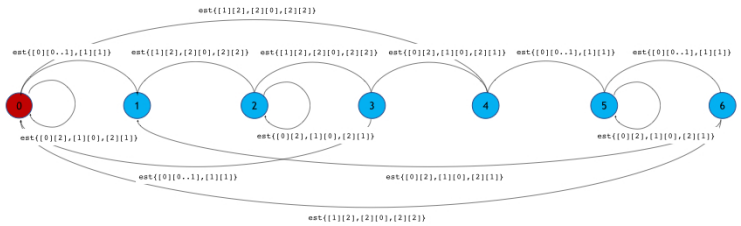

Fig. 3: Assumption A_w^{Σ} for TaxiNet when $\Sigma = Est$ and $\texttt{MaxCTE} = 2$.

```
Assumption_TaxiNet_Err = Q0,
    Q0  = (est{[0][0..1], [1][1]} → Q1
          |est{[0][2], [1][0], [2][1]} → Q8
          |est{[1][2], [2].{[0], [2]}} → Q9),
    Q1  = (act[2][2] → Q3),
    Q3  = (est{[0][0..2], [1][0..1], [2][1]} → ERROR
          |est{[1][2], [2].{[0], [2]}} → Q4),
    Q4  = (act[2][0] → Q5),
    Q5  = (est{[0][0..1], [1][1]} → ERROR
          |est{[0][2], [1][0], [2][1]} → Q4
          |est{[1][2], [2].{[0], [2]}} → Q6),
    Q6  = (act[1][1] → Q7),
    Q7  = (est{[1][2], [2].{[0], [2]}} → ERROR
          |est{[0][0..1], [1][1]} → Q8
          |est{[0][2], [1][0], [2][1]} → Q9),
    Q8  = (act[1][0] → Q0),
    Q9  = (act[0][1] → Q10),
    Q10 = (est{[0][2], [1].{[0], [2]}, [2][0..2]} → ERROR
          |est{[0][0..1], [1][1]} → Q11),
    Q11 = (act[0][0] → Q12),
    Q12 = (est{[1][2], [2].{[0], [2]}} → ERROR
          |est{[0][2], [1][0], [2][1]} → Q11
          |est{[0][0..1], [1][1]} → Q13),
    Q13 = (act[1][2] → Q14),
    Q14 = (est{[0][0..1], [1][1]} → ERROR
          |est{[0][2], [1][0], [2][1]} → Q1
          |est{[1][2], [2].{[0], [2]}} → Q8).
```

Fig. 4: Assumption $A_{err}^{\Sigma_I}$ for TaxiNet when $\Sigma_I = Est \cup Act$ and $\texttt{MaxCTE} = 2$. We show it in textual form for readability.

itself with labels $\texttt{est}[0][2], \texttt{est}[1][0], \texttt{est}[2][1]$. This indicates that if the DNN keeps estimating either $[0][2]$ or $[1][0]$ or $[2][1]$ for cte and he, then the system continues to remain safe, regardless of the actuals. Intuitively, this is true because the system starts in initial actual state $[1][0]$ and all three estimates $([0][2], [1][0], [2][1])$ lead to the same action issued by the controller, which is GoStraight, ensuring that the system keeps following the straight line, never going to error.

The assumption captures precisely the allowed DNN output sequences that ensure that the overall system does not violate the prescribed safety properties. Thus it can be seen as a *temporal specification* of the DNN behaviour which was derived automatically from the behaviour of M_1 with respect to the desired safety properties.

Assumption $A_w^{\Sigma_I}$**.** The assumption $A_{err}^{\Sigma_I}$ generated for the purpose of synthesizing local specifications, using the alphabet Σ_I with both actual and estimated values ($\texttt{act}[\texttt{CTERange}][\texttt{HERange}], \texttt{est}[\texttt{CTERange}][\texttt{HERange}]$), is shown in Figure 4. Recall that this assumption is the result of step 3 in Algorithm II.1; thus it encodes the *error behaviour* of the perception in terms of estimated and ground-truth (actual) output values for the DNN.

Let us consider how Algorithm III.1 synthesizes local, non-temporal specifications using this assumption. For instance, in state Q3, estimates $\{[0][0..2], [1][0..1], [2][1]\}$ lead to error, thus only estimates $[1][2]$, $[2][0]$, and $[2][2]$ are safe. Furthermore, Q3 is reached (from Q1) when the actual state is $[2][2]$. Following similar reasoning, in state Q5, estimates $[0][2]$, $[1][0]$, $[1][2]$, $[2][0..2]$ are safe (since estimates $\{[0][0..1], [1][1]\}$ lead to error) and Q5 is reached when actual is $[2][0]$. Similar patterns can be observed for Q7, Q10, Q12, and Q14.

From Q3, we can infer the following local specification for the DNN: $(\texttt{cte}^* = 2 \wedge \texttt{he}^* = 2) \Rightarrow ((\texttt{cte} = 1 \wedge \texttt{he} = 2) \vee (\texttt{cte} = 2 \wedge \texttt{he} = 0) \vee (\texttt{cte} = 2 \wedge \texttt{he} = 2))$. Here '*' denotes actual state values. This specification gets translated back to the original, continuous DNN outputs as follows: $(\texttt{cte}^* \in [2.7, 8) \wedge \texttt{he}^* \in (11.66, 35.0]) \Rightarrow ((\texttt{cte} \in [-2.7, 2.7] \wedge \texttt{he} \in (11.66, 35.0]) \vee (\texttt{cte} \in [2.7, 8) \wedge \texttt{he} \in [-11.67, 11.66]) \vee (\texttt{cte} \in [2.7, 8) \wedge \texttt{he} \in (11.66, 35.0]))$. This specification can be interpreted as follows. For an input image that has ground truth $\texttt{cte}^* \in [2.7, 8) \wedge \texttt{he}^* \in (11.66, 35.0]$, the output of the DNN on that image should satisfy $(\texttt{cte} \in [-2.7, 2.7] \wedge \texttt{he} \in (11.66, 35.0]) \vee (\texttt{cte} \in [2.7, 8) \wedge \texttt{he} \in [-11.67, 11.66]) \vee (\texttt{cte} \in [2.7, 8) \wedge \texttt{he} \in (11.66, 35.0])$.

We note that this local specification tolerates some error in the estimation. Unlike in the standard DNN testing objective, which requires that for every input image, the output of the DNN should match the ground truth, our local specification tolerates DNN output values that are different than the ground truth, as they do not affect the safety of the overall system.

MaxCTE	Assumption size	M_1 size	Time (seconds)	Memory (KB)
2	7	99	0.079	9799
4	13	261	0.126	10556
6	19	495	0.098	9926
14	43	2151	0.143	13324
30	91	8919	0.397	31056
50	151	23859	2.919	45225
100	301	92709	81.529	132418

TABLE I: Effect of discretization granularity on assumptions.

V. Evaluation

a) ***Assumptions for Increasing Alphabet Sizes****:* Our approach is independent on the granularity of the discretization used for the system states (represented by cte and he); however, this granularity defines the size of the interface alphabet and can thus affect the scalability of the approach. To assess the scalability of assumption generation in the context of our problem, we generated assumptions for TaxiNet under different sizes of the interface alphabet; in particular, we provide results for different values of MaxCTE (defining the granularity for cte); the granularity of he stays the same. The results are shown in Table I.

We first note that the generated assumptions are much smaller than the corresponding M_1 components. For instance, for $\texttt{MaxCTE} = 2$, M_1 has 99 states (and 155 transitions) while the assumption is much smaller (7 states); it appears for this problem, the assumption size is linear in the size of the interface alphabet, making them good candidates for efficient run-time monitoring. The results indicate that the assumption generation

is effective even when the size of the interface alphabet—corresponding to the number of possible DNN output values—is large. For instance, when `MaxCTE` $= 100$, it means that `cte` has 101 intervals while `HE` has 3 intervals, thus the DNN can be seen as having $101 * 3 = 303$ possible discrete output values. The generated assumption has 301 states and the assumption generation is reasonably fast. The results indicate that our approach is promising in handling practical applications, even for DNNs (classifiers) with hundreds of possible output values.

In case assumption generation stops scaling, we can group multiple DNN output values into a single (abstract) value, guided by the logic of the downstream decision making components (similar to how we group together multiple continuous DNN output values into discrete values representing intervals in the TaxiNet example). Incremental techniques, that use learning and alphabet refinement [17] can also help alleviate the problem and we plan to explore them in the future.

*b) **Assumptions as Run-time Safety Monitors**:* For our TaxiNet case study, we explored the use of the automatically generated assumptions as run-time safety monitors that guarantee the satisfaction of the required safety properties by the autonomous system. The goal of our evaluation is to: (i) check that the TaxiNet system augmented with the safety monitor is guaranteed to be safe, (ii) quantify the permissivity of the monitor, i.e., the probability of the the assumption being violated during system operation.

To this end, we devised an experiment that leverages probabilistic model checking using the PRISM tool [18]. We built PRISM models for the TaxiNet $Controller$ and $Dynamics$ components that are equivalent to the corresponding LTSs encoded in the FSP language. We also had to encode the $Sensor$ and $Perception$ components since our goal is to study the behavior of the overall system. For this purpose, we use our prior work [4] to build a conservative probabilistic abstraction of $M_2 = Sensor \,||\, Perception$ that maps every actual system state value to a probability distribution over estimated system state values; the probabilities associated with the transitions from actual to estimated values are empirically derived from running the DNN on a representative data set provided by the industrial partner (11,108 images).

PRISM results. We first double-checked that the PRISM model of the TaxiNet system, augmented with the run-time monitor, does not violate the two safety properties, which PRISM confirmed, validating the correctness of our approach. We also analyzed a PCTL [18] property, $P =?[F(Q = -1)]$, that asks for the probability of the system reaching the state $\mathtt{Q} = -1$, thereby quantifying the permissiveness of the run-time monitor. The results for this property are shown in Figure 5 for two different versions of the $Perception$ DNN that vary in their accuracies. While the probability of violating the assumption as the horizon length increases tends towards 1 for both the DNNs, the rate of growth for the higher accuracy DNN is much slower. Developers can use the analysis to evaluate the quality of the trained DNN from the perspective of abort frequency.

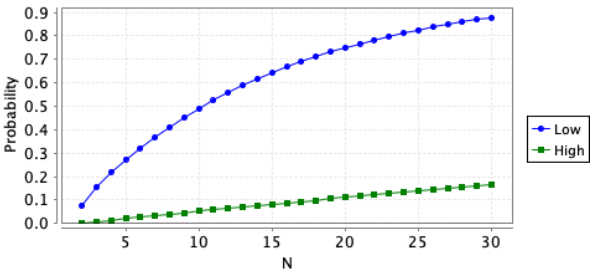

Fig. 5: Probability of the assumption being violated. N indicates horizon length. Low and High correspond to lower and higher accuracy DNNs.

VI. Related Work

There are several approaches for formally proving safety properties of autonomous systems with low-dimensional sensor readings [1, 19, 20, 21, 22, 23, 24]; however, they are intractable for systems that use rich sensors producing high-dimensional inputs such as images. More closely related works aim to build models based on the analysis of the perception components. However, they either do not provide guarantees [3] or do not scale to large networks [1].

The most closely related approach is the one in [5], which builds abstractions of the DNN components that guarantee the safety of the overall system. The method can not provide any strong system-level guarantees; instead they only provide a probabilistic result that measures empirically how *close* a real DNN is to the abstraction. Another difference is that the approach in [5] uses the training data to help discover the right abstraction, whereas we do not rely on any data.

In recent work [4], we built a probabilistic abstraction of the camera and perception DNN for the probabilistic analysis of the same TaxiNet case study. The approach facilitates obtaining *probabilistic* guarantees with respect to the satisfaction of safety properties of the entire system. In contrast, we focus here on obtaining strong (non-probabilistic) safety guarantees.

The work in [2] aims to verify the safety of the trajectories of a camera-based autonomous vehicle in a given 3D-scene. Their abstraction captures only one environment condition (i.e., one scene) and a one camera model, whereas our approach is not particular to any camera model and implicitly considers all the possible environment conditions

In a previous white paper [25], we advocated for a compositional framework with input-output DNN contracts obtained from a DNN-specific analysis. However in that work, we left open the problem of how to precisely relate the contracts to the system level properties. We solve that problem here, where instead of input-output contracts for a DNN, we derive assumptions that are based solely on the outputs of the DNN. The assumptions are derived without a DNN-specific analysis.

Our work is also related to safe shielding for reinforcement learning [26]. That work does not consider complex DNNs as part of the system and therefore does not discuss suitable techniques for them. Nevertheless, we note that our assumptions are monitoring the outputs of the DNN instead of the actions of the controller, and can thus be used to prevent errors earlier. They also act as local specifications for the DNN behaviour, enabling other activities such as testing or training.

VII. Conclusion and Future Work

We presented a compositional approach based on automated assumption generation for the verification of autonomous systems that use DNNs for perception. We demonstrated our approach on the TaxiNet case study. While our approach opens the door to analyzing autonomous systems with state-of-the-art DNNs, it can suffer from the well known scalability issues associated with model checking (due to state-explosion). We believe we can address this issue via judicious use of abstraction and compositional techniques. Incremental, more scalable, techniques for assumption generation can also be explored, see. e.g. [27].

We have presented our approach in the context of components modeled as LTSs. However, such components are often modeled as more complex hybrid automata. We believe that our proposed approach can be extended to reasoning about such systems, leveraging our previous work on assumption generation for hybrid automata [14].

Another direction for future work is to investigate autonomous systems with multiple sensors (e.g., both camera and LIDAR) and to develop techniques that decompose the generated assumptions into local specifications for each sensors. These local specifications can then be used to guide the development of the different sensing mechanisms and can also be deployed as sensor-specific run-time monitors.

References

[1] U. Santa Cruz and Y. Shoukry, "Nnlander-verif: A neural network formal verification framework for vision-based autonomous aircraft landing," in *NASA Formal Methods Symposium*. Springer, 2022, pp. 213–230.

[2] H. P, N. Deka, D. D'Souza, K. Lodaya, and P. Prabhakar, "Verification of camera-based autonomous systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1–1, 2023.

[3] S. M. Katz, A. L. Corso, C. A. Strong, and M. J. Kochenderfer, "Verification of image-based neural network controllers using generative models," *Journal of Aerospace Information Systems*, vol. 19, no. 9, pp. 574–584, 2022.

[4] C. S. Pasareanu, R. Mangal, D. Gopinath, S. G. Yaman, C. Imrie, R. Calinescu, and H. Yu, "Closed-loop analysis of vision-based autonomous systems: A case study," 2023 (to appear at CAV'23).

[5] C. Hsieh, Y. Li, D. Sun, K. Joshi, S. Misailovic, and S. Mitra, "Verifying controllers with vision-based perception using safe approximate abstractions," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 11, pp. 4205–4216, 2022.

[6] S. Ghosh, Y. V. Pant, H. Ravanbakhsh, and S. A. Seshia, "Counterexample-guided synthesis of perception models and control," in *2021 American Control Conference (ACC)*. IEEE, 2021, pp. 3447–3454.

[7] X. Huang, D. Kroening, W. Ruan, J. Sharp, Y. Sun, E. Thamo, M. Wu, and X. Yi, "A survey of safety and trustworthiness of deep neural networks: Verification, testing, adversarial attack and defence, and interpretability," *Computer Science Review*, vol. 37, p. 100270, 2020.

[8] G. Katz, D. A. Huang, D. Ibeling, K. Julian, C. Lazarus, R. Lim, P. Shah, S. Thakoor, H. Wu, A. Zeljic, D. L. Dill, M. J. Kochenderfer, and C. W. Barrett, "The marabou framework for verification and analysis of deep neural networks," in *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I*, 2019, pp. 443–452.

[9] D. Gopinath, G. Katz, C. S. Pasareanu, and C. W. Barrett, "Deepsafe: A data-driven approach for checking adversarial robustness in neural networks," *CoRR*, vol. abs/1710.00486, 2017.

[10] C. S. Pasareanu, D. Giannakopoulou, M. G. Bobaru, J. M. Cobleigh, and H. Barringer, "Learning to divide and conquer: applying the l* algorithm to automate assume-guarantee reasoning," *Formal Methods Syst. Des.*, vol. 32, no. 3, pp. 175–205, 2008. [Online]. Available: https://doi.org/10.1007/s10703-008-0049-6

[11] D. Giannakopoulou, C. S. Pasareanu, and H. Barringer, "Assumption generation for software component verification," in *17th IEEE International Conference on Automated Software Engineering (ASE 2002), 23-27 September 2002, Edinburgh, Scotland, UK*. IEEE Computer Society, 2002, pp. 3–12. [Online]. Available: https://doi.org/10.1109/ASE.2002.1114984

[12] D. Giannakopoulou and J. Magee, "Fluent model checking for event-based systems," in *Proceedings of the 11th ACM SIGSOFT Symposium on Foundations of Software Engineering 2003 held jointly with 9th European Software Engineering Conference, ESEC/FSE 2003, Helsinki, Finland, September 1-5, 2003*, J. Paakki and P. Inverardi, Eds. ACM, 2003, pp. 257–266. [Online]. Available: https://doi.org/10.1145/940071.940106

[13] D. Giannakopoulou and C. S. Pasareanu, "Abstraction and learning for infinite-state compositional verification," in *Semantics, Abstract Interpretation, and Reasoning about Programs: Essays Dedicated to David A. Schmidt on the Occasion of his Sixtieth Birthday, Manhattan, Kansas, USA, 19-20th September 2013*, ser. EPTCS, A. Banerjee, O. Danvy, K. Doh, and J. Hatcliff, Eds., vol. 129, 2013, pp. 211–228. [Online]. Available: https://doi.org/10.4204/EPTCS.129.13

[14] S. Bogomolov, G. Frehse, M. Greitschus, R. Grosu, C. S. Pasareanu, A. Podelski, and T. Strump, "Assume-guarantee abstraction refinement meets hybrid systems," in *Hardware and Software: Verification and Testing - 10th International Haifa Verification Conference, HVC 2014, Haifa, Israel, November 18-20, 2014. Proceedings*, ser. Lecture Notes in Computer Science, E. Yahav, Ed., vol. 8855. Springer, 2014, pp. 116–131. [Online]. Available: https://doi.org/10.1007/978-3-319-13338-6_10

[15] "X-plane flight simulator." [Online]. Available: https://www.x-plane.com/

[16] Y. Yang, Q. Zu, W. Ke, M. Zhang, and X. Li, "Real-time system modeling and verification through labeled transition system analyzer," *IEEE Access*, vol. 7, pp. 26 314–26 323, 2019.

[17] M. Gheorghiu, D. Giannakopoulou, and C. S. Pasareanu, "Refining interface alphabets for compositional verification," in *Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference, TACAS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007 Braga, Portugal, March 24 - April 1, 2007, Proceedings*, ser.

Lecture Notes in Computer Science, O. Grumberg and M. Huth, Eds., vol. 4424. Springer, 2007, pp. 292–307. [Online]. Available: https://doi.org/10.1007/978-3-540-71209-1_23

[18] M. Kwiatkowska, G. Norman, and D. Parker, "PRISM 4.0: Verification of probabilistic real-time systems," in *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*, ser. LNCS, G. Gopalakrishnan and S. Qadeer, Eds., vol. 6806. Springer, 2011, pp. 585–591.

[19] R. Ivanov, K. Jothimurugan, S. Hsu, S. Vaidya, R. Alur, and O. Bastani, "Compositional learning and verification of neural network controllers," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 20, no. 5s, pp. 1–26, 2021.

[20] R. Ivanov, J. Weimer, R. Alur, G. J. Pappas, and I. Lee, "Verisig: verifying safety properties of hybrid systems with neural network controllers," in *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control*, 2019, pp. 169–178.

[21] R. Ivanov, T. J. Carpenter, J. Weimer, R. Alur, G. J. Pappas, and I. Lee, "Verifying the safety of autonomous systems with neural network controllers," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 20, no. 1, pp. 1–26, 2020.

[22] R. Ivanov, T. Carpenter, J. Weimer, R. Alur, G. Pappas, and I. Lee, "Verisig 2.0: Verification of neural network controllers using taylor model preconditioning," in *International Conference on Computer Aided Verification*. Springer, 2021, pp. 249–262.

[23] C. Dawson, B. Lowenkamp, D. Goff, and C. Fan, "Learning safe, generalizable perception-based hybrid control with certificates," *IEEE Robotics and Automation Letters*, vol. 7, no. 2, pp. 1904–1911, 2022.

[24] C. Dawson, S. Gao, and C. Fan, "Safe control with learned certificates: A survey of neural lyapunov, barrier, and contraction methods," *arXiv preprint arXiv:2202.11762*, 2022.

[25] C. S. Pasareanu, D. Gopinath, and H. Yu, "Compositional verification for autonomous systems with deep learning components," *CoRR*, vol. abs/1810.08303, 2018. [Online]. Available: http://arxiv.org/abs/1810.08303

[26] M. Alshiekh, R. Bloem, R. Ehlers, B. Könighofer, S. Niekum, and U. Topcu, "Safe reinforcement learning via shielding," 2017.

[27] J. M. Cobleigh, D. Giannakopoulou, and C. S. Pasareanu, "Learning assumptions for compositional verification," in *Tools and Algorithms for the Construction and Analysis of Systems, 9th International Conference, TACAS 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings*, ser. Lecture Notes in Computer Science, H. Garavel and J. Hatcliff, Eds., vol. 2619. Springer, 2003, pp. 331–346. [Online]. Available: https://doi.org/10.1007/3-540-36577-X_24

```
//M2 = Sensor || Perception
[] cte=1 & pc=1 → 0.962: (cte_est'=1) & (pc'=2) + 0.002: (
    cte_est'=0) & (pc'=2) + 0.036: (cte_est'=2) & (pc'=2);
[] cte=0  & pc=1 → 0.681: (cte_est'=1) & (pc'=2) + 0.319: (
    cte_est'=0) & (pc'=2);
[] cte=2 & pc=1 → 0.398: (cte_est'=1) & (pc'=2) + 0.602: (
    cte_est'=2) & (pc'=2);

[] he=0 & pc=2 → 0.675: (he_est'=0) & (pc'=3) + 0.304: (
    he_est'=1) & (pc'=3) + 0.021: (he_est'=2) & (pc'=3);
[] he=1 & pc=2 → 0.043: (he_est'=0) & (pc'=3) + 0.957: (
    he_est'=1) & (pc'=3);
[] he=2 & pc=2 → 0.377: (he_est'=0) & (pc'=3) + 0.107: (
    he_est'=1) & (pc'=3) + 0.516: (he_est'=2) & (pc'=3);
```

```
// Safety Monitor
[] Q=0 & pc=3 & ((cte_est=0 & he_est=2)|(cte_est=1 & he_est
    =0)|(cte_est=2 & he_est=1)) → 1: (Q'=0) & (pc'=4);
[] Q=0 & pc=3 & ((cte_est=0 & he_est<2)|(cte_est=1 & he_est
    =1)) → 1: (Q'=1) & (pc'=4);
[] Q=0 & pc=3 & ((cte_est=1 & he_est=2)|(cte_est=2 & (
    he_est=0|he_est=2))) → 1: (Q'=4) & (pc'=4);

[] Q=1 & pc=3 & ((cte_est=0)|(cte_est=1 & he_est<2)|(
    cte_est=2 & he_est=1)) → 1: (Q'=-1) & (pc'=6);
[] Q=1 & pc=3 & ((cte_est=1& he_est=2)|(cte_est=2 & (he_est
    =0|he_est=2))) → 1: (Q'=2) & (pc'=4);

[] Q=2 & pc=3 & ((cte_est=0 & he_est<2)|(cte_est=1 & he_est
    =1)) → 1: (Q'=-1) & (pc'=6);
[] Q=2 & pc=3 & ((cte_est=0 & he_est=2)|(cte_est=1 & he_est
    =0)|(cte_est=2 & he_est=1)) → 1: (Q'=2) & (pc'=4);
[] Q=2 & pc=3 & ((cte_est=1 & he_est=2)|(cte_est=2 & (
    he_est=0|he_est=2))) → 1: (Q'=3) & (pc'=4);
...
```

Fig. 6: TaxiNet M_2 and safety monitor in PRISM.

APPENDIX

a) **PRISM *Encoding for TaxiNet with Safety Monitor*:** We show the PRISM code for M_2 and the safety monitor in Figure 6. We use the output of step 4 in procedure `BuildAssumption` (Section II-0b) as a safety monitor, i.e., the assumption LTS has both err and $sink$ states, with a transition to err state interpreted as the system aborting.

The PRISM encoding of the safety monitor closely follows the transitions of the assumption computed for M_1 over alphabet $\Sigma'_I = Est$, with an additional $abort$ (or fail-safe) state added to trap the output behaviours of the DNN that violate the assumption and potentially lead to safety violations of the overall system.

In the code, variable `pc` encodes a program counter. M_2 is encoded as mapping the actual system state (represented with variables `cte` and `he`) to different estimated states (represented with variables `cte_est` and `he_est`). The transition probabilities are empirically estimated based on profiling the DNN; for simplicity we update `cte_est` and `he_est` in sequence. The monitor maintains its state using variable `Q` (initially `0`); it transitions to its next state after `cte_est` and `he_est` have been updated; the abort state (`Q` $= -1$) traps behaviours that are not allowed by the assumption; there are no outgoing transitions from such an abort state.