Title
Querying Labelled Data with Scenario Programs for Sim-to-Real Validation

ISBN
9781665409674

Authors
Kim, Edward
Shenoy, Jay
Junges, Sebastian
et al.

Publication Date
2022-05-06

DOI
10.1109/iccps54341.2022.00010

Peer reviewed

Querying Labelled Data with Scenario Programs for Sim-to-Real Validation

Edward Kim, Jay Shenoy
University of California at Berkeley
USA

Sebastian Junges
Radboud University
Nijmegen, Netherlands

Daniel J. Fremont
University of California at Santa Cruz
USA

Alberto Sangiovanni-Vincentelli, Sanjit A. Seshia
University of California at Berkeley
USA

ABSTRACT

Simulation-based testing of autonomous vehicles (AVs) has become an essential complement to road testing to ensure safety. Consequently, substantial research has focused on searching for failure scenarios in simulation. However, a fundamental question remains: are AV failure scenarios identified in simulation *meaningful* in reality — i.e., are they reproducible on the real system? Due to the sim-to-real gap arising from discrepancies between simulated and real sensor data, a failure scenario identified in simulation can be either a spurious artifact of the synthetic sensor data or an actual failure that persists with real sensor data. An approach to validate simulated failure scenarios is to identify instances of the scenario in a corpus of real data, and check if the failure persists on the real data. To this end, we propose a formal definition of what it means for a labelled data item to match an abstract scenario, encoded as a scenario program using the SCENIC probabilistic programming language. Using this definition, we develop a querying algorithm which, given a scenario program and a labelled dataset, finds the subset of data matching the scenario. Experiments demonstrate that our algorithm is accurate and efficient on a variety of realistic traffic scenarios, and scales to a reasonable number of agents.

1 INTRODUCTION

Simulation-based testing is becoming a core element of assessing the safety of autonomous vehicles (AVs) by government and industry. For example, the National Highway Traffic Safety Administration has stated that self-driving technology should be tested in simulation before deployment [1], and Waymo recently used simulation to support the claim that self-driving cars are safer than human drivers [2]. A number of open-source simulation environments designed to support automated AV testing are available [3–5], as well as simulators which focus on realistic rendering of specific types of sensors such as LiDAR and radar [6, 7]. There are also a variety of black-box and white-box techniques to *search* for failure scenarios causing an AV to violate its safety specifications (e.g. [8–13]).

Despite these advances, the fundamental question: "Are AV failures scenarios found in simulation *meaningful*, i.e., are they reproducible on the real system?" has not been answered. Because of the so-called "sim-to-real gap" — the discrepancy between simulated and real sensor data and physics — a failure scenario found in simulation can be spurious, with no similar failure being possible for the real system operating on real data. There is a need for techniques to *validate* a simulated scenario against real sensor data. For brevity, we will refer to this problem as the *scenario validation* problem.

Previous work on bridging the sim-to-real gap has focused on *training* [14–17] AV components, such as perception and behavior prediction modules, on simulated data only and demonstrating that they perform robustly on real data. However, there is less work on *testing*, e.g., to investigate how simulated AV behaviors transfer to real roads *without* having to physically reconstruct these scenarios, an activity that is labor-intensive and not scalable. In contrast, we propose a *data-driven* approach for the scenario validation problem.

To tackle this problem, we leverage two technological trends in the AV domain. First, both in academia and industry, there are active community efforts to collect and share large amounts of real sensor data with high quality *labels* [18–21]. Second, AV companies are *mapping* road infrastructure, gathering detailed geometric information about intersections, lanes, sidewalks, etc. Several such efforts have produced open-source datasets where real sensor data is synchronized with map information, such as nuScenes [22] and Argoverse [23]. Given these trends, in this paper, we assume we have access to *a large set of labelled real sensor data and the corresponding map information*.

Based on this assumption, we propose to validate a candidate failure scenario by *querying* a dataset to find real-world instances of the scenario. Specifically, we want a query algorithm that, given (1) a formal description of a failure scenario previously identified in simulation and (2) a set of labelled real sensor data with map information, outputs the subset of labels (and, therefore, of real sensor data) that *matches* the scenario description. Developers could then test whether the AV failure is reproducible on this subset of real data to validate the failure scenario, as illustrated in Fig. 1. If the query returns an empty subset, then we have identified a *rare* scenario not present in our corpus. Hence, our work also enables simulation-based testing to guide both real-road testing and data collection by identifying which rare scenarios to focus on.

In order to make the query problem precise and provide correctness guarantees, we must formally define what it means for a label to "match" a scenario. We consider labels which provide positions and orientations for each object, and potentially other semantic features (see Sec. 4 for details). We formally model scenarios as programs in the SCENIC probabilistic programming language [24], which is designed for scenario specification. In this paper, we deal with *static* scenarios describing the environment of the system at one moment in time, which is sufficient for testing many AV perception systems. We can then define a label as matching a scenario if it has nonzero probability under the distribution defined by the SCENIC program. We develop an algorithm that checks this condition by encoding it as a series of *Satisfiability Modulo Theories* (SMT) [25] problems.

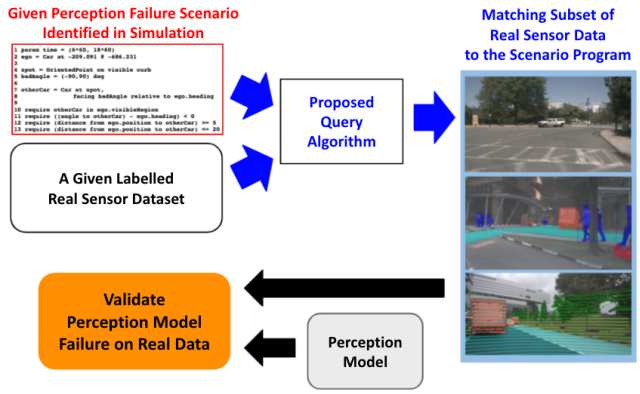

Figure 1: Using our proposed query algorithm to validate whether a failure of an AV's perception system previously identified in simulation persists in reality.

```
param weather = Uniform('sunny','rainy')
param time = Range(10, 12) # pm

ego = Car on road, facing roadDirection
otherCar = Car ahead of ego by Range(4,10) #meters
require not (otherCar in intersection)
```

Figure 2: A SCENIC program describing a car ahead of the ego car by 4–10 meters and which is not in an intersection.

Because our algorithm queries labels, it is applicable to data of any type from which geometry can be extracted, including RGB, LiDAR, and radar data (for simplicity we refer to "images" henceforth). Consequently, it can be used to validate failure scenarios for a wide variety of AV static perception tasks (e.g. classification, detection, segmentation) utilizing various sensor types.

Contributions. In this paper, we provide:

- A novel formulation of the problem of querying a labelled dataset against a formal scenario description, enabling simulated failure scenarios for a perception model to be validated against real-world counterparts;
- A sound SMT-based algorithm solving the query problem for static scenarios given by SCENIC programs, which can be used to validate scenarios across a wide variety of perception tasks and sensor types;
- Experiments demonstrating the accuracy and scalability of our algorithm on realistic AV scenarios.

We start with an overview of our approach in Sec. 2. Section 3 presents background allowing us to formally define the query problem in Sec. 4. We describe our algorithm in Sec. 5 and our experiments in Sec. 6. Finally, we discuss related work in Sec. 7 and overall conclusions in Sec. 8.

2 OVERVIEW

Suppose we wish to query a label against the simple SCENIC program in Fig. 2. The program describes, for some fixed map with information about roads and intersections, a scenario where there is a car ahead of the ego vehicle but not in an intersection. A label consists of semantic features such as time of day, weather conditions, and positions and orientations of vehicles. For this particular example, assume that our (simplified) label l consists of features $l_{e_x}, l_{e_y}, l_{e_h}, l_{e_{cl}}$ denoting the xy-coordinates, heading, and object class (e.g. car or pedestrian) of the ego car respectively, and, likewise, features $l_{c_x}, l_{c_y}, l_{c_h}, l_{c_{cl}}$ for the other car.

When we query a label against this program, we want to answer the question: *is the situation specified by the label an instance of the scenario in Fig. 2?* More precisely, we ask whether the label can be obtained by instantiating the random variables in the scenario (i.e. for some choice of the weather, time, ego's position "on road", and a distance between 4–10 meters).

Our approach to this problem is summarized in Fig. 3. Before we go into details, let us clarify that in a nutshell, the approach is to translate the SCENIC program and the label into constraints represented as a Satisfiability Modulo Theory (SMT) formula (see Sec. 3.2 for an overview of SMT). The resulting formula will be satisfiable if and only if the given label matches the program. While the problem can be captured by a single (monolithic) SMT formula, the size of this formula increases linearly with the size of the scenario, and for even relatively simple scenarios can exceed the capabilities of state-of-the-art SMT solvers. To alleviate the scalability problem, we take advantage of the structure of the SCENIC program to decompose it into several SMT formulas, determining incrementally whether parts of the label match the program. Below, we discuss the key stages of Fig. 3 in more detail.

Expression forest. To enable the decomposition process, we operate on the internal representation of a SCENIC program. The SCENIC compiler converts a program into an *expression forest*: a simplified forest for the program in Fig. 2 is shown in Fig. 4. The expression forest is made up of a set of expression trees, each of which essentially corresponds to the syntax used to define the distribution of one of the semantic features. For example, the leftmost tree in Fig. 4 shows that the ego's position is defined to be a uniformly random point in the `road` region of the map.

Dependency analysis. Expression trees can have dependency relations with each other, as shown by the bold blue arrows in Fig. 4. Such dependencies naturally occur in scenario modeling. For example, to compute the `otherCar`'s position (which is `ahead of` the ego car), we need to first know the ego's position and heading. Our dependency analysis uses the SCENIC expression forest to sort the list of semantic features in dependency order: in Fig. 4, the ego's position is first since it depends on no other features, while the `otherCar`'s heading is last.

Modular translation. Next, we *modularly* translate each expression tree and *incrementally* check its consistency with the label in dependency order. In our example, we start with the ego's position, symbolically representing its coordinates by variables e_x and e_y. We encode its definition given by the leftmost expression tree in Fig. 4 with the constraint $\texttt{On}(\texttt{RoadRegion}, e_x, e_y)$, where `On` is a predicate requiring that the point (e_x, e_y) is contained in the polygon that defines the `RoadRegion`. We can then check whether the value of this feature given in the label is in fact possible in the SCENIC

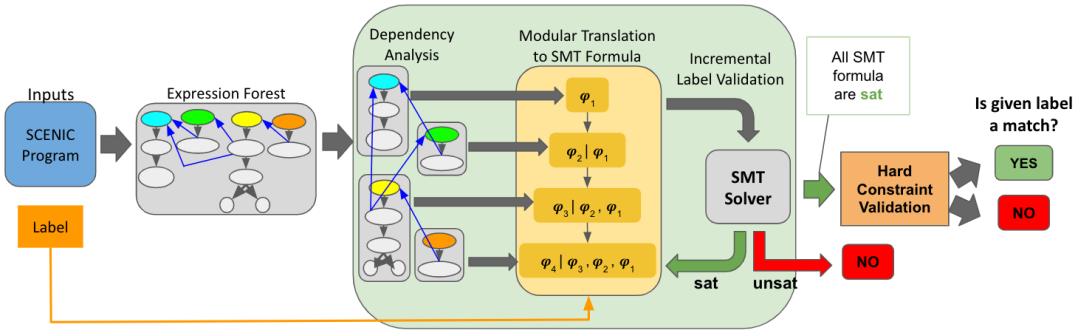

Figure 3: An overview of our algorithm to determine if a label matches a SCENIC program.

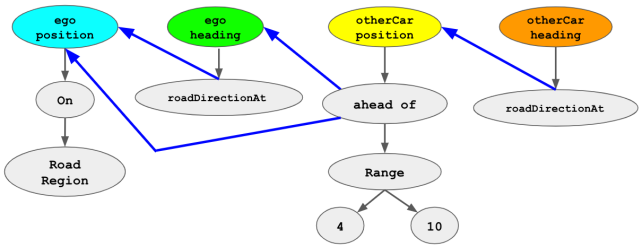

Figure 4: A partial expression forest for the SCENIC program in Fig. 2. The 4 top nodes represent the semantic features, and the blue arrows show dependencies among them.

program by checking the satisfiability of the SMT formula

$$\varphi_1 = \texttt{On}(\texttt{RoadRegion}, e_x, e_y) \wedge (e_x = l_{e_x}) \wedge (e_y = l_{e_y})$$

where as above l_{e_x} and l_{e_y} are the label's concrete values for e_x and e_y, and $\wedge$ represents a logical AND.

Incremental validation. If the formula is unsatisfiable, the observed label cannot be generated by the SCENIC program, and so does not match the scenario. If the formula *is* satisfiable, then this first semantic feature is consistent with the scenario and we can move on. To avoid reasoning about this feature again, we *condition* the ego position expression tree by replacing it with the actual position value from the label. This conditioning will apply when we proceed to check the consistency of the next feature in our order, namely the ego's heading. Translating the second expression tree to SMT yields the formula

$$e_x{=}l_{e_x} \wedge e_y{=}l_{e_y} \wedge (e_h{=}\texttt{roadDir}(e_x, e_y)) \wedge (e_h{=}l_{e_h})$$

which we call $\varphi_2|\varphi_1$ to indicate the conditioning. Note that the ego position's expression tree, involving the `On` predicate, is no longer present, simplifying the formula: this is the essence of our incremental approach. We then solve the new SMT formula, and repeat this modular feature translation and incremental validation process until either all the features in the label are validated, or a feature is invalidated. If all the features are valid, then as a final step we condition all the features as in the label and check whether all the explicit constraints in the SCENIC program (e.g. the `require` statement in Fig. 2) are satisfied. If so, the label matches the scenario; otherwise, it does not.

3 BACKGROUND

3.1 SCENIC: A Scenario Description Language

SCENIC [24, 26] is an object-oriented probabilistic programming language designed for *modeling* and *generating* scenarios. Because SCENIC is probabilistic, a single SCENIC program can model *abstract* scenarios like "bumper-to-bumper traffic" that cover a wide range of environments, parametrized by semantic features such as weather, time of day, objects' positions, orientations and color. A SCENIC program defines a distribution over *scenes*, configurations of the environment at one point in time, defined by assigning concrete values to the semantic features (SCENIC can also define *dynamic* scenarios [26], but here we consider only static scenarios). A scene is thus equivalent to a *label* as described above, and we use the terms interchangeably. The ability of a SCENIC abstract scenario to generate many scenes will be useful to us when querying, since we can search for data matching a broad description of a scenario. For the rest of the paper, we refer to abstract scenarios as simply *scenarios* for brevity.

To aid in scenario modeling, SCENIC supports an intuitive syntax to specify complex geometric relations among objects: the example SCENIC program in Fig. 2 uses the `on` construct for generating a point uniformly at random within a region, and the `ahead of` construct for placing one object relative to another. In addition, SCENIC enables a user to impose declarative constraints on the distribution of scenes, as in the last line of Fig. 2, using a `require` statement. Here, the constraint succinctly captures the user's intention for the scenario without requiring an explicit description of the parts of the road that are sufficiently far from intersections. An execution of the SCENIC program *samples* a scene that must satisfy any `require` statements specified in the program.

In order to use constructs like `road` and `intersection` in Fig. 2, SCENIC requires basic geometric information about the road network being simulated. More sophisticated scenarios involving details of the infrastructure (e.g. layout of lanes and crosswalks) require correspondingly detailed maps. SCENIC is able to read standard map formats including those provided with the datasets used in this paper.

3.2 Satisfiability Modulo Theories (SMT)

The satisfiability problem is the question of whether a propositional formula has a solution, i.e., whether there is an assignment to the

Boolean variables that makes the formula evaluate to true. Over the last three decades, satisfiability solvers have made tremendous progress despite the theoretical hardness of the problem, scaling to formulas with billions of variables [27].

To support problems that involve, e.g., arithmetic, SMT solvers ask for a given a first-order formula φ over a set of variables and a fixed theory[1] whether there is an assignment to the variables that makes the formula evaluate to true [25]. In this paper, we use a fragment of the theory of quantifier-free nonlinear real arithmetic, with formulas generated by the following grammar:

$$e ::= x \mid a \mid e + e \mid e \times e \mid -e \mid sin(e)$$
$$c ::= e < 0 \mid e \leq 0 \mid c \wedge c \mid c \vee c \mid \neg c$$

Here, x is a real-valued variable, a a real-valued constant, and all operators have their standard meanings. The satisfiability problem for such formulas is undecidable in general (due to the presence of trigonometric functions) [28]. However, solvers such as dReal [29] can either prove unsatisfiability or return a variable assignment which *approximately* satisfies the formula (in a suitable formal sense; see Gao et al. [29] for details). Thus, in this paper, we do not consider the question of decidability further.

4 PROBLEM STATEMENT

Let a *label* l consist of (1) a set of objects O, (2) a set of semantic features F, and (3) a function $s\colon O \times F \to V$ which maps an object's semantic features to concrete values. The domain of values V can include real numbers, integers, categorical values (for examples, see Sec. 2) and the special value $\perp$, which indicates that the object does not have the corresponding semantic feature. Let $[\![P]\!]$ denote the *support* of a SCENIC program P, i.e., the set of labels that can be generated by P. Then, the problem to solve could be:

Problem P0: *Given a* SCENIC *program P and a label l, is $l \in [\![P]\!]$?*, i.e., is the probability (density) of l under the distribution defined by P greater than 0?

However, the above problem statement is too strict for our purposes. In particular, it may answer "no" for a label $l = (O, F, s)$ which is not in $[\![P]\!]$ even if an essentially equivalent label $l' = (O', F', s')$ *is* in $[\![P]\!]$. This can happen in two ways:

(1) The semantic feature spaces of l and l' may differ, i.e., $F \neq F'$. For instance, a car's color and model might be included in F but not in F'. In such cases, we will only consider features in $F \cap F'$ as being relevant to our query.

(2) More importantly, l may contain multiple objects, and the correspondence between O and O' is unknown; in fact, we may want to consider l as matching P even if it contains additional objects not having any counterpart in P (for example, we may want a program for "two perpendicular cars" to match whenever two such cars exist, even if there is a third car in the vicinity).

Therefore, we generalize **P0** as follows.

Problem P1: Let a label l *match* a program P if there exists $l' = (O', F', s') \in [\![P]\!]$ and an injective[2] function $C\colon O' \to O$ such that $\forall o' \in O', \forall f \in F \cap F', s'(o', f) = s(C(o'), f)$. Here the function C, which we call the *object correspondence*, maps each object in P to a distinct object in l. Then: *Given program P and label l, determine whether l matches P.*

This definition supports matching noisy labels: simply modify the SCENIC program to add bounded perturbations (or more realistic noise models) to any semantic features that might be noisy.

Our problem formulation can be applied to validate the behaviors of models for different perception tasks using various sensor types (e.g. LiDAR segmentation, RGB detection) across synthetic and real data. Because we query the labels, our problem formulation broadly applies to querying various labelled sensor types such as 2D RGB, radar, 3D LiDAR point clouds, etc. For example, it is possible that for the same label, data from multiple sensors exists. Fig. 5 shows 2D RGB image and 3D LiDAR point cloud data that corresponds to the same label. As we query labels rather than the raw sensor data, we can query both sensor types.

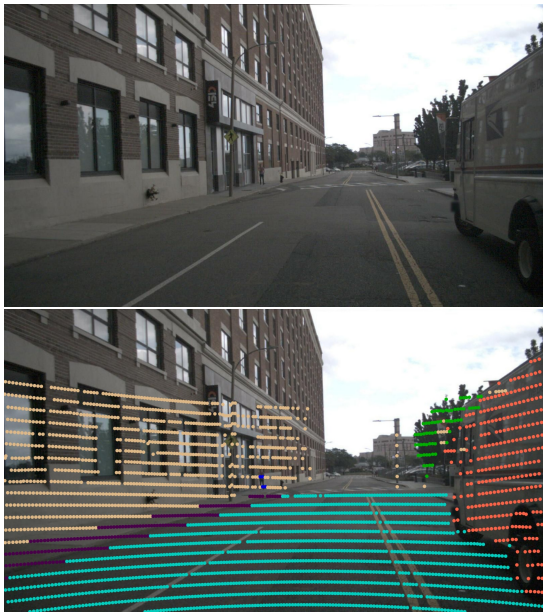

Figure 5: The benefit of querying a label is that we can retrieve corresponding sensor data of multiple types. Here, our algorithm finds both an RGB image and a segmented 3D LiDAR point cloud corresponding to a single label.

Assumptions on the Labels and their Sources. To use our methodology in the context of validating sensor data, the sensor data must be adequately labelled. Specifically, at minimum the labels must contain information about the position, orientation, and type (e.g. pedestrian, car, bicycle) of each object. Furthermore, the label information needs to be aligned with the map information (i.e. share the same coordinate system and global orientation). Labels may also include more diverse global or object-specific semantic features, allowing more expressive descriptions of scenes. For example, global semantic features could include weather and time of day (which determines the location of the sun and, therefore, of shadows). Object semantic features could include color and physical dimensions.

[1] A theory fixes the interpretation of the operators, e.g. + meaning addition and $|x|$ denoting the absolute value of x.

[2] If we want to require an *exact* match in the sense that l does not contain any objects beyond those defined in P, we can also require C to be surjective. Our algorithm extends to this case with only trivial changes.

To query AV scenarios involving road geometry, we require corresponding map information: specifically, polygons for regions such as lanes, roads, and intersections, as well as the traffic flow directions within those regions. This information is needed to interpret SCENIC constructs such as `on road`, seen in our example above. More detailed information may be needed depending on the scenes one may model. For example, if one wishes to model scenes involving cyclists on a bicycle lane in a SCENIC program, a polygonal region and traffic flow direction for bicycle lanes in the map must be provided. For more details, see Sec. 3.1.

The above two assumptions are needed for querying labels. However, for our motivating application of validating a failure scenario of a perception model, the real labelled dataset should also contain relevant ground truth labels to evaluate the perception task (e.g. segmentation, detection). For example, if the task is 3D detection, the 3D bounding boxes should be included in the label. Then, once we retrieve a matching subset of real data, the perception model can be evaluated on this subset with the relevant ground truth labels.

There are open-source datasets that satisfy the two assumptions for querying and also provide ground truth labels sufficient for various perception tasks, such as nuScenes [22] and Argoverse [23]; we used nuScenes for our experiments.

5 METHODOLOGY

Given a label and a SCENIC program, the key idea behind our approach is to translate the program to an SMT formula that is satisfied if and only if the label matches the program.

5.1 Monolithic Approach

The basic set-up of the SMT formula, as outlined in Sec. 2, defines variables for all objects and semantic features in the intersection of program and the label. There are three aspects to the SMT formula; the first aspect of the formula maps objects in the label with objects in the program (which we referred to as *correspondence* in Sec. 4), the second aspect describes the constraints over all the semantic features in the program as dictated by the semantics of SCENIC, and the third aspect asserts that the semantic feature values observed in the label satisfies those corresponding constraints in the program.

The SCENIC compiler represents a compiled SCENIC program as an *expression forest*. Each semantic feature is the root node of an *expression tree* appearing in the forest, which captures the semantics of how the feature's value is derived from the values of other features and random parameters. To encode the semantics of the SCENIC program, we walk the expression forest, generating SMT equivalents of each of the nodes. For example, the `On` node in Fig. 4 is encoded by a set of constraints enforcing that the variable representing the ego's position must lie in the `road` region[3]. All SCENIC constructs[4] can be encoded as real arithmetic constraints fairly easily, following the SCENIC semantics outlined in [24]. The details of our SMT encoding and the fragment of SCENIC we support are explained in Appendix A.

[3]This is done by triangulating the region, and using a disjunction of linear inequalities to assert that the position lies in one of the triangles. See Appendix A for more details.

[4]Our implementation handles a large fragment of SCENIC, supporting its built-in operators and functions. However, it does not support the inclusion of arbitrary Python code, which SCENIC allows in some contexts.

Algorithm 1: Determining if SCENIC program P matches label l

```
EF ← Compile(P) // get expression forest
SortedFeatures ← AnalyzeDependencies(EF, l)
badOCs ← ∅ // partial correspondences that don't work
for all possible object correspondences C do
    if C extends a correspondence in badOCs then
        continue; // skip this correspondence
    failed ← false
    for nextFeatures ∈ SortedFeatures do
        φ ← TranslateSMT(nextFeatures, l, C, EF)
        if Satisfiable(φ) then
            Condition(nextFeatures, l, C, EF)
        else
            failed ← true
            Uncondition(EF) // reset forest
            add the used part of C to badOCs
            break
    if (not failed) and SatisfiesHardConstraint(l) then
        return Yes
return No
```

This method yields a sizeable SMT formula, which is difficult to solve beyond toy examples. Therefore, we consider a modular approach outlined below.

5.2 Modular and Incremental Approach

Our approach is formalized as Algorithm 1. The input to the algorithm is a SCENIC program P and a label l, and the output is whether l matches P. The algorithm has three main steps: (1) dependency analysis of objects and their features, (2) incremental translation and validation of the program to a series of SMT formulas, and (3) validation of hard constraints. We now discuss each step in detail.

5.2.1 Object/Feature Dependency Analysis. A key feature of our approach is to exploit dependency structure in the SCENIC program, specifically its compiled expression forest, to split the monolithic SMT query into smaller parts. We define two types of dependencies in the expression forest: *dependent* and *jointly dependent*. If an expression tree of a feature, X, has a reference to another feature, Y, then X is *dependent* on Y. Such dependencies are *acyclic* because a feature that is specified first in SCENIC cannot reference an object defined afterwards. If two or more feature expression trees share internal nodes which are not features, then those features are *jointly dependent*. These shared node(s) are intermediate (i.e., unobserved) variables which are not part of the scene. Therefore, to check whether there exists a feasible value for the intermediate variables, jointly dependent features must be considered in the same SMT query. For example, Fig. 6 shows a SCENIC program describing a distribution of scenes with two cars positioned in parallel, adjacent to a spot uniformly randomly selected from a curb region. In this case, the ego and side car's position features are *jointly* dependent on the intermediate variable, `spot`. Note that `spot` is an internal variable of the SCENIC program and would not appear in a label. Hence, we need to encode both cars into the SMT query to check if there exists a value of `spot` which satisfies the constraints given

```
spot = OrientedPoint on curb
ego = Car at (spot offset by (Range(2,4), Range(5,10)))
sideCar = Car left of spot by Range(1,3)
```

Figure 6: A SCENIC program with an intermediate variable, `spot`, shared between position features of `ego` and `sideCar` objects.

their labelled positions. Note that for the following dependency analysis among semantic features, as we stated in Sec. 4, we are only analyzing semantic features that exist both in the SCENIC program and the label. Hence, this dependency analysis takes as input both the expression forest of the given SCENIC program and the label (line 1 of Alg. 1).

Once we identify these dependency relations across all object features, we sort the features in dependency order, giving jointly dependent features the same rank in the order (line 2 in Alg. 1). For example, analyzing the expression forest in Fig. 4, with no jointly dependent features, yields the order [{`ego` position}, {`ego` heading}, {`otherCar` position}, {`otherCar` heading}]. For the SCENIC program in Fig. 6 with jointly dependent features, our analysis outputs [{`ego` position, `sideCar` position}, {`ego` heading}, {`sideCar` heading}].

5.2.2 Modular and Incremental SMT Translation. Given the sorted feature dependency list obtained as above, we translate only one feature expression tree (or multiple trees, if jointly dependent) at a time, in the order of dependency (lines 8-9 in Alg. 1). Checking the resulting SMT query, if the formula is unsatisfiable then it is impossible for the current features to take on their observed values and so the label does not match the program (lines 12-13). If instead the formula is satisfiable, then the observed values are feasible given the semantics of the program, and we need not consider the current features further: we *condition* the expression forest on their observed values, substituting the values in for their expression trees (lines 10-11). Then we move on to the next feature(s) in the dependency order and repeat. If a previously-checked feature is referenced by a later expression tree, we do not need to encode it again, since it now has a constant value. This modular approach can significantly simplify the generated SMT queries: for example in Fig. 4, instead of one query encoding the entire forest, we have one query for each of the 4 top nodes encoding only the nodes directly below it in the order of dependency (refer to Sec. 2 for more detail).

We remark that our modular translation requires fixing the correspondence of label and program objects *a priori*. As seen in the outermost loop of Alg. 1 (line 4), we currently brute-force enumerate all possible combinations, with one refinement: if a partial correspondence is already enough to make the SMT query unsatisfiable, we can exclude all further correspondences extending it. In particular, when a feature of object O fails the SMT check under correspondence C, the part of C consisting of all objects up to and including O (in dependency order) will also yield unsatisfiability. So we maintain a set of partial correspondences known to fail (line 15) and skip any correspondence which extends one of them (line 6). Our experiments show that this approach scales to a reasonable number of objects.

A further note is that encoding polygonal regions of the map (e.g. lanes, roads, intersections) can yield large formulas. For example, nuScenes, the dataset used for our experiments, provides a map of the city of Boston. Encoding the entire road network of Boston would be impractical; however, since we are only interested in the scene around the ego vehicle, which determines the reference viewpoint of sensors (e.g. camera, LiDAR), it suffices to encode only a neighborhood of the ego. We can extract bounds on the visible distance from the SCENIC program and only encode the region of the map within that radius.

5.2.3 Hard Constraint Validation. Finally, if the SCENIC program contains any `require` statements encoding hard constraints (see Sec. 3.1), we need to check that these are satisfied by the label. After all features have been validated and conditioned on their observed values, we simply check that the `require` constraints all evaluate to true. Note that if any `require` constraints were jointly dependent (see Sec. 5.2) with any feature, for soundness we would have to encode the constraints into the SMT query for that feature. Since the SCENIC compiler currently does not generate expression trees for requirements, we instead assume that the program does not have any such joint dependencies (restricting our SCENIC fragment; see Appendix A). This assumption holds for the vast majority of the scenarios in the SCENIC distribution [24].

Our proposed query algorithm is sound as stated in Theorem 1 (see Appendix B for a proof).

THEOREM 1. *Given a label and a* SCENIC *program, the* SCENIC *query algorithm outputs* `Yes` *if and only if the label matches the program as defined in Sec. 4; otherwise, it outputs* `No` *(assuming the underlying SMT solver correctly answers all queries).*

6 EXPERIMENTS

Recall in Sec. 1 we motivated the query problem with the application of validating failure scenarios. Once our algorithm queries a labelled, real dataset with a scenario encoded in SCENIC and retrieves a matching subset, then validating a perception model's behavior on that subset is straightforward (assuming the dataset also contains the relevant ground truth labels for the perception task: see Sec. 4). In fact, the most time-consuming aspect in this validation process is executing the query. Thus, to evaluate how useful our algorithm is, we ask the following questions:

(1) Given a SCENIC program and a real labelled dataset, does the algorithm efficiently find the matching data points?
(2) Does the output of the algorithm correspond with the intuitive notion of scenario matching?
(3) How does the algorithm scale with scenario complexity, in terms of number of agents and program structure?

The second question is important to address since it directly relates to the *interpretability* of the scenario validation process, which is crucial for debugging. Therefore, we need to check whether our algorithm operates in a manner intuitive to humans.

To answer these questions, we conducted two different experiments. First, our efficacy experiment answers the first two questions. In a nutshell, it demonstrates that the formal querying problem we define and solve corresponds well to our intuition of what it means to match an image against a high-level scenario and is efficient in comparison to manual querying. Our second experiment demonstrates that our approach remains feasible even on fairly large scenarios.

Figure 7: Matching images for Scenarios 1 through 4 (left to right), queried using our algorithm.

Table 1: For several scenarios, the number of images identified by 3 human subjects (unanimously) and our algorithm.

Scenario #	1	2	3	4	5
Matching images (humans)	42	5	0	2	0
Matching images (our algorithm)	58	7	2	2	0

In both experiments, we used the dReal SMT solver [29] with its default parameter settings. For the first experiment, we set the ego visible distance to 50 meters; for the second, we set it to 200 meters to accommodate the larger number of agents.

6.1 Efficacy Experiment

Setup. There is no baseline or benchmark to which we can compare our algorithm since there are no existing algorithms for the problem of querying with a formal scenario description, or open-source autonomous driving image datasets that provide detailed formal scenario descriptions with which to test our algorithm. Hence, we asked 3 human participants to manually query a set of images with 5 different scenarios and then compared their results with the outputs of our algorithm. We asked each participant to select 5 subsets of images matching more detailed versions of the natural language descriptions of our test scenarios below. To acquire the most accurate queried subsets, we kept only the images which all 3 humans agreed matched for each scenario. We then compared these subsets with those returned by our algorithm.

Scenarios. We used five scenarios, involving 2–4 agents and a variety of realistic traffic situations. Here we provide natural language descriptions; the SCENIC encodings are shown in Figs. 12–16. Several example matching images are shown in Fig. 7.

(1) A pedestrian in an intersection facing nearly perpendicularly or towards the ego.
(2) Two vehicles in an intersection, travelling perpendicular to the ego.
(3) A rare, hazardous situation, where the ego vehicle is driving against traffic and another vehicle is visible within 10 meters.
(4) Four vehicles in a typical situation on a two-lane road, with two vehicles going in each direction.
(5) A cut-in scenario where a car in the adjacent lane to the right cuts in front of the ego.

Data. We use a selection of RGB images from nuScenes [22]. nuScenes provides the map of Boston where the images were collected; our scenarios used map information about intersection and lane regions as well as traffic flow directions. nuScenes labels contain three semantic features per object: its position, heading, and class. The object classes include vehicles, pedestrians, and static objects such as traffic cones. nuScenes also defines a special object class, “ego vehicle”, indicating the reference viewpoint, meaning the camera for image collection is mounted on it. As we will describe below, our experiment required humans to identify matches between programs and labelled images; to avoid objects being missed by the humans due to visual occlusion, we filtered out images containing more than 4 objects. After filtering, we randomly selected 700 images from the subset, which we believe is a reasonably large dataset for human participants to manually query.

Note that the semantic features in the label and the program determines the matching real data. In our experiment, we limited our query to use only semantic features included in the nuScenes labels, namely position, heading, and type for each object.

Results. Our results, summarized in Table 1, show that our algorithm corresponds to the intuitive notion of scenario matching. For all 5 scenarios, our algorithm correctly returned all images identified by the human participants, and, in some cases, found additional matching images that they missed. For Scenarios 2 and 3, our algorithm found 4 additional images that our participants missed by mistake. An example missed image for Scenario 3 is shown in Fig. 7. For Scenario 1, our algorithm identified 16 more images than the participants; however, upon visual inspection, 8 of these did not match the scenario description and the remaining were missed by our participants. Investigating further, we found that these errors were caused by inaccurate labels in nuScenes, e.g. a pedestrian’s position being in an intersection according to the label but being on a sidewalk near an intersection in the image. Our algorithm correctly identified such labels as matching Scenario 1, even though the sensor data disagreed. This illustrates a limitation of our approach in that it hinges on the accuracy of the provided labels. Finally, for Scenario 5, both the human participants and the algorithm agreed that there are no matching scenes in the given real dataset. This shows the strength of our algorithm in identifying "rare" scenarios with respect to a given dataset. If an AV perception model fails on such rare scenarios in simulation, this finding can guide the real-world data collection process to *systematically* gather more examples of such scenarios.

The algorithm’s runtime over all 700 labels ranged from 7 minutes (Scenario 1) to 40 minutes (Scenario 5) depending on the complexity of scenarios. The human participants took on average about 1 hour to complete their manual queries on the five provided scenarios. This result demonstrates that our algorithm can replace the arduous task of manually querying matching real sensor data for scenario validation with higher accuracy, provided that the labels are accurate.

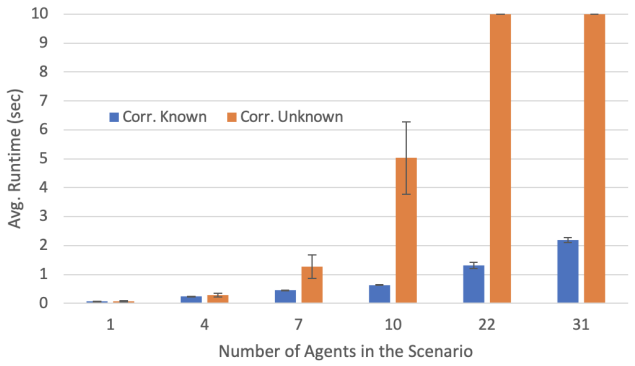

Figure 8: Runtime results for scaling number of agents in the SCENIC program shown in Fig. 10. Two runtimes represent cases when the object correspondence between the program and the label is known versus unknown.

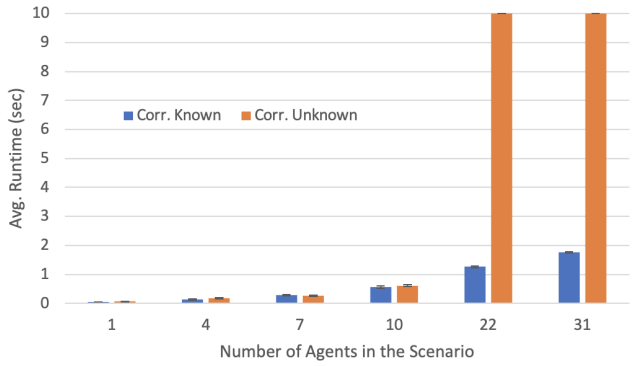

Figure 9: Runtime results for scaling number of agents in the SCENIC program shown in Fig. 11. Two runtimes represent cases when the object correspondence between the program and the label is known versus unknown.

6.2 Scalability Experiment

To test the scalability of our algorithm, we used two additional syntactically-rich SCENIC programs, shown in Fig. 10 and 11. We increased the scenario complexity by scaling the number of agents while maintaining the same program structure. For evaluation, we generated 10 labels from each SCENIC program for a range of numbers of agents (from 1 to 31).

Figures 8 and 9 show the average runtimes for querying these 10 labels with our algorithm for each number of agents (with a 10-second timeout). In each plot, we give separate runtimes for the cases where the correspondence between the objects in the SCENIC program and the label is known and unknown respectively. For the unknown case, we randomly shuffled the ordering of the objects in the labels.

We observe a consistent behavior from these two plots. When the correspondence is unknown, the runtime increases exponentially in the number of agents due to the combinatorial object correspondence matching process in our algorithm (refer to Sec. 5.2). On the other hand, when the object correspondence is known, we consistently observed that the runtime of the query algorithm increases approximately linearly in the number of agents, and scales to a sizeable number.

```
def placeObjs(car, numCars):
    for i in range(numCars):
        car = Car ahead of car by Range(4, 5)
        leftCar = Car left of car by Normal(2,0.1),
            facing roadDirection
        rightCar = Bicycle right of car by Normal(3,0.1),
            facing Range(0,10) deg relative to ego.heading
    return leftCar, rightCar

spawn_point = 207.26 @ 8.72
ego = Car at spawn_point,
        with visibleDistance 200

leftCar, rightCar = placeObjs(ego, 2)
require (distance to leftCar) < 200
require (distance to rightCar) < 200
```

Figure 10: A SCENIC program used for our scalability experiment, modeling bumper-to-bumper traffic. The number of vehicles in the scenario is scaled by increasing `numCars`.

```
def placeObjs(numPeds):
    for i in range(numPeds):
        Pedestrian offset by Range(-5,5) @ Range(0,200),
            facing Range(-120, 120) deg relative to ego.heading

spawn_point = 207.26 @ 8.72
ego = Car at spawn_point,
        with visibleDistance 200

placeObjs(3)
```

Figure 11: A SCENIC program used for our scalability experiment, modeling a parade scenario where pedestrians are walking on a street. The number of pedestrians in the scenario is scaled by increasing `numPeds`.

7 RELATED WORK

The most related study to ours is Fremont et al. [30], where failure scenarios of an autopilot in simulation were validated by *physically* reconstructing them at a track testing facility. However, this manual validation approach is labor-intensive and not scalable. Our approach aims to automate such validation in a data-driven manner.

Domain adaptation [31] aims to reduce the sim-to-real gap in the context of *training*: the objective is primarily to obtain good performance of a perception model for a particular task (e.g. segmentation, detection, localization) on *real* sensor data despite only training on simulated sensor data [14–17]. Generative adversarial networks (GANs) [32] have been a key technique employed to *adapt*, or convert, synthetic data to more realistic data of various types such as RGB images, 3D LiDAR point cloud, etc. These data are used for training. On the contrary, our paper aims to reduce the gap in the context of *testing*, investigating for a *pre-trained* model whether it behaves differently in the same scenario across two different domains: simulated and real data.

```
ego = Car on drivableRoad,
        facing Range(-15,15) deg relative to roadDirection,
        with visibleDistance 50,
        with viewAngle 135 deg
ped = Pedestrian on roadsOrIntersections,
        with regionContainedIn roadRegion,
        facing Range(-180, 180) deg

require abs(relative heading of ped from ego) > 70 deg
```

Figure 12: Scenario #1 in the Efficacy Experiment

```
ego = Car on drivableRoad,
        facing Range(-15,15) deg relative to roadDirection,
        with visibleDistance 50,
        with viewAngle 135 deg
other1 = Car on intersection,
            facing Range(50,135) deg relative to ego.heading
other2 = Car on intersection,
            facing -1 * Range(50,135) deg relative to ego.heading

require abs(relative heading of other1 from other2) > 100 deg
require (distance from ego to intersectionRegion) < 10
```

Figure 13: Scenario #2 in the Efficacy Experiment

Visual question answering (VQA) [33, 34] considers answering questions about static images phrased in natural language. The VQA area combines approaches common in captioning with a large natural language processing component: part of the challenge is to understand the question. Much like VQA, we decide whether an image matches a given query. However, our queries are expressed using a formal probabilistic programming language and we query the *label*, not the sensor data. This allows us to formulate a well-defined querying problem and develop an algorithm which is guaranteed to be sound: the returned subset of images are exactly those which match the scenario program, if the labels are accurate.

Our approach can be seen as a specialised form of inference in probabilistic programming languages (PPLs). SCENIC allows making probabilistic assertions of propositional statements, e.g., 'the car is within the visible region'. Such declarative *hard constraints* make scenario modeling much easier and more intuitive. Some PPLs, e.g. ANGLUIN, actively prevent specifying hard constraints to prevent programmers from 'accidentally' posing NP-hard questions [35]. PPLs such as PYRO [36] and EDWARD [37] use Bayesian inference schemes that require tracking derivatives [38–41]. Some PPLs allow hard constraints and non-continuity, but either have (significant) restrictions or limited efficiency [42–44], putting trigonometry and continuous domains out of reach. Moreover, the typical inference task is to compute posterior distributions relative to a prior, whereas we are primarily interested in filtering using a Boolean membership query. Finally, we note that while sampling-based approaches may do well in answering membership queries positively, they are not well suited for providing negative answers.

8 CONCLUSION

We proposed an algorithm to query a labeled dataset using a scenario program encoded in the SCENIC language. This algorithm can be used to shrink the gap between simulation-based and real-world testing by identifying counterparts of simulated scenarios in real data,

```
offset = Uniform(-1,1) * Range(90, 180) deg

ego = Car on drivableRoad,
        facing offset relative to roadDirection,
        with visibleDistance 50,
        with viewAngle 135 deg

otherCar = Car on visible road,
            facing Range(-15, 15) deg relative to roadDirection

require (distance from ego to otherCar) < 10
```

Figure 14: Scenario #3 in the Efficacy Experiment

```
ego = Car on drivableRoad,
        facing Range(-15, 15) deg relative to roadDirection,
        with visibleDistance 50,
        with viewAngle 135 deg

point1 = OrientedPoint ahead of ego by Range(0,40)
Car at (point1 offset by Range(-1,1) @ 0),
        facing Range(-15, 15) deg relative to roadDirection

oppositeCar = Car offset by (Range(-10, -1), Range(0, 50)),
        facing Range(140, 180) deg relative to ego.heading

point2 = OrientedPoint ahead of oppositeCar by Range(0,40)
Car at (point2 offset by Range(-1,1) @ 0),
        facing Range(-15, 15) deg relative to roadDirection
```

Figure 15: Scenario #4 in the Efficacy Experiment

```
lanesWithRightLane = filter(lambda i: i._laneToRight, network.laneSections)
egoLane = Uniform(*lanesWithRightLane)

ego = Car on egoLane,
        facing Range(-15,15) deg relative to roadDirection
cutInCar = Car offset by Range(0, 4) @ Range(0, 5),
            facing -1* Range(15, 30) deg relative to roadDirection
```

Figure 16: Scenario #5 in the Efficacy Experiment

which can then be used to validate the fidelity of the simulations. More broadly, our algorithm enables a principled way to explore and understand the range of scenarios present in a dataset by expressing scenarios of interest in a formal language. In future work, we plan to explore using program analysis and other techniques to alleviate the combinatorial explosion when the object correspondence is unknown, and to generalize our algorithm to support *dynamic* scenarios, as well as *probabilistic* queries that take the likelihood of labels into account.

ACKNOWLEDGMENTS

This work was supported in part by National Science Foundation grants CNS-1545126 (VeHICaL), CNS-1739816, and CCF-1837132, by DARPA contracts FA8750-16-C0043 (Assured Autonomy) and FA8750-20-C-0156 (Symbiotic Design of Cyber-Physical Systems), by Berkeley Deep Drive, by Toyota through the iCyPhy center, and by the Toyota Research Institute.

REFERENCES

[1] NHTSA, "Automated driving systems," https://www.nhtsa.gov/vehicle-manufacturers/automated-driving-systems, 2021, accessed: 2021-09-02.

[2] A. J. Hawkins, "Waymo simulated real-world crashes to prove its self-driving cars can prevent deaths," https://www.theverge.com/2021/3/8/22315361/waymo-autonomous-vehicle-simulation-car-crash-deaths, 2021, published: 2021-03-08.

[3] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun, "CARLA: An open urban driving simulator," in *Proceedings of the 1st Annual Conference on Robot Learning*, 2017, pp. 1–16.

[4] G. Rong, B. H. Shin, H. Tabatabaee, Q. Lu, S. Lemke, M. Možeiko, E. Boise, G. Uhm, S. M. Mark Gerow, E. Agafonov, T. H. Kim, E. Sterner, K. Ushiroda, M. Reyes, D. Zelenkovsky, and S. Kim, "Lgsvl simulator: A high fidelity simulator for autonomous driving," *arXiv:2005.03778*, 2020.

[5] S. Shah, D. Dey, C. Lovett, and A. Kapoor, "Airsim: High-fidelity visual and physical simulation for autonomous vehicles," *Field and Service Robotics(FSR) Conference*, 2017.

[6] S. Manivasagam, S. Wang, K. Wong, a. Zeng, M. Sazanovich, S. Tan, B. Yang, W.-C. Ma, and R. Urtasun, "Lidarsim: Realistic lidar simulation by leveraging the real world," in *Computer Vision and Pattern Recognition (CVPR)*, 2020.

[7] A. Gruber, M. Gadringer, H. Schreiber, D. Amschl, W. Bösch, S. Metzner, and H. Pflügl, "Highly scalable radar target simulator for autonomous driving test beds," in *European Radar Conference*, 2017.

[8] G. E. Mullins, P. G. Stankiewicz, and S. K. Gupta, "Automated generation of diverse and challenging scenarios for test and evaluation of autonomous vehicles," in *International Conference on Robotics and Automation*, 2017.

[9] M. Koren, S. Alsaif, R. Lee, and M. J. Kochenderfer, "Adaptive stress testing for autonomous vehicles," in *Intelligent Vehicles Symposium*, 2018.

[10] M. O'Kelly, A. Sinha, H. Namkoong, J. Duchi, and R. Tedrake, "Scalable end-to-end autonomous vehicle testing via rare-event simulation," in *Neural Information Processing Systems (NeurIPS)*, 2018.

[11] C. E. Tuncali, G. Fainekos, H. Ito, and J. Kapinski, "Simulation-based adversarial test generation for autonomous vehicles with machine learning components," in *Intelligent Vehicles Symposium*, 2018.

[12] T. Dreossi, D. J. Fremont, S. Ghosh, E. Kim, H. Ravanbakhsh, M. Vazquez-Chanlatte, and S. A. Seshia, "VerifAI: A toolkit for the formal design and analysis of artificial intelligence-based systems," in *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I*, ser. Lecture Notes in Computer Science, I. Dillig and S. Tasiran, Eds., vol. 11561. Springer, 2019, pp. 432–442. [Online]. Available: https://doi.org/10.1007/978-3-030-25540-4_25

[13] E. Kim, D. Gopinath, C. Pasareanu, and S. Seshia, "A programmatic and semantic approach to explaining and debugging neural network based object detectors," in *Conference on Computer Vision and Pattern Recognition(CVPR)*. IEEE, 2020, pp. 11 125–11 134.

[14] M. Johnson-Roberson, C. Barto, R. Mehta, S. N. Sridhar, K. Rosaen, and R. Vasudevan, "Driving in the matrix: Can virtual worlds replace human-generated annotations for real world tasks?" *International Conference on Robotics and Automation (ICRA)*, 2017.

[15] X. Pan, Y. You, Z. Wang, and C. Lu, "Virtual to real reinforcement learning for autonomous driving," *Proceedings of the British Machine Vision Conference (BMVC)*, 2017.

[16] S. Zhao, B. Li, X. Yue, Y. Gu, P. Xu, R. Hu, H. Chai, and K. Keutzer, "Multi-source domain adaptation for semantic segmentation," *Conference on Neural Information Processing Systems (NeurIPS)*, 2019.

[17] Z. Murez, S. Kolouri, D. Kriegman, R. Ramamoorthi, and K. Kim, "Image to image translation for domain adaptation," *Computer Vision and Pattern Recognition (CVPR)*, 2018.

[18] F. Yu, H. Chen, X. Wang, W. Xian, Y. Chen, F. Liu, V. Madhavan, and T. Darrell, "Bdd100k: A diverse driving dataset for heterogeneous multitask learning," in *Computer Vision and Pattern Recognition*, 2020.

[19] M. Menze and A. Geiger, "Object scene flow for autonomous vehicles," in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015.

[20] P. Wang, X. Huang, X. Cheng, D. Zhou, Q. Geng, and R. Yang, "The apolloscape open dataset for autonomous driving and its application," *IEEE transactions on pattern analysis and machine intelligence*, 2019.

[21] P. Sun, H. Kretzschmar, X. Dotiwalla, A. Chouard, V. Patnaik, P. Tsui, J. Guo, Y. Zhou, Y. Chai, B. Caine, V. Vasudevan, W. Han, J. Ngiam, H. Zhao, A. Timofeev, S. Ettinger, M. Krivokon, A. Gao, A. Joshi, S. Zhao, S. Cheng, Y. Zhang, J. Shlens, Z. Chen, and D. Anguelov, "Scalability in perception for autonomous driving: Waymo open dataset," in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020.

[22] H. Caesar, V. Bankiti, A. H. Lang, S. Vora, V. E. Liong, Q. Xu, A. Krishnan, Y. Pan, G. Baldan, and O. Beijbom, "nuScenes: A multimodal dataset for autonomous driving," *arXiv arXiv:1903.11027*, 2019.

[23] M.-F. Chang, J. Lambert, P. Sangkloy, J. Singh, S. Bak, A. Hartnett, D. Wang, P. Carr, S. Lucey, D. Ramanan, and J. Hays, "Argoverse: 3d tracking and forecasting with rich maps," in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019.

[24] D. J. Fremont, T. Dreossi, S. Ghosh, X. Yue, A. L. Sangiovanni-Vincentelli, and S. A. Seshia, "Scenic: a language for scenario specification and scene generation," in *PLDI*. ACM, 2019, pp. 63–78.

[25] C. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli, "Satisfiability modulo theories," in *Handbook of Satisfiability*. IOS Press, 2009, ch. 26, pp. 825–885.

[26] D. J. Fremont, E. Kim, T. Dreossi, S. Ghosh, X. Yue, A. L. Sangiovanni-Vincentelli, and S. A. Seshia, "Scenic: a language for scenario specification and data generation," *Machine Learning*, 2022. [Online]. Available: https://doi.org/10.1007/s10994-021-06120-5

[27] J. M. Silva, I. Lynce, and S. Malik, "Conflict-driven clause learning SAT solvers," in *Handbook of Satisfiability*, ser. Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press, 2009, pp. 131–153.

[28] D. Richardson, "Some undecidable problems involving elementary functions of a real variable," *J. Symb. Log.*, vol. 33, no. 4, pp. 514–520, 1968.

[29] S. Gao, S. Kong, and E. Clarke, "dReal: an SMT solver for nonlinear theories over the reals," in *International Conference on Automated Deduction(CADE)*, 2013.

[30] D. J. Fremont, E. Kim, Y. V. Pant, S. A. Seshia, A. Acharya, X. Bruso, P. Wells, S. Lemke, Q. Lu, and S. Mehta, "Formal scenario-based testing of autonomous vehicles: From simulation to the real world," in *23rd IEEE International Conference on Intelligent Transportation Systems, ITSC 2020, Rhodes, Greece, September 20-23, 2020*. IEEE, 2020, pp. 1–8. [Online]. Available: https://doi.org/10.1109/ITSC45102.2020.9294368

[31] M. Wang and W. Deng, "Deep visual domain adaptation: A survey," *Neurocomputing*, vol. 312, pp. 135–153, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0925231218306684

[32] A. Creswell, T. White, V. Dumoulin, K. Arulkumaran, B. Sengupta, and A. A. Bharath, "Generative adversarial networks: An overview," *IEEE Signal Processing Magazine*, vol. 35, no. 1, pp. 53–65, 2018.

[33] Y. Goyal, T. Khot, D. Summers-Stay, D. Batra, and D. Parikh, "Making the V in VQA matter: Elevating the role of image understanding in visual question answering," *International Journal of Computer Vision*, vol. 127, no. 4, pp. 398–414, 2019.

[34] H. Xu and K. Saenko, "Ask, attend and answer: Exploring question-guided spatial attention for visual question answering," in *ECCV (7)*, ser. Lecture Notes in Computer Science, vol. 9911. Springer, 2016, pp. 451–466.

[35] F. Wood, J. W. van de Meent, and V. Mansinghka, "A new approach to probabilistic programming inference," in *AISTATS*, ser. JMLR Workshop and Conference Proceedings, vol. 33. JMLR.org, 2014, pp. 1024–1032.

[36] E. Bingham, J. P. Chen, M. Jankowiak, F. Obermeyer, N. Pradhan, T. Karaletsos, R. Singh, P. Szerlip, P. Horsfall, and N. D. Goodman, "Pyro: Deep universal probabilistic programming," *J. Mach. Learn. Res.*, vol. 20, pp. 28:1–28:6, 2019.

[37] D. Tran, M. D. Hoffman, R. A. Saurous, E. Brevdo, K. Murphy, and D. M. Blei, "Deep probabilistic programming," in *International Conference on Learning Representations(ICLR)*. OpenReview.net, 2017.

[38] M. D. Hoffman and A. Gelman, "The no-u-turn sampler: adaptively setting path lengths in hamiltonian monte carlo," *J. Mach. Learn. Res.*, vol. 15, no. 1, pp. 1593–1623, 2014.

[39] D. Wingate and T. Weber, "Automated variational inference in probabilistic programming," *CoRR*, vol. abs/1301.1299, 2013. [Online]. Available: http://arxiv.org/abs/1301.1299

[40] A. Kucukelbir, R. Ranganath, A. Gelman, and D. M. Blei, "Automatic variational inference in stan," in *Neural Information Processing Systems(NIPS)*, 2015.

[41] A. Kucukelbir, D. Tran, R. Ranganath, A. Gelman, and D. M. Blei, "Automatic differentiation variational inference," *J. Mach. Learn. Res.*, vol. 18, pp. 14:1–14:45, 2017.

[42] A. Nori, C.-K. Hur, S. Rajamani, and S. Samuel, "R2: an efficient MCMC sampler for probabilistic programs," in *Association for the Advancement of Artificial Intelligence (AAAI)*. AAAI Press, 2014, pp. 2476–2482.

[43] B. Gram-Hansen, Y. Zhou, T. Kohn, H. Yang, and F. Wood, "Discontinuous hamiltonian monte carlo for probabilistic programs," *CoRR*, vol. abs/1804.03523, 2018.

[44] S. Holtzen, G. V. den Broeck, and T. D. Millstein, "Dice: Compiling discrete probabilistic programs for scalable inference," *Object-Oriented Programming, Systems, Languages & Applications(OOPSLA)*, vol. abs/2005.09089, 2020.

[45] Scratchapixel, "Ray tracing: Rendering a triangle," https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-rendering-a-triangle/barycentric-coordinates, 2021, last Access: 2021-10-23.

A SMT ENCODING OF THE SCENIC FRAGMENT

We support a large fragment of SCENIC including 26 different position and heading specifiers as well as a wide variety of operators, which together provide an expressive language to flexibly model a diverse range of scenarios. Specifically, the SCENIC fragment supported by our SMT encoding, which enables our query algorithm, includes all (static) SCENIC syntax except for:

(1) the operators `following F [from V] for S`, `angle [from X] to Y`, and `apparent heading of X [from Y]`;
(2) `require` statements referring to variables (i.e. semantic features) not present in the label;
(3) imports of external Python libraries.

To succinctly present our encoding, we first introduce some notations. First, we denote SCENIC semantics with double brackets, $[\![\cdot]\!]$, and denote the encoding of an expression into an SMT term as $E(\cdot)$. For example, to access a position of a object O as we would in SCENIC, we write $[\![O.position]\!]$ which is equivalent to the object's xy-coordinates, $\langle O_x, O_y \rangle$. SCENIC employs an ego-centric syntax, meaning it requires that `ego` be defined and its syntax, by default, assumes ego as a reference object if not otherwise specified. We will use O to represent a SCENIC object, heading as H, vector (i.e. position) as V. $\top$ denotes True. Basic notations are shown in Fig. 17.

A SCENIC expression can be categorized into three types: (1) built-in functions, (2) predicates, and (3) distributions. For each expression, we create a new SMT variable, encode any constraints on its value implied by the SCENIC semantics, and gather these variables and encoded constraints throughout each incremental encoding process. For example, to encode the expression `Range(2,5)`, we create a new SMT variable *range1* and add the formula $2 \leq range1 \leq 5$ to our set of constraints. The SMT constraints for SCENIC's built-in distributions are shown in Fig. 18.

Second, all the specifiers and operators in our SCENIC fragment are built-in functions in SCENIC. Therefore, they can be abstractly represented in the following form: $f(a_1, ..., a_k)$ where the function, f, represents the SCENIC specifier or operator, and $a_1, ..., a_k$ are input arguments, where each input argument is a SCENIC expression. We first encode the input arguments to SMT terms and then encode the function according to the semantics of the specifier or operator as defined in Appendices C.2–C.5 of the SCENIC paper [24]. Formally, the encoding is defined by the relation $E(f(a_1, ..., a_k)) = [\![f]\!](E(a_1), ..., E(a_k))$. It is possible that the arguments of f may themselves consist of specifiers and operators with additional input arguments, creating a tree of syntax with its leaf nodes being constants and distributions. In such a case, we traverse down to the leaf nodes and recursively encode the tree toward the root node.

For example, the SMT formula for the position of `otherCar` in the SCENIC program in Fig. 2 has the form:
$[\![\text{ahead of}]\!](E(ego.position), E(ego.heading), E(Range(4, 10)))$.
Here, $E(ego.position)$ is evaluated as a pair of SMT variables $\langle x, y \rangle$, which must satisfy the constraint $[\![\texttt{On}]\!](\texttt{road}, \langle x, y \rangle)$. The semantics of the `On` predicate are in turn encoded into constraints requiring that $\langle x, y \rangle$ actually lie within the `road` region (as we will see below). Once all the arguments have been encoded as SMT terms (with associated constraints), we substitute them into $[\![\texttt{ahead of}]\!]$ to obtain the final term for the position of `otherCar`.

$$
\begin{aligned}
\langle x, y \rangle &= \text{point with the given XY coordinates} \\
E(V) &= \left\langle E(V_x), E(V_y) \right\rangle \\
E(V_1) \pm E(V_2) &= \left\langle E(V_{1,x}) \pm E(V_{2,x}), E(V_{1,y}) \pm E(V_{2,y}) \right\rangle \\
k * (E(V_1) \pm E(V_2)) &= \left\langle k * (E(V_{1,x}) \pm E(V_{2,x})), k * (E(V_{1,y}) \pm E(V_{2,y})) \right\rangle, \\
&\quad \text{where } k \in \mathbb{R} \\
\mathit{rotate}\,(\langle x, y \rangle, \theta) &= \langle E(x)\cos(E(\theta)) - E(y)\sin(E(\theta)), \\
&\quad E(x)\sin(E(\theta)) + E(y)\cos(E(\theta)) \rangle \\
\mathit{offsetLocal}\,(O, v) &= [\![\text{O.position}]\!] + \mathit{rotate}\,(v, [\![\text{O.heading}]\!]) \\
OP(V, \theta) &= \texttt{OrientedPoint} \text{ with position V and heading } \theta
\end{aligned}
$$

Figure 17: Notation used to define the SMT encoding of SCENIC syntax.

$$
\begin{aligned}
Range(l, u) &= E(l) \leq \mathrm{z} \leq E(u) \\
Normal(m, s) &= \top \\
Option(a_1, a_2, ..., a_n) &= (z == E(a_1)) \vee ... \vee (z == E(a_n))
\end{aligned}
$$

Figure 18: Encoding of SCENIC distributions, where z is the SMT variable representing the value sampled from the distribution.

Finally, the SMT encoding for the `On` region-containment predicate and associated operations on SCENIC regions are shown in Fig. 19. We encode containment within regions which are fixed (or which become fixed after conditioning) by triangulating the region. For non-fixed regions (discs and sectors), we generate constraints encoding the geometry of the region.

B PROOF SKETCH OF THEOREM 1

Due to the page limit on the Appendix, we provide a sketch of our proof.

Proposition 1: *Given a fixed object correspondence, the monolithic encoding algorithm returns* `Yes` *if and only if the label matches the program with that correspondence.*

Proof Sketch: According to the object correspondence, the label provides values for the semantic features of all the objects in the program. Since we disallow `require` statements referring to variables not present in the label, evaluating the requirements with the observed feature values is well-defined. If a requirement is violated, the label cannot match the program and the algorithm correctly returns No. Otherwise, all requirements are satisfied, so the label matches the program if and only if the label has nonzero probability (density) with respect to the rest of the program after excluding all `require` statements. The SMT encoding defined above yields a formula which is satisfiable exactly when this is the case. So the algorithm is again correct.

Proposition 2: *Given a fixed object correspondence, the incremental SMT encoding for a* SCENIC *program is equivalent to its monolithic encoding.*

$$
\begin{aligned}
maxX\,(V_1, V_2) &= max\,(E(V_{1,x}), E(V_{2,x}));\\
&\quad maxY\,(V_1, V_2) \text{ defined likewise}\\
minX\,(V_1, V_2) &= min\,(E(V_{1,x}), E(V_{2,x}));\\
&\quad minY\,(V_1, V_2) \text{ defined likewise}\\
rangeX\,(V_1, V_2) &= [minX\,(V_1, V_2), maxX\,(V_1, V_2)]\\
rangeY\,(V_1, V_2) &= [minY\,(V_1, V_2), maxY\,(V_1, V_2)]\\
slope\,(V_1, V_2) &= (E(V_{2,y}) - E(V_{1,y}))/(E(V_{2,x}) - E(V_{1,x}))\\
offset\,(V_1, V_2) &= E(V_{1,y}) - slope\,(E(V_1), E(V_2)) * E(V_{1,x})\\
lineSeg\,(V_1, V_2, x, y) &= \text{point } \langle x, y\rangle \text{ is on the line segment}\\
&\quad (y == slope\,(V_1, V_2) * x + offset\,(V_1, V_2)),\\
&\quad \text{if x} \in rangeX\,(V_1, V_2), \text{y} \in rangeY\,(V_1, V_2)\\
leftLine\,(V_1, V_2, x, y) &= \text{point } \langle x, y\rangle \text{ is to the left of the line}\\
&\quad \text{whose direction is } V_1 \text{ to } V_2\\
&\quad = D_x * T_y - D_y * T_x > 0,\\
&\quad \text{where } D = E(V_2) - E(V_1), T = \langle x, y\rangle - E(V_1)\\
Disc\,(c, r, x, y) &= \text{point } \langle x, y\rangle \text{ is within a disc at } c \text{ of radius } r\\
&\quad = ((x - E(c_x))^2 + (y - E(c_y))^2 \le E(r)^2)\\
Sector\,(c, r, h, a, x, y) &= \text{point } \langle x, y\rangle \text{ is in a sector of } Disc\,(c, r, \cdot, \cdot)\\
&\quad \text{with heading h and angle a}\\
&\quad = Disc\,(c, r, x, y) \wedge rightLine\,(c, V_1, x, y)\\
&\quad \wedge leftLine\,(c, V_2, x, y),\\
&\quad \text{where } V_1 = offsetLocal\,(OP(c, h - a/2), \langle 0, r\rangle),\\
&\quad V_2 = offsetLocal\,(OP(c, h + a/2), \langle 0, r\rangle)\\
visibleRegion\,(X, x, y) &= \begin{cases}
Sector\,([\![\text{X.position}]\!], [\![\text{X.viewDistance}]\!],\\
\qquad [\![\text{X.heading}]\!], [\![\text{X.viewAngle}]\!], x, y),\\
\text{if X is } \texttt{OrientedPoint}\\
Disc\,([\![\text{X.position}]\!], [\![\text{X.viewDistance}]\!], x, y),\\
\text{if X is } \texttt{Point}
\end{cases}\\
tri\,(V_0, V_1, V_2, x, y) &= \text{point } \langle x, y\rangle \text{ is in the triangle defined}\\
&\quad \text{by points } V_0, V_1, \text{ and } V_2\\
&\quad = (\langle x, y\rangle == E(V_1) + (E(V_1) - E(V_2)) * s\\
&\quad + (E(V_2) - E(V_0)) * t),\\
&\quad \text{where } \exists s, \exists t, 0 \le s \le 1, 0 \le t \le 1, s + t \le 1,\\
&\quad \text{using barycentric coordinate system [45]}\\
[\![On]\!](\text{region}, \langle x, y\rangle) &= \bigvee_{i=1}^{n} tri\,(V_0, V_1, V_2, x, y),\\
&\quad \text{for all triangles } (V_0, V_1, V_2) \text{ in a triangulation}\\
&\quad \text{of the region}
\end{aligned}
$$

Figure 19: Encoding of region-containment, where $\langle x, y\rangle$ is the point constrained to lie in the region. Disc and Sector regions (which can have random parameters) use the specialized formulas above; all other regions are fixed and use the generic encoding for `On` at the bottom of the figure.

Proof Sketch: Suppose there are n semantic features $s_1, \ldots, s_n$, in the label and the program. We denote by $\phi_1, \ldots, \phi_n$ the corresponding SMT encodings, respectively. For brevity, assume that these SMT formulas include constraints asserting the equality of the observed semantic features to their values in the label. Then the monolithic encoding of the program is equivalent to $\phi_1 \wedge \cdots \wedge \phi_n$ (recalling that `require` statements are not included in the SMT encoding).

For simplicity, let's first consider a SCENIC program with only dependent semantic features, and no *jointly* dependent features. Suppose our dependency analysis step (refer to Sec. 5.2.1) returns the order $s_1, \ldots, s_n$. The dependency order implies a containment relation: for example, if s_2 is dependent on s_1, this means that s_2's expression tree contains that of s_1. Then our incremental encoding is equivalent to $\phi_1 \wedge \phi_2 | \phi_1 \wedge \phi_3 | (\phi_2, \phi_1) \wedge \cdots \wedge \phi_n | (\phi_{n-1}, \ldots, \phi_1)$ where the vertical bar, |, represents conditioning of the semantic features to the corresponding observed values in the label (as discussed in Sec. 2). The SMT formula $\phi_i | (\phi_{i-1}, \ldots, \phi_1)$ results from substituting the expression trees of $s_1, \ldots, s_{i-1}$ where they occur in the expression tree of s_i with their corresponding observed values in the label, and then encoding s_i using the resulting tree. Now because ϕ_1 implies that s_1 has its observed value, the substitution above means that $\phi_1 \wedge \phi_2 = \phi_1 \wedge \phi_2 | \phi_1$. Applying this rule iteratively, we obtain $\phi_1 \wedge \cdots \wedge \phi_n = \phi_1 \wedge \phi_2 | \phi_1 \wedge \cdots \wedge \phi_n | (\phi_{n-1}, \ldots, \phi_1)$. Therefore, in this case, the incremental SMT encoding of dependent semantic features is equivalent to the monolithic encoding of the program.

This result extends to the case when semantic features are jointly dependent. Suppose there is a single set of $m \le n$ jointly-dependent semantic features: then for some $i \ge 1$ the features $s_i, s_{i+1}, \ldots, s_{i+m-1}$ are jointly dependent. Then the incremental SMT encoding is:

$$
\begin{aligned}
&\phi_1 \wedge \phi_2 | \phi_1 \wedge \cdots \wedge (\phi_i \wedge \phi_{i+1} \wedge \cdots \wedge \phi_{i+m-1}) | (\phi_{i-1}, \ldots, \phi_1)\\
&\wedge \phi_{i+m} | (\phi_{i+m-1}, \ldots, \phi_1) \wedge \cdots \wedge \phi_n | (\phi_{n-1}, \phi_{n-2}, \ldots, \phi_1)\\
&= \phi_1 \wedge \phi_2 \wedge \cdots \wedge \phi_{i-1} \wedge (\phi_i \wedge \phi_{i+1} \wedge \cdots \wedge \phi_{i+m-1}) | (\phi_{i-1}, \ldots, \phi_1)\\
&\wedge \phi_{i+m} | (\phi_{i+m-1}, \ldots, \phi_1) \wedge \cdots \wedge \phi_n | (\phi_{n-1}, \phi_{n-2}, \ldots, \phi_1)\\
&= \phi_1 \wedge \phi_2 \wedge \cdots \wedge \phi_{i-m+1}\\
&\wedge \phi_{i+m} | (\phi_{i+m-1}, \ldots, \phi_1) \wedge \cdots \wedge \phi_n | (\phi_{n-1}, \phi_{n-2}, \ldots, \phi_1)\\
&= \phi_1 \wedge \phi_2 \wedge \cdots \wedge \phi_n
\end{aligned}
$$

by repeatedly applying the rule $\phi_1 \wedge \phi_2 = \phi_1 \wedge \phi_2 | \phi_1$. Finally, this result trivially extends to the case where there is more than one set of jointly-dependent semantic features. Therefore, the incremental SMT encoding is equivalent to the monolithic encoding of the SCENIC program.

Proof Sketch of Theorem 1: Consider an iteration of the main loop, in which the object correspondence is fixed. As we argued in the proof of Proposition 1, for a fixed object correspondence our algorithm correctly rejects labels which violate any `require` statements. Otherwise, by Proposition 2 the incremental SMT encoding is equivalent to the monolithic one, and so by Proposition 1 the SMT queries will all be satisfiable if and only if the label matches the program for the fixed object correspondence. If so, we return Yes; otherwise we continue the main loop with a new object correspondence. Since we try all possible correspondences (except for those which cannot work because at least one of their incremental SMT formulas will be identical to one which was unsat in an earlier iteration), if there is any correspondence under which the label matches we will return Yes, and otherwise we will return No. So the algorithm returns Yes if and only if the label matches the program.