3D Environment Modeling for Falsification and Beyond with Scenic 3.0

Eric Vin, Shun Kashiwa, Matthew Rhea, Daniel J. Fremont, Edward Kim, Tommaso Dreossi, Shromona Ghosh, Xiangyu Yue, Alberto L. Sangiovanni-Vincentelli, and Sanjit A. Seshia

Presented by: Adam Bouafia
University of L'Aquila
Professor: Igor Melatti

UNIVERSITÀ
DEGLI STUDI
DELL'AQUILA

Summary

- **Introduction:** Discusses the challenges of designing CPS, the need for formal models, and how Scenic addresses these challenges. (p. 2)
- **Scenic 3.0 vs. Other Tools:** Compares Scenic to rule/grammar-based tools, probabilistic programming languages, and ML-based scene generation tools, highlighting its unique advantages. (p. 4)
- **Limitations of Scenic 2.0:** Explains the limitations of Scenic 2.0, such as restriction to 2D environments and the use of bounding boxes. (p. 7)
- **Need for 3D in Scenic 3.0:** Justifies the need for 3D modeling, including real-world complexity, perception challenges, and verification needs. (p. 8)
- **Key Innovations in Scenic 3.0:** Covers new features like 3D geometry, mesh shapes and regions, precise visibility, temporal requirements, and a rewritten parser. (p. 10)

Summary (cont.)

Introduction

The Challenge: Designing safe and reliable cyber-physical systems (CPS) like autonomous vehicles is difficult due to the complexity and diversity of real-world environments.
The Need: Formal models are essential to accurately represent these environments for rigorous verification and analysis.
Scenic to the Rescue: Scenic, a probabilistic programming language, offers a readable and precise way to model CPS environments.

Scenic 3.0 vs. Other Tools

- **U**nlike rule/grammar-based tools, Scenic offers more control and enforces requirements over generated data.
- **U**nlike probabilistic programming languages, Scenic provides specialized syntax for geometric scenarios and dynamic behaviors.
- **U**nlike ML-based scene generation, Scenic offers more specificity and control.

Scenic 3.0 vs. Other Tools (cont.)

Scenic vs. Rule/Grammar-Based Tools:

- **Example:** In rule-based tools, generating scenarios for an autonomous vehicle might involve manually defining the rules for every possible interaction at an intersection. For example, a rule might state that "a car must stop if there is a pedestrian crossing".
- **Scenic's Advantage:** Scenic allows for more flexible scenario generation by using probabilistic programming. We can define probabilistic behaviors and constraints rather than exhaustive rules. For instance, specifying that "the probability of a pedestrian crossing the street within 5 seconds is 0.3," which adds variability and realism to the scenarios.

Scenic 3.0 vs. Other Tools (cont.)

Scenic vs. Probabilistic Programming Languages:

- **Example:** Probabilistic programming languages like Stan or Pyro are powerful for statistical modeling but lack specialized syntax for describing geometric and dynamic scenarios crucial for CPS.
- **Scenic's Advantage:** Scenic provides a specialized syntax to define spatial configurations and temporal behaviors directly. For example, you can easily specify that "an object is placed on a surface with a random orientation within specified bounds," which is particularly useful for testing perception algorithms in autonomous vehicles.

Scenic 3.0 vs. Other Tools (cont.)

Scenic vs. Machine Learning-Based Scene Generation:

- **Example:** ML-based tools, such as those using Generative Adversarial Networks (GANs), can generate realistic-looking scenes but lack control over specific elements within the scene.
- **Scenic's Advantage:** Scenic offers precise control over each element of the scene. For instance, in a generated urban environment, you can specifically place a stop sign at a particular distance from an intersection and ensure it is visible to the vehicle's sensors under certain conditions, which is critical for testing specific perception capabilities.

Limitations of Scenic 2.0

Scenic 2.0 Limitations:

- Restricted to 2D environments, limiting its applicability to certain domains (e.g., aerial vehicles, underwater robots).
- Used bounding boxes for object representation, leading to inaccurate collision detection and visibility checks.

Limitations of Scenic 2.0 (cont.)

Scenic 2.0 Limitations:

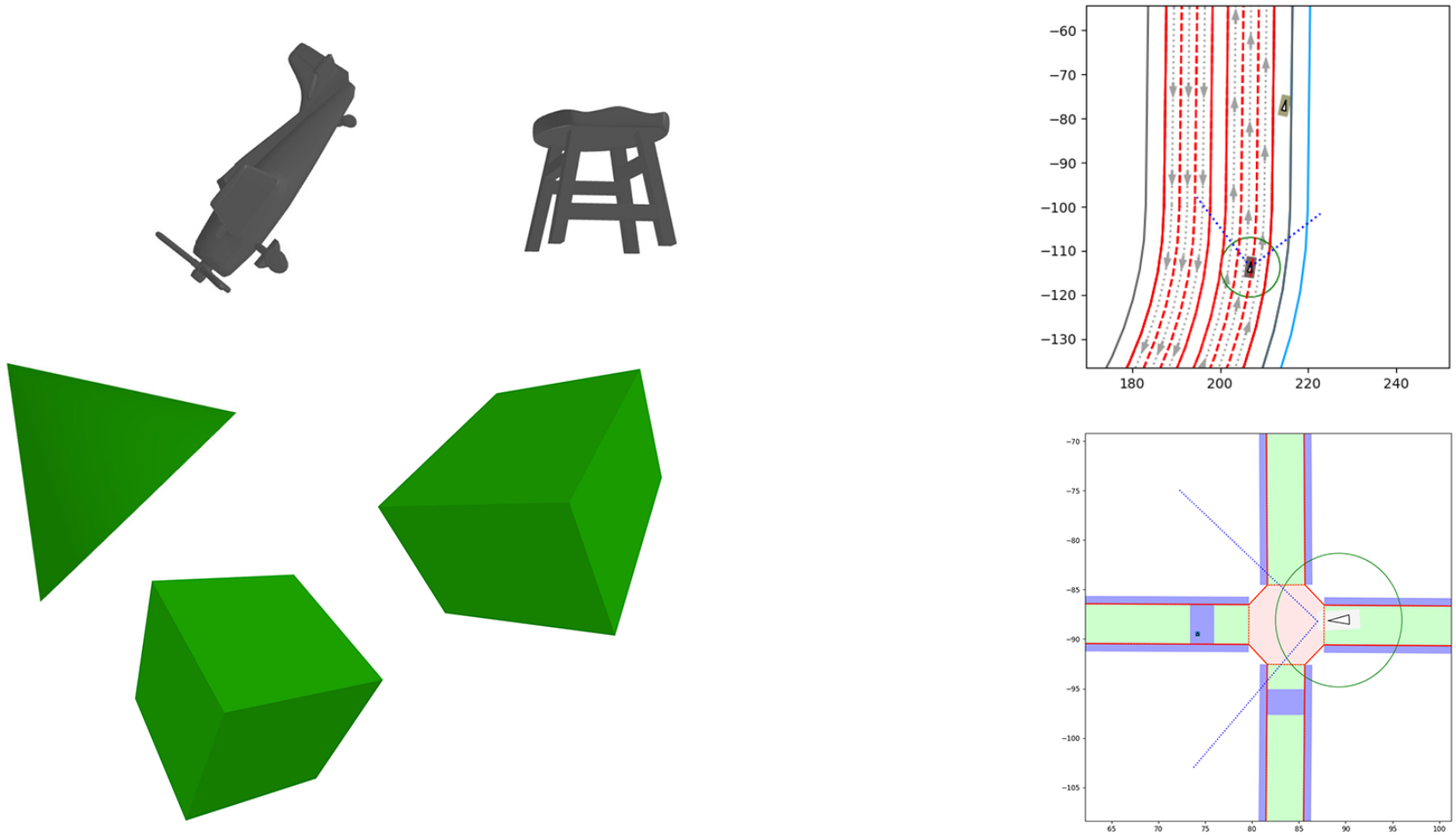


Figure 1: 2D vs 3D environments

The Need for 3D in Scenic 3.0

Why 3D Matters:

- **Real-World Complexity:** Many CPS operate in 3D spaces.
- **Perception Challenges:** Accurate perception requires understanding occlusion and complex object shapes.
- **Verification Needs:** Formal verification demands precise modeling of 3D geometry and physics.

Scenic 3.0: Key Innovations - 3D Geometry (cont.)

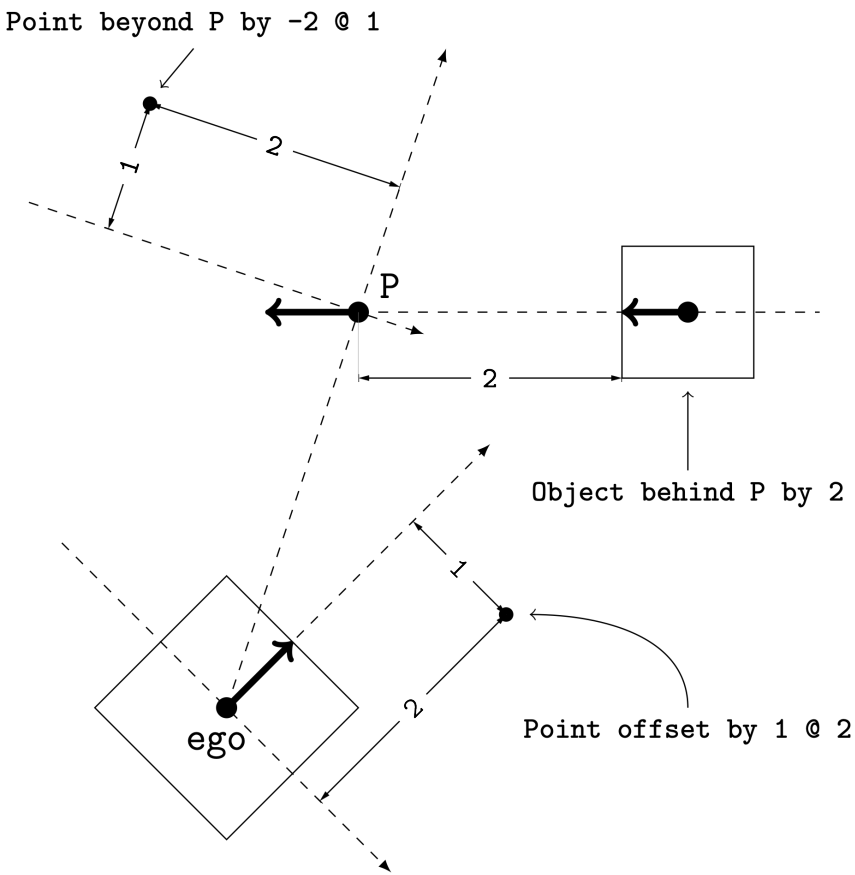


Figure 2: Object and Point Placement in Scenic 3.0

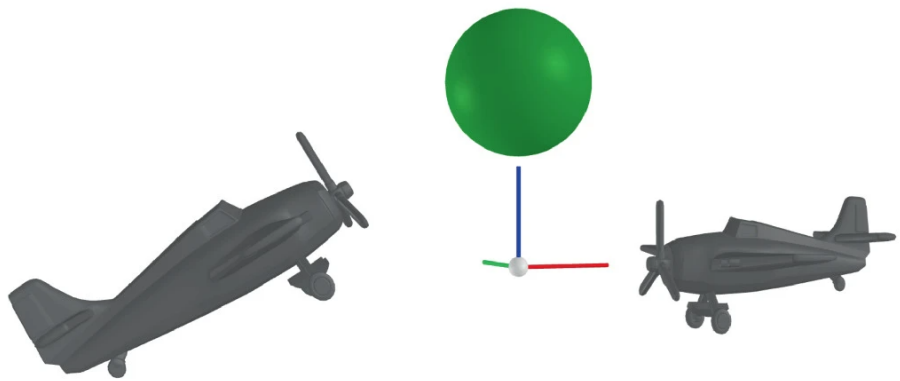
Scenic 3.0: Key Innovations - 3D Geometry (cont.)

- **Code Example of Line-of-sight-based orientations:**

Fig. 1.

From: 3D Environment Modeling for Falsification and Beyond with Scenic 3.0

```
1  ego = new Ball at (0,0, 1.25)
2  new Plane at (2,0,0), facing toward ego
3  new Plane at (-2,0,0), facing directly toward ego
```



Line-of-sight-based orientations in Scenic. The ego ball (highlighted green) is placed above the origin, as seen by the RGB global coordinate axes, with one plane facing towards the ego and another facing directly toward the ego. (Color figure online)

Figure 3: Line-of-sight-based orientations in Scenic. The ego ball is placed above the origin with one plane facing towards the ego and another facing directly toward the ego.

Scenic 3.0: Key Innovations - 3D Geometry (cont.)

Explanation of Line-of-sight-based orientations Example Code:

- **ego = new Ball at (0,0, 1.25):** This line creates a new object named ego, which is a ball placed at coordinates (0,0,1.25) in the 3D space.
- **new Plane at (2,0,0), facing toward ego:** This line creates a new plane at coordinates (2,0,0). The plane is oriented such that it faces towards the ego ball.
- **new Plane at (-2,0,0), facing directly toward ego:**This line creates another plane at coordinates (-2,0,0). This plane is oriented to face directly toward the ego ball, meaning it adjusts its pitch and yaw to align its front towards the ball.

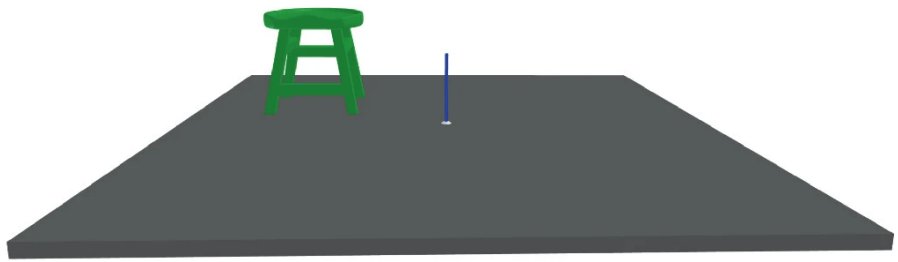
Scenic 3.0: Key Innovations - 3D Geometry (cont.)

- **Code Example of placing a chair on a floor:**

Fig. 2.

From: 3D Environment Modeling for Falsification and Beyond with Scenic 3.0

```
1  floor = Object with width 5, with length 5, with height 0.1
2  ego = new Chair on floor
```



A Scenic program placing a chair on a floor. The Z-axis of the global coordinate axes protrudes from the floor, indicating which direction is up.

Figure 4: A Scenic program placing a chair on a floor. The Z-axis of the global coordinate axes protrudes from the floor, indicating which direction is up.

Scenic 3.0: Key Innovations - 3D Geometry (cont.)

Explanation of placing a chair on a floor Example Code:

- **floor = Object with width 5, with length 5, with height 0.1:** This line creates a new object representing the floor, with specified dimensions (width, length, height).
- **ego = new Chair on floor:** This line creates a new chair object named ego, placing it on top of the floor object.

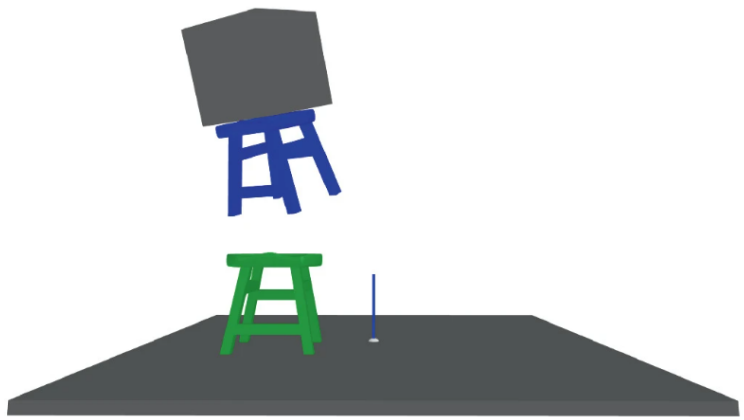
More on 3D Geometry

Handling Complex Placements: Specifiers for placing objects on surfaces, with options for adjusting orientation and position.

Fig. 3.

From: 3D Environment Modeling for Falsification and Beyond with Scenic 3.0

```
1  floor = new Object with width 5, with length 5, with height 0.1
2  air_cube = new Object at (Range(-5,5), Range(-5,5), 3),
3          facing (Range(0,360 deg), Range(0,30 deg), 0)
4  new Chair below air_cube, with color (0,0,200)     # blue chair
5  ego = new Chair below air_cube, on floor     # green chair
```



A Scenic program placing a green chair on the floor under a rotated cube in midair. A blue chair is placed directly under the cube for clarity. (Color figure online)

Figure 5: A Scenic program placing a green chair on the floor under a rotated cube in midair. A blue chair is placed directly under the cube for clarity.

More on 3D Geometry (cont.)

Explanation of Handling Complex Placement Example Code:

- **floor = new Object with width 5, with length 5, with height 0.1:** Creates a floor object with specified dimensions.
- **air_cube = new Object at (Range(-5,5), Range(-5,5), 3), facing (Range(0,360 deg), Range(0,30 deg), 0):** Creates a cube object placed randomly within a range of coordinates in the XY plane at a fixed height of 3 units. The cube faces a random direction within the specified ranges of yaw and pitch angles.
- **new Chair below air_cube, with color (0,0,200):** Creates a blue chair placed directly below the air_cube.
- **ego = new Chair below air_cube, on floor:** Creates a green chair placed directly below the air_cube and on the floor.

Key Innovations - Mesh Shapes and Regions

Precise 3D Shapes: Objects are represented using detailed 3D meshes, not just bounding boxes.

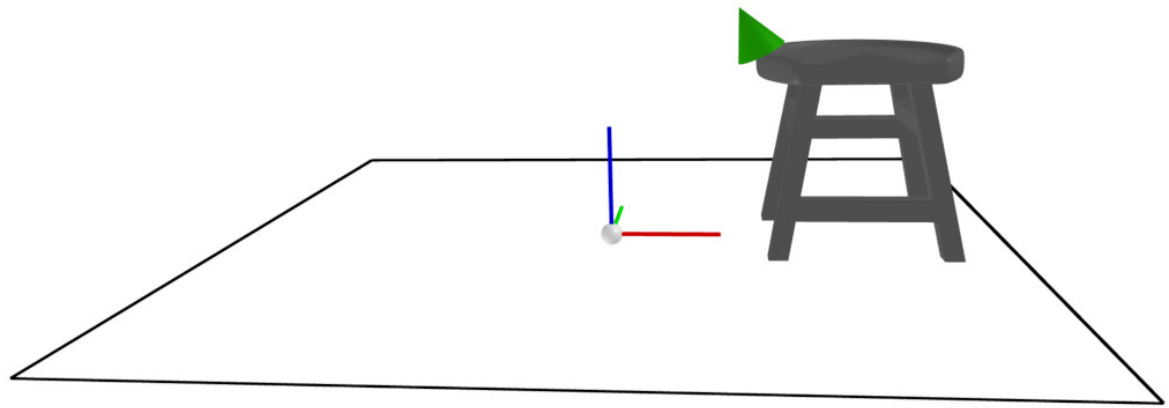


Figure 6: Example of Precise 3D Shapes.

Key Innovations - Precise Visibility

Accurate Collision Detection: Mesh-based collision checks ensure realistic object interactions.
Mesh Regions: Define surfaces and volumes for precise object placement and sampling as it uses the Trimesh library to handle the 3D meshes which enable accurate collisions and containment checks.
Ray Tracing: Visibility checks use ray tracing to account for occlusion and complex shapes.

Key Innovations - Temporal Requirements

Realistic Sensor Modeling: Supports various view cones (e.g., cameras, LiDAR) for accurate sensor simulation.

Performance Optimizations: Heuristics for efficient ray tracing in simple cases.

Temporal Requirements: Express complex temporal constraints using Linear Temporal Logic (LTL).

Key Innovations - Rewritten Parser and Visualization

Rewritten Parser: Improved parser based on a formal grammar for better error handling and extensibility as it is based on a Parsing Expression Grammar (PEG) which improves error handling and allows for easier extension of the language.

- **Code example:**

```
require (carA not in intersection and carB not in intersection
    until carC in intersection)
```

Figure 7: Scenic 3 extends to arbitrary properties in Linear Temporal Logic, allowing natural properties like this to be concisely expressed:

Case Study 1: Robot Vacuum Falsification

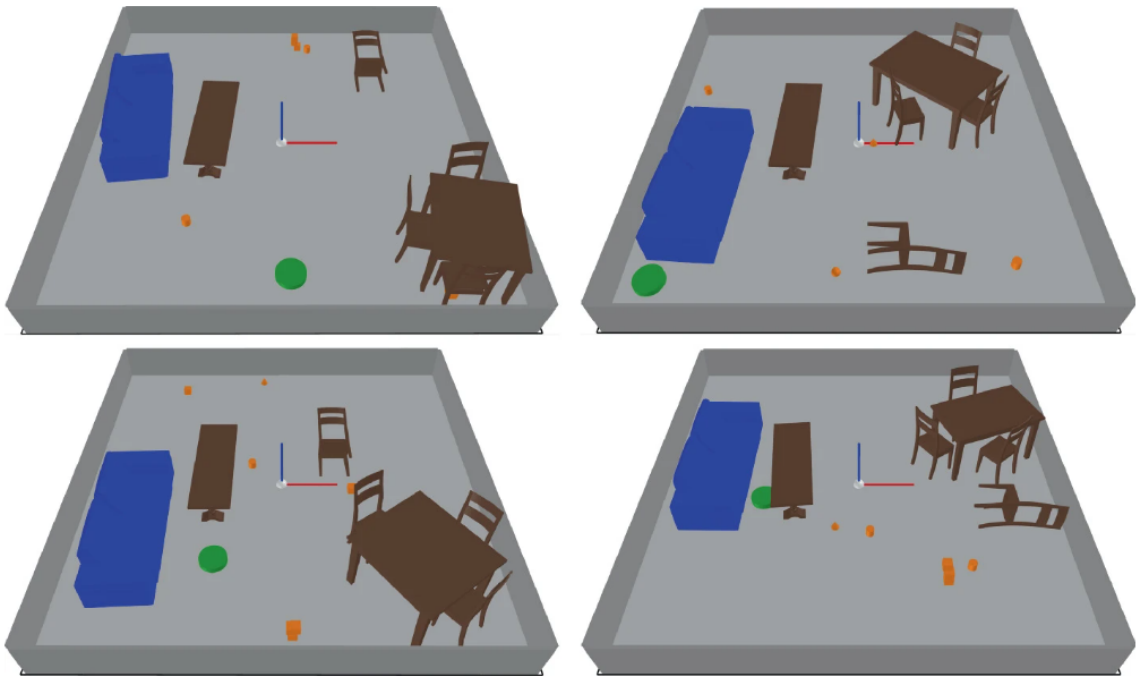
Goal: Evaluate the performance of a robot vacuum in cluttered environments.
Scenic 3.0's Role: Generate diverse 3D room layouts with varying object placements.
Specification: The robot must clean at least a third of the room within 5 minutes.

Case Study 1: Results and Analysis

Results: Revealed limitations in the vacuum's navigation and cleaning capabilities, particularly in the presence of many obstacles.

Fig. 4.

From: 3D Environment Modeling for Falsification and Beyond with Scenic 3.0



Several sampled scenes from the robot vacuum scenario.

Figure 8: Several sampled scenes from the robot vacuum scenario.

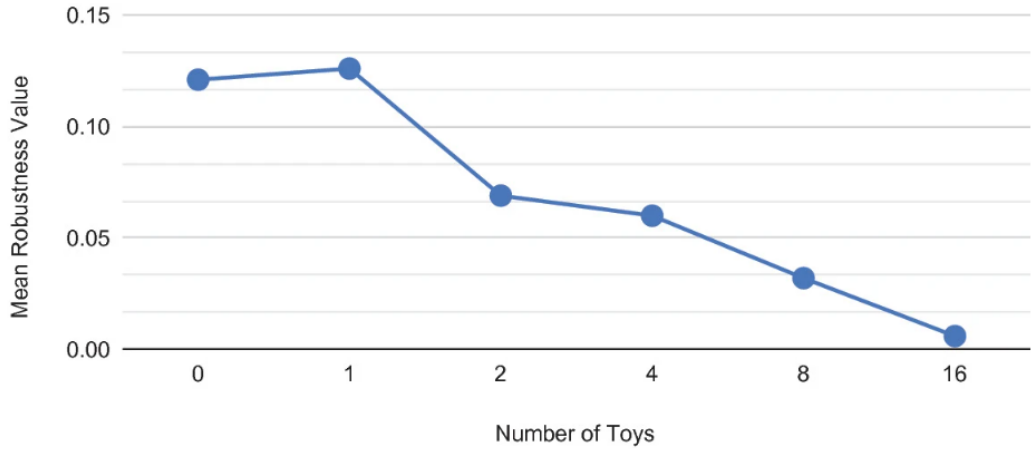
Case Study 1: Results and Analysis (cont.)

Results: We tested the default controller for the vacuum against 0, 1, 2, 4, 8, and 16-toy variants of our Scenic scenario, running 25 simulations for each variant.
For each simulation, we computed the robustness value of our spec .
The average values are plotted in Fig. 5, showing a clear decline as the number of toys increases. Many of the runs actually falsified : up to 44% with 16 toys.

Case Study 1: Results and Analysis (cont.)

Fig. 5.

From: 3D Environment Modeling for Falsification and Beyond with Scenic 3.0



Spec. robustness value vs. number of toys, averaged over 25 simulations.

Figure 9: Spec. robustness value vs. number of toys, averaged over 25 simulations.

Case Study 1: Why Scenic 3.0 Matters

Why Scenic 3.0 Matters:

- **3D Modeling:** Accurately represents the vacuum's interactions with furniture and toys.
- **Complex Scenarios:** Generates challenging scenarios with objects placed on top of each other or in tight spaces.
- **Diverse Layouts:** Samples from a wide range of room configurations for comprehensive testing.

Case Study 2: Constrained Data Generation for an Autonomous Vehicle

Goal: Test an autonomous vehicle's perception system in an urban intersection with occluding buildings.
Scenic 3.0's Role: Create realistic 3D models of the intersection, including buildings, parked cars, and pedestrians.

Case Study 2: Results and Analysis

Specification: The crossing car should not be visible until it is close to the autonomous vehicle.

Results: Produced a dataset of realistic and challenging scenarios for perception system evaluation, highlighting the importance of accurate occlusion modeling.

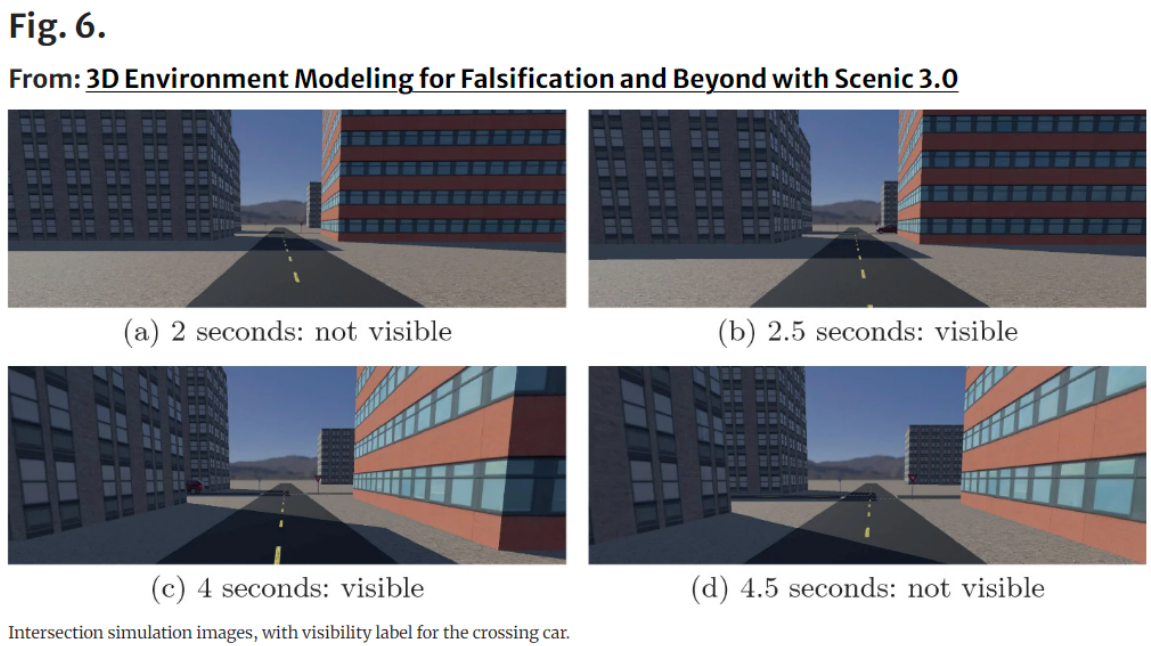


Figure 10: Intersection simulation images, with visibility label for the crossing car.

Case Study 2: Why Scenic 3.0 Matters

Why Scenic 3.0 Matters:

- **Precise Visibility:** Accurately models occlusion caused by buildings and other objects.
- **Temporal Requirements:** Ensures scenarios meet specific criteria (e.g., car not visible until a certain distance).
- **Diverse Scenarios:** Generates a wide range of challenging situations for robust perception system testing.

Scenic 3.0's Impact on Formal Methods

Extends Scope: Enables formal verification and analysis in new domains (e.g., aerial vehicles, underwater robots).
Realistic Scenarios: Generates more realistic and challenging scenarios for testing and validation.
Improved Accuracy: Precise 3D modeling and visibility checks lead to more accurate results.

Broader Applications of Scenic 3.0

Training Data Generation: Create large, diverse datasets for machine learning models.
Scenario-Based Testing: Design and execute comprehensive test suites for CPS.
Simulation and Visualization: Visualize and explore complex 3D environments.

Conclusion

Key Takeaway: Scenic 3.0 is a powerful tool for modeling, analyzing, and verifying CPS in complex 3D environments.
Contributions:

- Native 3D syntax and specifiers
- Precise mesh-based shapes and regions
- Ray tracing-based visibility
- Temporal requirements with LTL
- Rewritten parser

Future Directions

3D Scenario Optimization: Develop algorithms to automatically optimize 3D scenarios for specific verification goals.

Drone Applications: Explore the use of Scenic 3.0 for modeling and verifying drone behaviors in complex environments.

Custom Specifiers and Pruning: Allow users to define their own specifiers and pruning techniques for greater flexibility.