On Determinism of Game Engines used for Simulation-based Autonomous Vehicle Verification

Abanoub Ghobrial[1,3], Greg Chance[1,3], Kevin McAreavey[1,3], Severin Lemaignan[2,3], Tony Pipe[2,3], Kerstin Eder[1,2]

[1] University of Bristol, Bristol, UK
[2] University of the West of England, Bristol, UK
[3] Bristol Robotics Laboratory, Bristol, UK

***Abstract*–Game engines are increasingly used as simulation platforms by the autonomous vehicle (AV) community to develop vehicle control systems and test environments. A key requirement for simulation-based development and verification is determinism, since a deterministic process will always produce the same output given the same initial conditions and event history. Thus, tests are rendered repeatable and yield simulation results that are trustworthy and straightforward to debug. However, game engines are seldom deterministic.**

This paper first reviews and identifies the potential causes of non-deterministic behaviours in game engines. This is then followed by a case study using CARLA, an open-source autonomous driving simulation environment powered by Unreal Engine, which highlights its inherent shortcomings in providing sufficient precision in experimental results. Different configurations and utilisations of the software and hardware are explored to determine an operational domain where the simulation precision is satisfactory low i.e. variance between repeats becomes negligible for development and testing work.

Finally, a method of a general nature is proposed, that can be used to find the domains of permissible variance in game engines simulations for any given system configuration.

1 Introduction

Simulation-based verification of autonomous driving functionality is a promising counterpart to costly on-road testing, that benefits from complete control over (virtual) actors and their environment. Simulated tests aim to provide evidence to developers and regulators of the functional safety of the vehicle or its compliance with commonly agreed upon road conduct [1], national rules [2] and road traffic laws [3] which form a body of safe and legal driving rules, termed assertions, that must not be violated.

Design confidence is gained when the autonomous vehicle (AV) can be shown to comply with these rules, e.g. through assertion checking during simulation. There have been a number of fatalities with AVs, some of which could be attributed to insufficient verification and validation (V&V), e.g. [4]. Simulation environments offer a means to explore the vast parameter space in a safe and efficient manner [5] without the need for millions of miles of costly on-road testing [6]. In particular, simulations can be biased to increase the frequency at which otherwise rare events occur [7]; this includes testing how the AV reacts to unexpected behaviour of the environment [8].

Increasingly, the autonomous vehicle community is adopting game engines as simulation platforms to support the development and testing of vehicle control software. CARLA [9], for instance, is an open-source simulator for autonomous driving that is implemented in the Unreal Engine [10] a real-time 3D creation environment for the gaming and film industry as well as other creative sectors [11].

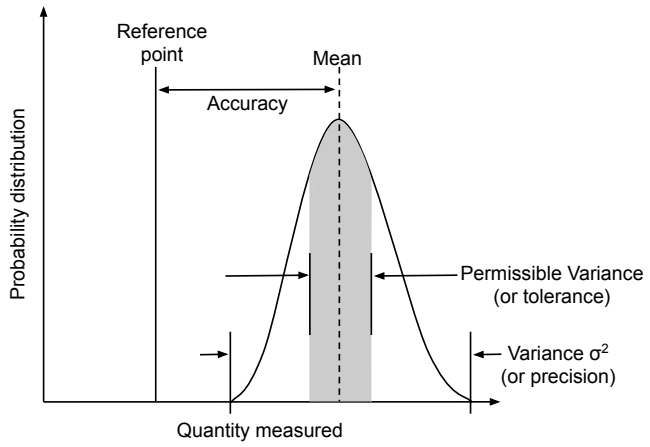
State-of-the-art game engines provide a convenient option for simulation-based testing. They offer sufficient realism [7] in the physical domain combined with realistic rendering of scenes, potentially suitable for perception stack testing and visual inspection of accidents or near misses. Furthermore, they are easy to setup and run with respect to on-road testing and are simple to control and observe, both with respect to the environment the AV operates in as well as the temporal development of actors [12]. Finally, support for hardware-in-the-loop development or a real-time test-bed for cyber-security testing [13] may also be required. Compared to the vehicle dynamics simulators and traffic-level simulators used by manufacturers [14], game engines offer a simulation solution that meets many of the requirements for the development and functional safety testing of AVs in simulation. However, while game engines are designed primarily for performance to achieve a good user experience, the requirements for AV verification go beyond that and include determinism.

1.1 Definitions

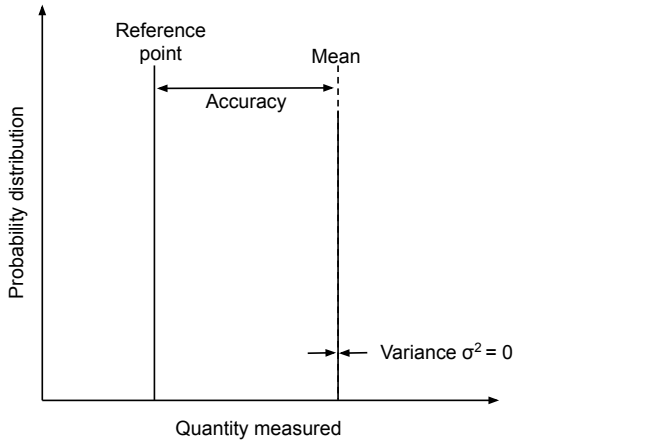
A list of definitions are given here which are used in the subsequent discussion. Refer to Fig. 1 throughout this section.

1.1.1 Determinism

Schumann describes determinism as the property of causality given a temporal development of events such that any state is completely determined by prior states [15]. However, in the context of simulation this should be expanded to include not just prior states but also the history of actions taken by all actors. Therefore, a deterministic simulation will always produce the same result given the same history of prior states and actions.



(a) Non-deterministic



(b) Deterministic

Fig. 1: Demonstration of variance, precision, tolerance and determinism

A simulation can be thought of as the generation or production of experimental data. In the case of a driving simulator, kinematics will describe future states of actors given the current conditions and actions taken, thereby generating new data. If a simulation is deterministic then there will be no variation in the generated output data, i.e. all future states are perfectly reproducible from prior states and actions. However, if a simulation is non-deterministic then there will be a variation in the generated output data.

1.1.2 Variance, Precision & Tolerance

We adopt some terms from the mechanical engineering and statistics domain that best describe when there is variation in the generated output data [16]. *Variance* is used here to define the spread, or distribution, of the generated output data with respect to the mean value. *Precision* is synonymous with *variance* although inversely related mathematically. Therefore, variance can indicate the degree to which a simulation can repeatedly generate the same result when executed with the same conditions and actions taken *Tolerance* is defined as the permissible limit of the variance, or in short the *permissible variance*.

As an analogy, the simulator can be thought of as a manufacturing process, producing data and to find the value of the precision then the output must be measured for differences when the process is repeated. Those differences describe the spread or variance in the process output and a hard limit on the variance can be defined, Fig. 1a, beyond which the output fails to meet the required tolerance, e.g. rejected for quality control. Real manufacturing fails to achieve absolute precision and hence the need for tolerances on design specifications to account for the variance in real-world processes.

If a simulator is deterministic then it will produce results with absolute precision or zero variance, Fig. 1b, and hence will be within acceptable tolerance. If the simulator is non-deterministic then there will be a variance in the output which can be measured.

1.1.3 Accuracy

Precision and tolerance should not to be confused with *accuracy* which describes how closely the mean of the generated output of a process aligns to a known standard or reference value. We therefore define accuracy as the difference between the true value, or reference value, and what has been achieved in the simulation or generation process. For a driving simulation the reference value may be the real world that the simulation seeks to emulate, where any divergence from this standard is termed the *reality gap*. In most cases such accuracy will not be possible due to computational demands of such an exact replica and in most cases is unnecessary, where some authors state that 'just the right amount of realism' is required to achieve valid simulation results [7].

1.1.4 Simulation Trace

A simulation trace is the output log from the simulator consisting of a time series of all actor positions (x, y, z) at regular time intervals. This definition could be extended to include other variables. A set of simulation traces derived from the same input then forms the experimental data on which variance is calculated.

1.1.5 Simulation Variance & Deviation

If the simulator is non-deterministic then how can this variance be measured? This can be achieved by monitoring any simulated output variables that should be consistent from run to run.

Actor path variance, derived from the simulation trace, is chosen over other variables as this distance based metric forms the basis for many downstream verification tests. Therefore the term *simulation variance* refers to a measure of actor position variance in the simulation with respect to time assuming fixed actions. *Deviation* (SI unit m) is the square root of variance (SI unit m^2), which is a more intuitive value to comprehend when considering interpretation of verification tests.

1.1.6 Scene, Scenario & Situation

We will adopt the terminology defined in [12], where *scene* refers to all static objects including the road network, street furniture, environment conditions and a snapshot of any dynamic elements. Dynamic elements are the elements in a scene whose actions or behaviour may change over time; these are considered actors and may include the AV or *ego vehicle*, other road vehicles, cyclists, pedestrians and traffic signals. The *scenario* is then defined as a temporal development between several scenes which may be specified by specific parameters. A *situation* is defined as the subjective conditions and determinants for behaviour at a particular point in time.

1.2 When is Determinism needed?

Determinism is a key prerequisite for simulation during AV development and testing. A deterministic simulation environment guarantees that tests are repeatable. A deterministic simulator can be considered to have zero *variance*. If the variance is non-zero it can no longer be considered deterministic but this may be sufficient for certain applications as long as it is *within tolerance*. Therefore, *tolerance* is the acceptable degree of variability between repeated simulation traces. When the simulation output is within tolerance, coverage results are stable and, when a test fails, debugging can rely on the test producing the same trace and outcome when repeated. This ensures that software bugs can be found and fixed efficiently, and that simulation results are trustworthy.

If the simulation is non-deterministic and has a non-permissible variance in, for example, actor positions, this may lead to assessment errors, making it difficult to understand and remove bugs. In the worst case, bugs that could have been identified in simulation remain undetected, resulting in false confidence in the safety of the AV's control software.

When used for gaming, game engines do not need to be deterministic nor do they even have any requirements on the limits of permissible variance; there are no safety implications from non-determinism in this domain, nor is finding and fixing all the bugs related to non-determinism a high priority for games developers. It could even be argued that simulation variance is a feature that enhances gaming and improves the user experience. However, the situation is very different for AV development and testing. Thus, our main research question is: How can one assess the extent to which a simulation environment is deterministic or has a permissible level of variance?

In this paper we investigated how the non-determinism of Carla, based on the Unreal game engine, when used for simulation-based AV verification affects the simulation results. In our case study, scenarios between pedestrian and vehicle actors have been analysed to identify conditions that result in non-deterministic simulation output by analysis of actor position variance. By analysing actor position variance we find that the Carla simulator is non-deterministic under certain conditions. Carla exhibits a permissible level of variance when system utilisation is restricted to 75% or less and ensuring termination of scenarios once a vehicle collision has been detected.

The insights gained from this case study motivated the development of a general step-by-step method for AV developers and verification engineers to determine the simulation variance for a given simulation environment. Knowing the simulation variance will help assess the impact that using a game engine for AV simulation may have on verification tasks. In particular, this can give a better understanding of the effects of non-determinism and to what extent simulation precision may impact on verification results.

This paper is structured as follows. Section 2 briefly introduces how game engines work before investigating in Section 3 the potential sources of non-determinism in game engines. Our case study of simulation variance for a number of scenarios involving pedestrian and vehicle settings using CARLA is presented in Section 4. Section 5 presents the step-by-step method to assess the suitability of a simulation system for AV verification in general. We conclude in Section 6 and give an outlook on future work.

2 Background

There are numerous game engines with their associated development environments which could be considered suitable for AV development, e.g. Unreal Engine [10], Unity [17], CryEngine [18]. Specific autonomous driving research tools have been created to abstract and simplify the development environment, some of which are based on existing game engines, e.g. CARLA [9], AirSim [19], Apollo [20], and some have been developed for cloud based simulation, e.g. Nvidia Drive Constellation [21].

Investigating the determinism of game engines has not attracted much research interest since performance is more critical for game developers than accurate and repeatable execution. Ensuring software operates deterministically is a non-trivial task and catching intermittent failures, or flaky tests [22], in a test suite that cannot be replayed makes the debugging process equally difficult [23]. This section gives an overview of the internal structure of a game engine and what sources or settings in the engine may affect *simulation variance*.

Central to a game engine are the main game logic, the artificial intelligence (AI) component, the audio engine, and the physics and rendering engines. For AV simulation, we focus on the latter two. The game loop is responsible for the interaction between the physics and rendering engines. Fig. 2 depicts a simplified representation of the process flow in a game engine loop, where initialisation, game logic and decommissioning have been removed [24]. A game loop is broken up into three distinct phases: processing the inputs, updating the game world (Physics Engine), and generating outputs (Rendering) [25].

The game loop cycle starts with initialising the scene and actors. Input events from the User or AI are then processed followed by a physics cycle which may repeat more than once per rendered frame if the physics time step, dt, is less than the render update rate. This is illustrated by the loop in the physics update in Fig. 2. The render

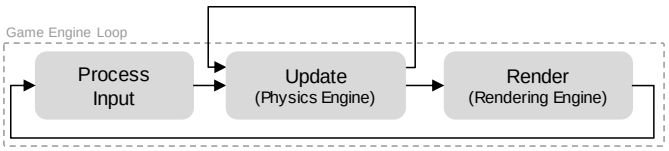


Fig. 2: Game engine loop block diagram [26].

update will process frames as fast as the computational processing will allow up to the maximum monitor refresh rate [27]. When the frame is rendered the game loop cycle returns to processing inputs. An intuitive and more detailed description of the interplay between the physics and render cycles is given in [28].

The physics engine operates according to a time step, *dt*. The shorter this time step is, the smoother the interpretation of the physical dynamics will be. To use a fixed physics time step, the user's display refresh rate needs to be known in advance. This requires an update loop to take less than one render tick (one frame of real world time). Given the range of different hardware capabilities, a variable delta time is often implemented for game playing, taking the previous frame time as the next *dt*. However, variable *dt* can lead to different outcomes in repeated tests and in some cases unrealistic physical representations [29]. Semi-fixed or limited frame rates ensure *dt* does not exceed some user-defined limit to meet a minimum standard of physical representation but allow computational headroom for slower hardware. Some engines provide sub-stepping which processes multiple physics calculations per frame at a greater CPU cost, e.g. Unreal Engine [30]. If the engine tries to render between physics updates, *residual lag* can occur, which may result in frames arriving with a delay to the simulated physics. Thus, extrapolation between frames may need to be performed to smooth transition between scenes. Note that both residual lag and extrapolation could affect perception stack testing. In exceptional cases where computational resources are scarce, the fixed time step can be greater than the time between render ticks and the simulation will exhibit lag between input commands and rendered states, resulting in unsynchronised and unrealistic behaviour as can be experienced when games are executed on platforms not intended for gaming. Considering the objectives for gaming and comparing them to these for AV development and testing, there are fundamental differences. Providing game players with a responsive real-time experience is often achieved at the cost of simulation accuracy and precision. The gamer neither wishes a faithful representation of reality, i.e. the gamer will accept low *accuracy*, nor do they require for repeated actions to result in the same outcome to within a particular tolerance, i.e. can be low *precision*. In contrast, the precision required for AV development and testing is much higher, especially for perception stack testing, but also for visual inspection of accidents or near misses. If necessary, it is acceptable to achieve the required level of tolerance on the precision at the cost of real-time performance. For example, for perception stack testing the sensors need to get input that is repeatable so that if any software bugs are found they can be re-played and the issue resolved. This may only be realisable by slowing down the rendering to enable more extensive physics calculations.

3 Potential Sources of Non-Determinism

The following review discusses the potential sources of non-determinism that were found in the literature or found as part of our investigation into game engines. We have examined hardware- as well as software-borne sources of non-determinism that occur at different layers of abstraction. A good analysis of potential sources is given by Strandberg et al. [22] although the AV simulation domain introduces it's own unique challenges that were not considered in that paper.

3.1 Floating-Point Arithmetic

Floating-point representation of real numbers is limited by the fixed bit width available in the hardware, resulting in the finite precision with which numbers can be represented in a computation. Thus, the results of arithmetic operations must fit into the given bit width, which is achieved through rounding to the nearest representable value. This gives rise to rounding errors [31, 32]. Even operations such as the average of an array can result in issues with overflows, underflows and non-associative addition [33] which may result in non-deterministic behaviour. Calculation results may differ due to the nature of non-associative floating point arithmetic if there is a change in compiler, a change in the execution order or even if the execution is performed on a GPU rather than a CPU which may have different register widths [34]. Some authors even suggest avoiding the use of floating point entirely for assertion testing due to imprecision and non-deterministic behaviour [35].

In the context of AV simulation, such rounding errors could result in accuracy issues of, for example, actor positions within the environment, leading to falsely satisfied or failed assertions. Thus, one could jump to the conclusion that floating-point errors cause non-deterministic behaviour, resulting in loss of repeatability.

However, these precision issues are better described as *incorrectness*, rather than them being a source of non-determinism. It is normally reasonable to assume that the processors on which game engines run are designed to be deterministic with respect to floating-point arithmetic. Thus, when a rounding error occurs, then the same processor should produce the same *incorrect* result given the same operands. So, even if the result of a floating-point operation is incorrect due to floating-point rounding errors, it should always be equally *incorrect* as long as the implementation conforms to the IEEE floating-point standard [36].

Beyond hardware limitations, aggressive optimisations by the compiler can also introduce incorrectness in floating-point arithmetic [37]. However, the same executable should still return the same output for identical input. Therefore, it can be concluded that floating-point

arithmetic does not contribute to *simulation variance* for repeated tests using the same executable on the same hardware with the same configuration.

3.2 Scheduling, Concurrency and Parallelisation

Runtime scheduling is a resource management method for sharing computational resources between tasks of different or equal priority where tasks are executed dependent on the scheduler policy of the operating system. A scheduler policy may be optimised in many ways such as for task throughput, deadline delivery or minimum latency [38]. Changing the scheduling policy and thread priorities may increase simulation variance.

It is, however, unlikely that changes to thread priorities or scheduling policy would occur during repeated controlled tests for the same hardware and operating system configuration. Given that hardware is considered deterministic and if the thread execution order does not change between tests, then for the same hardware and operating system configuration the same output should be given.

However, if some aspects of the game loop are multi-threaded [39], then, even with a clear thread scheduling order, any background process may interrupt the otherwise deterministic sequence of events. This may, for example, alter the number of physics calculations that can be performed within the game loop and hence result in simulation variance. Using multiple threads has been found to affect initialisation ordering for training machine learning models which can lead to unpredictable ordering of training data and non-deterministic behaviour [40, 41]. Interference may also occur when a scheduler simply randomly selects from a set of threads with equal priority, resulting in variation of the thread execution order.

Similar to thread scheduling, scheduling at the hardware level on a multi-core system determines on which processor core to execute processes. This may be decided based on factors such as throughput, latency or CPU utilisation. If due to the CPU utilisation policy the same single-threaded script executes multiple times across different physical cores of the same CPU type, then execution should still produce the same output. This is because the processor cores are identical and any impurities across the bulk silicon and minor perturbations in the semiconductor processing across the chip that may exist should have been accommodated for in design and manufacturing tolerances. However, scheduling multiple processes across several processing cores, where the number of cores is smaller than the number of processes, can result in variation of the execution order and cause simulation variance unless explicitly constrained or statically allocated prior to execution.

Indeed, the developers of the de-bugging program `RR` [42] took significant steps to ensure deterministic behaviour of their program by executing or context-switching all processes to a single core, which avoids data races as single threads cannot concurrently access shared memory. This allowed control over the scheduling and execution order of threads promoting deterministic behaviour [23].

Likewise, simulation variance may be observed for game engines that use GPU parallelisation to improve performance by offloading time-critical calculations to several dedicated computing resources simultaneously. While this would be faster than a serial execution, the order of execution arising from program-level concurrency is often not guaranteed.

Overall, scheduling, concurrency and parallelisation may be reasons for *simulation variance*.

3.3 NUMA Non-Uniform Memory Access

For a repeated test that operates over a number of cores based on a CPU scheduling policy, memory access time may vary depending on the physical memory location relative to the processor. Typically a core can access its own memory with lower latency than that of another core resulting in lower interprocessor data transfer cost [43]. Changes in latency between repeated tests may, in the worst case, cause the game engine to operate non-deterministically if tasks are processed out of sequence using equal priority scheduling, or, perhaps, simply with an increased data transfer cost, i.e. slower. By binding a process to a specific core for the duration of its execution, the variations in data transfer time can be minimised.

3.4 Error Correcting Code (ECC) Memory

ECC Memory is used ubiquitously in commercial simulation facilities and servers to detect and correct single bit errors in DRAM memory [44]. Single bit errors may occur due to malfunctioning hardware, ionising radiation (background cosmic or environmental sources) or from electromagnetic radiation [45]. If single bit errors go uncorrected then subsequent computational processing will produce incorrect results, potentially giving rise to non-determinism due to the probabilistic nature of such errors occurring. Estimating the rate of error is difficult and dependent on hardware, environment and computer cycles [46].

Any simulation hardware not using ECC memory that runs for 1000's of hours, typical in AV verification, is likely to incur significant CPU hours and is therefore subject to increased exposure to these errors. To counter this, commercial HPC and simulation facilities typically employ ECC memory as standard.

3.5 Game Engine Setup

The type and version of the engine code executed should be considered, paying attention to the control of pseudo-random numbers, fixed physics calculation steps (dt), fixed actor navigation mesh, deterministic ego vehicle controllers and engine texture loading rates especially for perception stack testing. For example, in Unreal Editor the *unit*[47] command can be used to monitor performance metrics such as *Frame* which reports the total time spent generating one frame, *Game* for game loop execution time and *Draw* for render thread time. With

respect to perception stack testing, weather and lighting conditions in the game engine should be controlled as well as any other dynamic elements to the simulation environment, e.g. reflections from surface water, ensuring textures are not randomly generated.

3.6 Actor Navigation

There is existing evidence to suggest actor navigation, typically pedestrians, could be the cause of non-deterministic simulation behaviour [48]. Unreal Engine is the underlying framework for the CARLA simulator which uses the A* algorithm for actor navigation [49]. The A* algorithm [50] will give deterministic outcomes as long as the environment is deterministic [51, 52].

A navigation mesh is a fixed area of the environment where actors are free to navigate. Navigation meshes are used by the path planning algorithm to find the shortest distance between navigable points in the environment and could be considered as a potential source of non-determinism if the navigation mesh is not a fixed entity. However, the environment management in CARLA implements fixed binary files for navigation meshes that are linked to each road scene or driving environment and cannot be unknowingly modified and therefore can be considered fixed.

Unless other sources that can alter the fixed environment exist, the A* algorithm should give deterministic results. Thus, actor navigation should not be considered a source of non-determinism.

3.7 Summary

We have investigated the potential sources of non-determinism affecting game engines and explored the impact they may have on simulation variance. Memory checking not withstanding, errors associated with the lack of ECC are likely to be minimal unless there is significant background radiation or 1000's of hours of computation are expected. To ensure precise simulation outcomes the physics setting dt must be fixed, along with any actor navigation meshes, random number seeds, game engine setup and simulation specific parameters. NUMA should only affect interprocessor data transfer cost and without control measures will only make the computation cycle longer. Relative access times between different caches are likely to be small although may have a more pronounced impact on high throughput systems, e.g. HPC. Basic thread scheduling should not affect the simulation's determinism unless changing scheduling policy, operating system or migrating between machines with different setups. However, should new and unexpected threads start during the simulation, then the interruption to execution order or additional resource demand may affect timing of subsequent steps, thus reducing the number of physics updates within a game loop. Likewise, uncontrolled allocation of hardware resources such as CPUs or GPUs can potentially give rise to non-determinism.

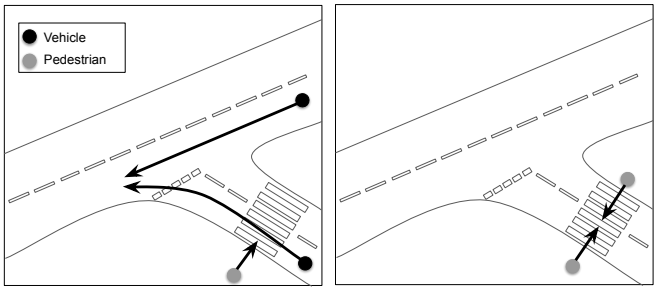


Fig. 3: Schematic of test scenarios for (a) Tests 1-4, (b) Tests 5-6. Descriptions are given in Table 1.

4 Case Study of Simulation Variance

We present an empirical investigation into using game engines for simulation-based verification of autonomous vehicles with a focus on characterising sources of non-determinism in order to understand the impact they have on simulation variance. Gao et al. [53] took a similar approach investigating Java applications, where a set of sources of non-determinism (termed factors) were shown to impact on repeatability of testing. Ultimately, our objective is to control non-determinisim to minimise simulation variance.

We first describe the context, scene and scenario of interest before discussing and defining a tolerance for what is considered an acceptable simulation variance in this context.

4.1 Context, Scene and Scenario

This case study draws on a setup used to verify an urban mobility and transport solution, where the primary verification objective is to test the behaviour of an ego vehicle in an urban environment at a T-junction in response to pedestrians and other vehicles. Thus, the scene for our investigation is the T-junction (T-intersection) and the scenarios are shown in Figure 3.

This scene was used to create a number of scenarios involving pedestrians and vehicles in order to identify any changes in the actor trajectories over repeated tests executed under a variety of systematically designed conditions and hence study any simulation variance. The vehicles and pedestrians were given trajectories, via predefined waypoints, that would result in either colliding with or avoiding other actors.

4.2 Tolerable Simulation Variance

To achieve stable verification results over repeated test runs, the simulated actor states must be precise to a specific tolerance. Deterministic behaviour would result in zero variance of the simulated actor states but if this cannot be achieved then what is permissible? This tolerance must be appropriate to allow accurate assertion checking and coverage collection in the simulation environment,

but not so small as to fail with minor computational perturbations. Thus, a tolerance must be defined to reflect the precision at which repeatability of simulation execution is required.

For this case study a tolerance on actor position of 1m would be insufficient when considering the spacial resolution required to distinguish between a collision and a near-miss event. A very small value, e.g. $1 \times 10^{-12} m$, may be overly-sensitive to minor computational perturbations and generate false positives. Therefore, for this case study and more broadly our general verification requirements, a tolerance of 1cm has been selected. Thus any variance of less than 1cm is permissible. To put this another way, we can accept a precision with a tolerance of $\leq \pm 1cm$.

Case study results are shown in terms of the maximum deviation, $\max \sigma$, from the mean actor path over the entire simulation history where any value higher than the specified tolerance is considered non-permissible.

4.3 Actor Collisions

Previous investigations into the Unreal Engine indicated that collisions between dynamic actors and solid objects, termed *blocking physics bodies* in Unreal Engine documentation [54], can lead to high simulation variance, [55]. Collisions and the subsequent physics calculations that are processed, termed *event hit* callback in Unreal Engine, were seen as potentially key aspects to the investigation into simulation variance.

The tests are listed in Table 1 and were chosen to cover a range of interactions between actor types. Tests 1 & 2 involve 2 vehicles meeting at a junction where they do not collide (test 1) and when they do tiggering an *event hit* callback in the game engine (test 2). In both cases the trajectories of the vehicles are hard-coded to follow a set of waypoints spaced at 0.1m intervals using a PID controller. In test 3 a mixture of different actor types are introduced where two vehicles drive without collision and a pedestrian walks across the road at a crossing point. In test 4 this pedestrian collides with one of the vehicles at the crossing, triggering a *event hit* callback, see Fig 3a. Similar to vehicles, pedestrians navigate via a set of regularly spaced waypoints at 0.1m intervals using the A* algorithm which is the default controller for the CARLA pedestrian actors. Tests 5 & 6 involve only pedestrians that do not collide (test 5) and that do collide (test 6), see Fig. 3b.

4.4 Experiment Description

For each test the position of each actor was logged at $0.1s$ intervals providing a trace of that actor's trajectory with respect to simulation time. Repeated traces are compared at the same time index t for actor a to provide a value for the variance, $\sigma_a^2(t)$. Herein the results are given in terms of the deviation, $\sigma_a(t)$, which indicates the dispersion of the actor position data relative to the mean and is helpfully in the same units as actor position, m, for ease of interpretation. The maximum variance over the entire set of n simulations is therefore defined as the largest variance of any actor at any time in the simulation,

$$\max_{a,t} \sigma_a^2(t), \tag{1}$$

and therefore the maximum deviation is simply square root of the maximum variance and herein referred to as $\max \sigma$ for brevity.

The maximum deviation, $\max \sigma$, was analysed against the different scenarios and settings that were identified as potential sources of non-determinism, and compared against the limit of *permissible variance* to indicate if the simulation was reliable for verification purposes.

4.5 Internal Settings

Within Unreal Engine there are numerous internal settings relating to the movement and interaction of physical bodies in the simulation. Settings can be adjusted to alter how actors interact and path plan via the navigation mesh of the environment, e.g. *Contact Offset* and *Navmesh Voxel Size*, or can be changed to improve the fidelity of physics calculations between game update steps, e.g. *Physics Sub-Stepping* and *Max Physics Delta Time*. Other options such as *Enable Enhanced Determinism* were investigated along with running the engine from the command line with options for more deterministic behaviour `-deterministic`, floating point control

Test	Actors	Collision	Collision Type	n	$\max \sigma$ (m) (unrestricted)	$\max \sigma$ (m) (restricted)
1	Two vehicles	No	N/A	1000	0.03	7.0×10^{-3}
2	Two vehicles	Yes	Vehicle and Vehicle	1000	0.31	9.8×10^{-3}
3	Two vehicles and a pedestrian	No	N/A	1000	0.07	5.2×10^{-4}
4	Two vehicles and a pedestrian	Yes	Vehicle and Pedestrian	1000	0.59	1.5×10^{-12}
5	Two pedestrians	No	N/A	1000	5.6×10^{-13}	5.6×10^{-13}
6	Two pedestrians	Yes	Pedestrian and Pedestrian	1000	5.6×10^{-13}	5.6×10^{-13}

Table 1: A description of the test scenarios showing the test number, the actors included, if a collision occurred and between which actors. Where n is the number of repeats and $\max \sigma$ is the maximum simulation deviation. The term *unrestricted* refers to an unrestricted account of the results including results of any resource utilisation. To understand the impact of collisions and high resource utilisation, the *restricted* column shows a subset of the results where post-collision data and experiments above 75% resource utilisation have been removed.

`/fp:strict` and headless mode `-nullrhi` along with running the test as a packaged release by building and cooking [56]. An initial study into the Unreal Engine using a pedestrian and a moving block was used to investigate simulation variance against these settings. The results were compared to a baseline of the default engine settings. However, none of these options improved simulation variance significantly and all internal setting were set restored to the default values. Details on this previous investigation can be found on the Trustworthy Systems GitHub [55].

4.6 External Settings

After discovering that internal engine settings could not achieve a simulation variance that met the required *tolerance*, settings external to the game engine were explored. Game engines, and simulation hardware more generally, will utilise available system resources such as central and graphical processing units (CPU and GPU respectively), to perform physics calculations and run the simulation environment. For high performance simulations the demand on these resources may be a significant fraction of the total available and an initial hypothesis was that as this ratio tended toward one, there would be an increase in *simulation variance*. This hypothesis was supported by our initial work performed on the unreal engine [55] and was explored more fully in this work using the CARLA platform.

To replicate in a controlled manner the high computational loads that may be anticipated for high performance simulations, software that artificially utilises resources on both the CPU and GPU were executed alongside the simulation. Resource utilisation was artificially increased for both CPU and GPU devices to include a range of values from 0-95% (see Section 5) using reported values of the system monitors `htop` and `nvidia-smi` respectively. Resource utilisation figures reported here should be considered approximate values.

Practitioners should also be aware that many libraries for calculating variance itself may require attention to get precise results. For example the `numpy` method of variance is sensitive to the input precision and will return an incorrect answer if the wrong parameters are set [57]. In `matlab`, the calculation of variance may switch from single thread to a multi-threaded execution not obviously apparent to the user when the input data size becomes large enough, opening up the potential for concurrency imprecision [58].

4.7 System Configuration and Screening

These tests were carried out on an Alienware Area 51 R5 with an i9 9960X processor with 64GB non-ECC DDR4 RAM at 2933MHz with an NVIDIA GeForce RTX 2080 GPU with 8GB GDDR6 RAM at 1515 MHz. Operating systems was Linux Ubuntu 18.04.4 LTS. To ensure reliability of results, each test was repeated 1000 times. Tests were carried out in CARLA (v0.9.6 and Unreal Engine v4.22) using synchronous mode with a fixed dt of 0.05s.

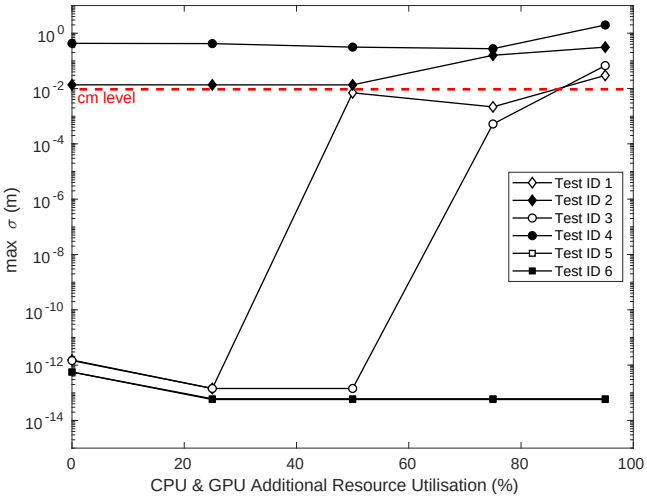


Fig. 4: Summary of results showing maximum deviation for each scenario against different resource utilisation levels. Tests 5 and 6 overlap having almost identical results.

A detailed guide for reproducing these experiments along with the scripts used are provided on github[4]. To eliminate some of the potential sources of non-determinism outlined in Section 3 a series of screening tests and analyses were performed on our system. These were:

- System memory: `memtest86` [59] full suite of tests ran, all passed.
- Graphical memory: `cuda_memtest` [60][61].
- Non-uniform memory access: `numactl` [62] was used to fix the simulator and test script to single cores but only a minor (2%) improvement in simulation variance was observed and therefore not used for subsequent testing.

4.8 Results and Discussion

A summary of all the results are shown Table 1 in the column $\max \sigma$ (unrestricted), where the value reported is the maximum deviation across all resource utilisation levels, i.e. the worst case for a given scenario. From these results it is clear that scenarios with only pedestrian actors (tests 5-6) display results within tolerance over all resource utilisation levels with or without a collision where $\max \sigma$ is 5.6×10^{-13} or 0.56pm. However, all other scenarios involving vehicles or a mixture of actor types breach the tolerance that has been set, with some deviation in actor path as large as 59cm. Clearly a deviation in results of such a large amount is unacceptable if simulation is to be used as a reliable verification tool.

Resource utilisation level was found to have a significant impact on *simulation variance*. Figure 4 shows $\max \sigma$ against the artificially increased resource utilisation level, where the x-axis indicates the approximate percentage of resource utilisation (for CPU & GPU). In this figure, anything above the 1cm level (dashed line) is considered

[4] `https://github.com/TSL-UOB/CAV-Determinism/tree/master/CARLA_Tests_setup_guide`

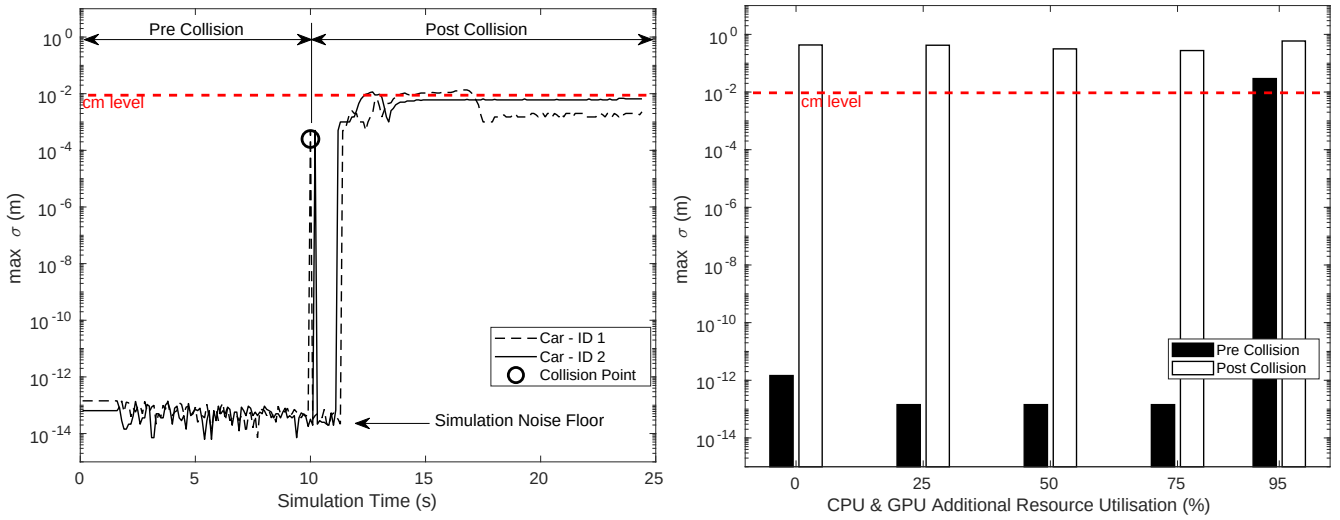


Fig. 5: Vehicle to vehicle collision (test 2) showing (a) maximum deviation against simulation time for 25% resource utilisation and (b) maximum deviation pre- and post-collision against resource utilisation. The simulation noise floor is shown in (a) which is the empirical lower limit of deviation for the hardware reported in this study.

non-permissible according to our specific tolerance level informed from our verification requirements.

A general pattern in the results indicates that; some scenarios consistently fail to produce results within permissible levels of variance at any resource utilisation level (Fig. 4 Test ID 2 & 4 above the dashed cm level), some are consistently within tolerance (Test ID 5 & 6 with pedestrians only), and that some cases only fail to meet the tolerance requirement when at higher utilisation levels, breaching the tolerance above 75% resource utilisation (Test ID 1 & 3).

However, the non-permissible results in Figure 4 (all those above the dashed line) are the worst case account of the situation, as per Equation 1, as the maximum variance is taken over the entire simulation period.

Examining specifically the results from tests 2 & 4 as a function of simulation time reveals further information about the simulation variance before and after an actor collision. Fig. 5a shows this examination for vehicle to vehicle collisions (test 2), where $\max \sigma$ switches from permissible prior to the vehicle collision to non-permissible post-collision. This pattern of permissible results prior to collision and non-permissible post-collision is maintained up to a resource utilisation level of approximately 75%, see Fig. 5b. This time series examination was repeated for vehicle to pedestrian collisions (test 4) and the results are shown in Fig. 6a. Similarly to vehicle-to-vehicle collisions, the variation of $\max \sigma$ for vehicle to pedestrian collisions indicates permissible pre-collision behaviour with up to 75% resource utilisation, see Fig. 6b. This is a key finding of this work; it suggests that verification engineers should consider terminating tests at the point of a collision, as any post-collision results will be non-permissible.

The second key finding of this work is illustrated in Fig. 6a. In this scenario (test 4), there is a collision between a vehicle (Car ID 2, solid line) and a pedestrian (Ped ID 3, dot dash line) which occurs at a simulation time of approximately 6s and a second vehicle actor (Car ID 1, dashed line), which is *not involved in the collision*. There are three observations here; firstly that the vehicle directly involved in the collision (Car ID 2) displays high simulation variance immediately after the collision. Secondly, that the maximum deviation of the pedestrian involved in the collision (Ped ID 3) is at a tolerable level throughout the test[5]. Thirdly, we observed a delayed effect on Car ID 1 showing high simulation variance with a 5s delay *even though this vehicle was not involved in the collision*. This final point should be of particular concern to verification engineers and research practitioners in the field as it implies that *any collision between actors can affect the simulation variance of the entire actor population* and could potentially result in erroneous assertion checking.

The main findings of this work suggest a working practice that would minimise the non-deterministic effects observed in this investigation. By limiting simulation results to pre-collision data and ensuring resource utilisation levels do not exceed 75%, the permissible variance of 1cm is achievable as shown in the *restricted* column in Table. 1. By applying this set of restrictions upon the simulation the maximum observed deviation across all experiments was 0.98cm which is within the target tolerance we set out to achieve. Practitioners may wish to set a stricter resource utilisation level, such as less than 50% to further reduce the potential dispersion of results if this is required for their chosen application.

4.9 Process Scheduling Priority

An investigation into the response of process scheduling on the simulation variance was undertaken. The experi-

[5] However, please note that in CARLA the pedestrian object is destroyed post-collision hence the flat line from $t = 6$s onwards.

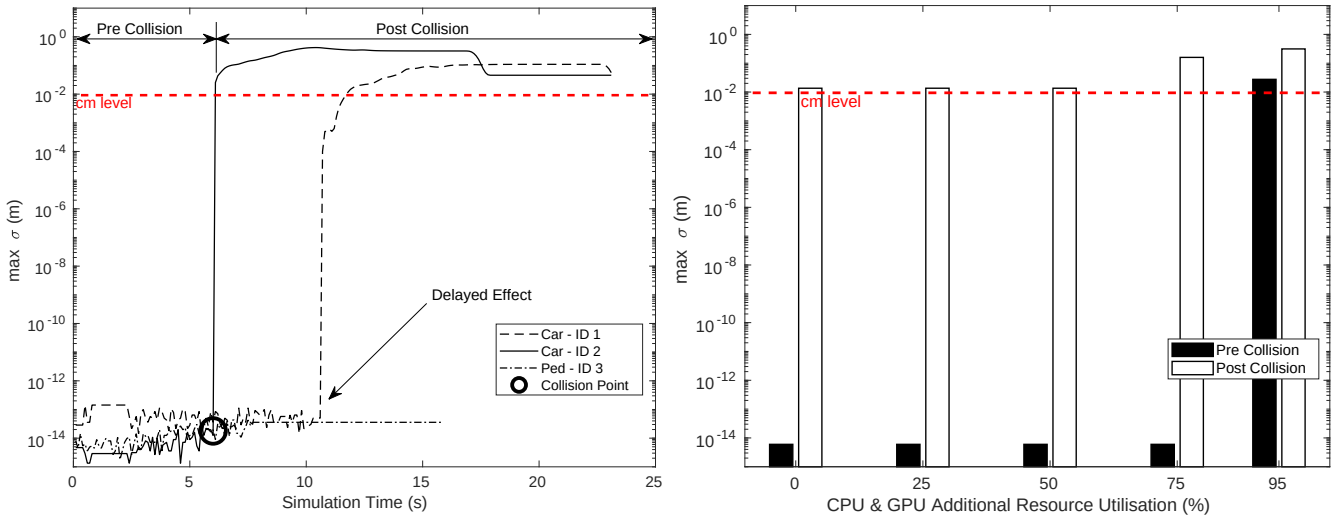


Fig. 6: Vehicle to pedestrian collision (test 4) showing (a) maximum deviation against simulation time for 25% resource utilisation and (b) maximum deviation pre- and post-collision for different resource utilisation levels.

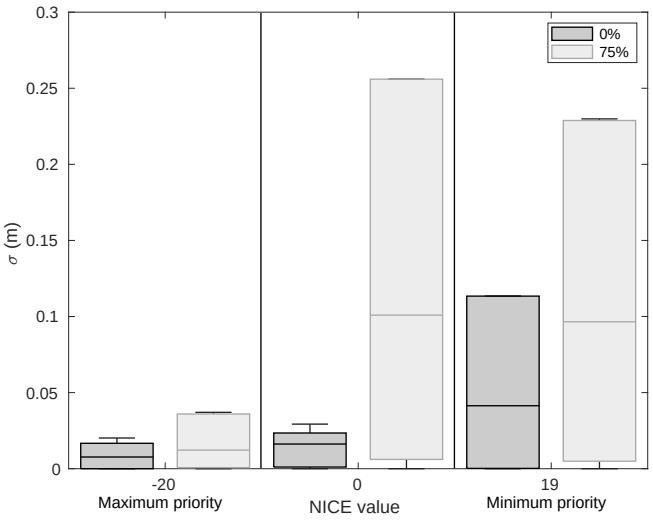


Fig. 7: Summary of investigating the effect of NICE priority setting for 0% and 75% additonal CPU & GPU resource utilisation.

ment was repeated ($n = 1000$) using test 1 but altering the process scheduling priority using the program `NICE`.[6] Setting a higher priority for the simulator process with respect to the resource utilisation processes, it was possible to determine if scheduling could account for the increased simulation variance when the system is under high resource utilisation. To give a process a high priority a negative NICE value is set with the lowest being -20. To decrease the priority a positive value is set, up to +19. The default NICE value is 0.

The results are presented in Fig. 7 where the box denotes the inter-quartile range, non-outlier limits by the whiskers and a horizontal bar for the median. The figure shows that decreasing the priority (right hand side of plot) has little effect on simulation variance when compared to a NICE value of 0 (central plot bars). Increasing priority (left hand side of plot) reduced variance for the 75% resource utilisation but this did not account for all the difference in the observed results. This is demonstrated in the maximum priority setting where the bars in plot are not equal, indicating an additional contribution to variance not accounted for by the NICE scheduling. This unaccounted difference in the variance may be due to the lack of absolute control that `NICE` has over the process scheduling.[7]

4.10 Investigation Summary

This empirical investigation has highlighted the shortcomings of using a games engine for simulation based verification and advice for best working practice. However, these results are specific to the hardware and software used in the study and may not be transferable to other systems directly. Therefore we now present a general methodology that practitioners can follow to find the *operational domains of permissible variance* for any hardware configuration.

5 Methodology

In this section a method for determining the variance of simulated actor path trajectories, termed *simulation variance*, and resolving the operational domains of permissible variance of the simulation presented here as a work flow, see Fig. 8. In addition a number of recommendations and best practices are suggested that other practitioners can follow to minimise simulation variance.

[6] `http://manpages.ubuntu.com/manpages/bionic/man1/nice.1.html`

[7] `https://askubuntu.com/questions/656771/process-niceness-vs-priority`

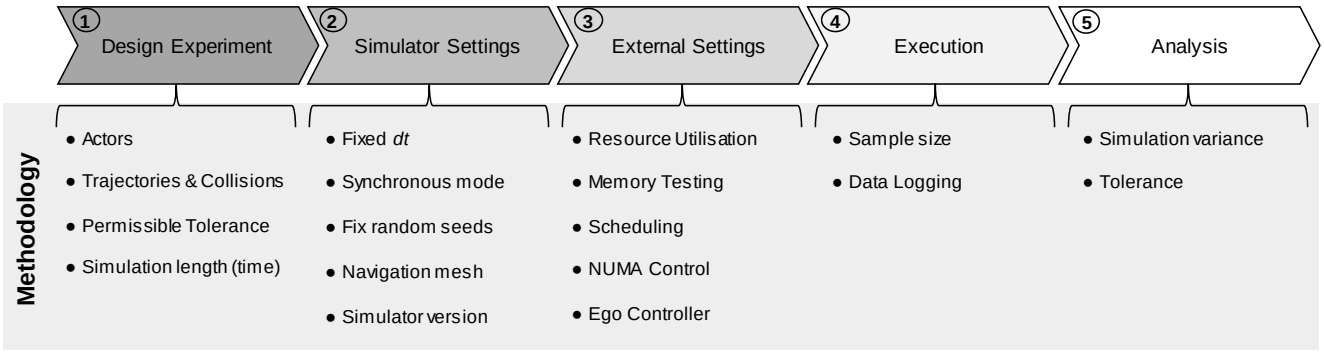


Fig. 8: Flow diagram of the methodology proposed to determine the simulation variance of a simulation.

5.1 Design Experiment

5.1.1 Actors

Different actor types may be handled differently by the game engine and our investigation indicated that CARLA pedestrians (termed *walkers*) do not suffer from simulation variance under any conditions that were tested. However the introduction of CARLA vehicles increased the simulation variance of the pedestrian-only test by 12 orders of magnitude. As such, all actors that could be included in any future simulation should be tested, including any non-standard CARLA or bespoke actors.

5.1.2 Trajectories and Collisions

Actor paths, or the sequence of actions required to generate paths, should be hard-coded to ensure repeatability. These paths should include collisions between actors and potentially collisions between actors and static scenery if this is likely to occur in the simulation or as part of the verification process. Actor paths without collision are also important to include as these will serve as a baseline to the other tests.

5.1.3 Setting a Tolerance

The verification engineer should set the *tolerance* based on their specific requirements which refers to the acceptable level of actor path variance. This tolerance level is analogous to the analogue signal level associated with each of the binary states in digital circuits. At some low analogue voltage the signal is interpreted as binary 0, this may be, for example, any voltage below 0.5V, and the same signal is interpreted as binary 1 at some higher voltage, e.g. above 3.5V. This convention enables the abstraction from the noisy physical signal values that are analogue voltages into the digital domain with binary representation. The tolerance that the user must set depends on the objectives of the simulation and the granularity at which the simulation environment operates. For example, this tolerance must be sufficiently small to enable accurate assertion checking. In our empirical investigation we set a tolerance of 1cm which is sufficient for urban scenario assertion checking. In practice, it may be necessary to determine this tolerance experimentally.

5.1.4 Simulation Length

The simulation time should be sufficient to record an interaction between actors but not so long that the testing take an inconvenient time to complete. In our empirical investigation a simulation time of $10-20$s was sufficient to monitor the distinct change in events such as the pre- and post-collision including the delayed effect seen by other actors, shown in Fig. 6a.

5.2 Simulator Settings

The settings internal to the game engine or other simulation environment should be set to ensure a fixed physics time step, dt. If using CARLA a small fixed value, say $0.05s$, can be set by using the following: `setting.fixed_delta_seconds = 0.05`, whereas in Unity the default fixed time step is set to $0.02s$ [63]. In CARLA, synchronous mode must be used to allow communication to external controllers which ensures no sensor data are passed out of order to the simulator which is particularly important if a complex ego controller is used [64]. The use of random numbers must be controlled through the use of fixed seeds which might be used to control variations of background effects, e.g. weather patterns, or the navigation of random pedestrian actors, external vehicle controllers or other clients connected to the simulation environment. Actors that navigate through the environment should use a fixed navigation mesh.

The version number of the CARLA and Unreal environment has also be shown to affect results, see [55], so ensuring consistent version number throughout testing is also important.

5.3 External Settings

5.3.1 Resource Utilisation

The resources available to the simulator have been shown to have a significant effect on the simulation variance of simulated vehicles and exploring this as a variable the analysis is required. CPU utilisation software, such as the linux `stress` tool which is a workload generator program can be used spawn workers on any number of cores or virtual threads on a system. This can be used to artificially increased the load on the system to explore at what point your system may become susceptible to *simulation*

variance. For GPU utilisation `gpu-burn` was employed using the `fur test` where different resolutions and multiple instances can be used to tune graphical utilisation levels [65]. Reported values of resource utilisation can be obtained using the system monitors `htop` and `nvidia-smi` for CPU and GPU respectively. These values can also be written into the data logging for completeness.

Alternatively, in place of artificial resource utilisation multiple instances of the simulation could be executed simultaneously. However, the granularity of control with this approach may be reduced.

5.3.2 Memory Testing

Prior to experimental execution the system hardware should be tested for memory conformity and to ensure no single bit errors are occurring, see 4.7. For mainboard memory `memtest86` can be used on most platforms to run a series of pre-defined memory test patterns. This memory testing software can also be used for ECC enabled hardware. Similarly, to test memory on Nvidia based graphical adaptors `cuda_memtest` can be used to ensure no memory errors exist.

5.3.3 Scheduling

We hypothesise that thread scheduling may be a major contributor to the non-deterministic results found in the empirical study. However, fine control over the scheduling policy and thread execution order may be non-trivial. Such policies are defined by the operating system and may execute threads with equal priority. In such a case, all tasks non-essential to the simulator should be terminated to prevent interruption to the simulator.

Setting the simulator process to a higher priority may help to alleviate conflicting task scheduling which can be achieved by using, for example `TaskSettings.Priority` in Windows [66] or `NICE` in Linux [67].

5.3.4 NUMA Control

Control over Non-Uniform Memory Access policy can be achieved using `numactl` for multiprocessors with shared memory. This control allows the simulator program to be fixed on a single core reducing memory access time. Initial screening tests done with NUMA control gave moderate improvements in simulation variance of the order of a few percent, see Section 4.7. This control may assist if simulation variance is borderline to the tolerance but not seen as essential.

5.3.5 Ego Vehicle Controller

No ego vehicle was used in the empirical investigation but this should be considered as an actor but ensuring that any randomness used in the controller is controlled with fixed seeds and that any adaptive learning algorithms should be controlled in the same manner.

5.4 Execution

5.4.1 Sample Size

Initial testing [55] indicated an actor path deviation of 1×10^{-13}cm for 997 out of 1000 tests but with 3 tests reporting a deviation of over ~10cm. While executing 100 repeats may seem sufficient, this sample size may fail to observe these events that occur with low probability, inferring a false confidence in the results. It is therefore recommended that the sample size is found empirically dependent on the number of results that exceed the tolerance.

5.4.2 Data Logging

Data logging should be used to record the actor positions at fixed time intervals throughout the simulation in order to determine the variance in actor path. Additional information can also be logged such as the CPU and GPU utilisation levels and engine specific metrics such as game loop latency. For the subsequent analysis, the actor should have an identifier and the repeat number and simulation time should also be captured.

5.5 Analysis

The maximum value of actor path deviation over all time samples and actors, $\max \sigma$, should be analysed to identify which of the candidate sources of non-determinism require restriction or control for permissible operation. Finding these boundaries will identify the *domains of permissible variance* for the user specific hardware.

This method can be extended to a broader actor states and actions including actor orientation, speed, and any other status indicators that may be of interest and also including appropriate actions that may be useful for verification purposes. This broader As such, Equation 1 would need to be adjusted to include these new variables.

6 Conclusions & Future Work

The autonomous vehicle community are adopting game engine simulators for such purposes as control system development and verification. Having a deterministic simulator is required for precise results, to find and fix software bugs and ensure results are trustworthy. If a simulator is non-deterministic then practitioners should at least be aware of, and how to find, the operational domains for reliable results. During an investigation into the CARLA simulator, a *simulation variance* was observed for repeated tests with the same initial conditions indicating non-deterministic execution. During the study we hypothesized and uncovered several parameters that contribute towards greater simulation variance and hence non-deterministic execution, such as actor collisions and system resource utilisation. The results of the investigation are hardware and software version specific, so we proposed a general methodology that can be used to find the *domains of permissible variance* for any given system

configuration. This methodology will allow the AV verification community using simulation tools to ensure their results are reliable and trustworthy.

A potential avenue of exploration is developing a version of the simulator which has been designed specifically for verification. In this version the simulator would be deterministic because the execution is suitably controlled along with sources of randomness and scheduling. A similar task was achieved with the `RR` program [42].

With the advent of more AI systems becoming adaptive, meaning that the system may change it's response to the same stimulus over time, the notion of repeatability will need reassessing in the context of verification. The main research question here is what will constitute 'the same' which will be important for passing or failing verification tests but also for determining coverage.

Acknowledgement

Abanoub Ghobrial and Greg Chance contributed equally to this paper.

This research has in part been funded by the ROBOPILOT and CAPRI projects. Both projects are part-funded by the Centre for Connected and Autonomous Vehicles (CCAV), delivered in partnership with Innovate UK under grant numbers 103703 (CAPRI) and 103288 (ROBOPILOT), respectively.

References

1. "Vienna convention on road traffic." `https://treaties.un.org`. Accessed: 2019-10-03.
2. "The Highway Code." `https://www.gov.uk/guidance/the-highway-code`, 2015. Accessed: 2019-11-25.
3. "Uk road traffic act 1988." `http://www.legislation.gov.uk/ukpga/1988/52/contents`. Accessed: 2019-10-03.
4. "Preliminary report highway hwy18mh010," *National Transportation Safety Board*, 2018.
5. K. Korosec, "Waymo's self driving cars hit 10 million miles." `https://techcrunch.com/2018/10/10/waymos-self-driving-cars-hit-10-million-miles`, 2019. Accessed: 2019-11-26.
6. N. Kalra and S. M. Paddock, "Driving to safety: How many miles of driving would it take to demonstrate autonomous vehicle reliability?," *Transportation Research Part A Policy and Practice*, vol. 94, pp. 182–193, 2016.
7. P. Koopman and M. Wagner, "Toward a Framework for Highly Automated Vehicle Safety Validation," *SAE Technical Paper Series*, vol. 1, pp. 1–13, 2018.
8. C. Hutchison, M. Zizyte, P. Lanigan, D. Guttendorf, M. Wagner, C. Le Goues, and P. Koopman, "Robustness testing of autonomy software," *International Conference on Software Engineering Software Engineering in Practice Track*, 2018.
9. "Carla: Open source simulator for autonomous driving research." `http://carla.org/`. Accessed: 2020-011-13.
10. "Unreal engine 4." `https://www.unrealengine.com/`. Accessed: 2020-011-13.
11. 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 (CoRL)*, pp. 1–16, 2017.
12. S. Ulbrich, T. Menzel, A. Reschka, F. Schuldt, and M. Maurer, "Defining and Substantiating the Terms Scene, Situation, and Scenario for Automated Driving," *IEEE Conference on Intelligent Transportation Systems, Proceedings, ITSC*, vol. 2015-Oct, pp. 982–988, 2015.
13. A. Y. Javaid, W. Sun, and M. Alam, "UAVSim A simulation testbed for unmanned aerial vehicle network cyber security analysis," *2013 IEEE Globecom Workshops, GC Wkshps 2013*, pp. 1432–1436, 2013.
14. Z. Saigol and A. Peters, "Verifying automated driving systems in simulation framework and challenges," *25th ITS World Congress, Copenhagen*, 2018.
15. J. Schumann, P. Gupta, and Y. Lui, "Application of Neural Networks in High Assurance Systems: A Survey," *Applications of Neural Networks in High Assurance Systems. Springer, Berlin, Heidelberg,*, pp. 1–19, 2010.
16. T. Atkins and M. Escudier, *A Dictionary of Mechanical Engineering*. Oxford University Press, 2013.
17. "Unity game engine." `https://unity.com/`. Accessed: 2020-011-13.
18. "Cry engine." `https://www.cryengine.com/`. Accessed: 2020-011-13.
19. "Airsim drones simulator." `https://microsoft.github.io/AirSim/`. Accessed: 2020-011-13.
20. "Apollo autonomous driving soultion." `http://apollo.auto/`. Accessed: 2020-011-13.
21. Nvidia, "Nvidia drive constellation." [Accessed: 24-06-2020].
22. P. E. Strandberg, T. J. Ostrand, E. J. Weyuker, W. Afzal, and D. Sundmark, "Intermittently failing tests in the embedded systems domain," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2020, (New York, NY, USA), pp. 337–348, Association for Computing Machinery, 2020.
23. C. Staff, "To catch a failure: The record-and-replay approach to debugging," *Commun. ACM*, vol. 63, pp. 34–40, July 2020.
24. Unity, "Order of execution for event functions." [Accessed: 24-06-2020].
25. J. Gregory, *Game Engine Architecture*. CRC Press, 2 ed., 2017.
26. R. Nystrom, *Game Programming Patterns*. Genever Benning, 1 ed., 2011.
27. T. d. Margerie, "Precise frame rates in unity." [Accessed: 24-06-2020].
28. J. Austin, "Fix your unity timestep." [Accessed: 04-03-2020].
29. G. Fiedler, "Fix your timestep," 2004. Accessed: 2019-12-18.
30. "Substepping ue4." `https://docs.unrealengine.com/en-US/Engine/Physics/Substepping/`. Accessed: 2020-011-13.
31. J.-M. Muller, N. Brunie, F. d. Dinechin, C.-P. Jeannerod, M. Joldes, V. Lefevre, G. Melquiond, N. Revol, and S. Torres, *Handbook of Floating-Point Arithmetic*. Springer International Publishing, 2 ed., 2018.
32. D. Goldberg, "What every computer scientist should know about floating-point arithmetic," *ACM Computing Surveys (CSUR)*, vol. 23, no. 1, pp. 5–48, 1991.
33. N. Kapre and A. DeHon, "Optimistic parallelization of floating-point accumulation," *Proceedings - Symposium on Computer Arithmetic*, pp. 205–213, 2007.
34. N. Whitehead and A. Fit-Florea, "Precision & Performance: Floating Point and IEEE 754 Compliance for NVIDIA GPUs." `https://developer.nvidia.com`, 2011.

35. Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, (New York, NY, USA), pp. 643–653, Association for Computing Machinery, 2014.
36. "Ieee standard for floating-point arithmetic," *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pp. 1–84, July 2019.
37. A. Nötzli and F. Brown, "Lifejacket: Verifying precise floating-point optimizations in llvm," in *Proceedings of the 5th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, SOAP 2016, (New York, NY, USA), pp. 24–29, Association for Computing Machinery, 2016.
38. C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM (JACM)*, vol. 20, no. 1, pp. 46–61, 1973.
39. "What is multithreading?." `https://docs.unity3d.com/Manual/JobSystemMultithreading.html`. Accessed: 2020-011-13.
40. D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, M. Young, J. F. Crespo, and D. Dennison, "Hidden technical debt in machine learning systems," *Advances in Neural Information Processing Systems*, vol. 2015-January, pp. 2503–2511, 2015.
41. E. Breck, S. Cai, E. Nielsen, M. Salib, and D. Sculley, "The ml test score: A rubric for ml production readiness and technical debt reduction," *Proceedings - 2017 IEEE International Conference on Big Data, Big Data 2017*, vol. 2018-January, pp. 1123–1132, 2017.
42. "Rr lightweight tool for recording." `https://github.com/mozilla/rr`. Accessed: 2020-011-13.
43. J. Nieplocha, R. J. Harrison, and R. J. Littlefield, "Global arrays: A nonuniform memory access programming model for high-performance computers," *The Journal of Supercomputing*, vol. 10, no. 2, pp. 169–189, 1996.
44. T. J. Dell, "A white paper on the benefits of chipkill-correct ECC for PC server main memory," *IBM Microelectronics Division*, pp. 1–23, 1997.
45. P. E. Dodd and L. W. Massengill, "Basic mechanisms and modeling of single-event upset in digital microelectronics," *IEEE Transactions on nuclear Science*, vol. 50, no. 3, pp. 583–602, 2003.
46. N. Mielke, T. Marquart, N. Wu, J. Kessenich, H. Belgal, E. Schares, F. Trivedi, E. Goodness, and L. R. Nevill, "Bit error rate in nand flash memories," in *2008 IEEE International Reliability Physics Symposium*, pp. 9–19, IEEE, 2008.
47. "Stat commands." `https://docs.unrealengine.com/en-US/Engine/Performance/StatCommands/`. Accessed: 2020-011-13.
48. F. Codevilla, "Carla 0.8.2 driving benchmark." `http://carla.org/2018/04/23/release-0.8.2/`. [Accessed: 26-03-2019].
49. "Unreal engine 4 ai programming essentials." `https://www.oreilly.com/library/view/unreal-engine-4/9781784393120/ch04s03.html`. Accessed: 2020-011-13.
50. L. Kliemann and P. Sanders, *Algorithm Engineering Selected Results and Surveys*. Springer, 2 ed., 2016.
51. S. Miglio, "Ai in unreal engine learning through virtual simulations." `https://www.unrealengine.com/en-US/tech-blog/ai-in-unreal-engine-learning-through-virtual-simulations`. [Accessed: 26-03-2019].
52. U. S. Team, "Ai and behavior trees." `https://docs.unrealengine.com/en-us/Gameplay/AI`. [Accessed: 26-03-2019].
53. Z. Gao, Y. Liang, M. B. Cohen, A. M. Memon, and Z. Wang, "Making system user interactive tests repeatable: When and what should we control?," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, pp. 55–65, 2015.
54. "Collision overview - unreal engine 4." `https://docs.unrealengine.com/en-US/Engine/Physics/Collision/Overview/index.html`. Accessed: 2020-011-13.
55. "Investigating unreal engine for deterministic behaviour." [Accessed: 06-03-2020].
56. "Releasing your project - unreal engine 4." `https://docs.unrealengine.com/en-US/Engine/Deployment/Releasing/`. Accessed: 2020-011-13.
57. "Numpy variance." `https://numpy.org/doc/stable/reference/generated/numpy.var.html`, 2020. Accessed: 2020-09-03.
58. "Does matlab use all cores by default when running a program? - matlab answers." `https://uk.mathworks.com/matlabcentral/answers/317128-does-matlab-use-all-cores-by-default-when-running` Accessed: 2020-011-13.
59. "Memtest86 - the standard for memory diagnostics." `https://www.memtest86.com/`. Accessed: 2020-011-13.
60. "Cuda memtest - tests gpu memory for hardware errors and soft errors using cuda." `https://github.com/ComputationalRadiationPhysics/cuda_memtest`. Accessed: 2020-011-13.
61. G. Shi, J. Enos, M. Showerman, and V. Kindratenko, "On testing gpu memory for hard and soft errors," in *Proc. Symposium on Application Accelerators in High-Performance Computing*, vol. 107, 2009.
62. "Numactl - control numa policy for processes or shared memory." `https://linux.die.net/man/8/numactl`. Accessed: 2020-011-13.
63. "Monobehaviour - unity game engine." `https://docs.unity3d.com/ScriptReference/MonoBehaviour.FixedUpdate.html`. Accessed: 2020-011-13.
64. Carla, "Carla: Configuring the simulation." [Accessed: 04-03-2020].
65. V. Timonen, "Gpu burn." [Accessed: 17-12-2019].
66. "Windows task settings priority property." `https://docs.microsoft.com/en-us/windows/win32/taskschd/tasksettings-priority`. Accessed: 2020-011-13.
67. "Nice - run a program with modified scheduling priority." `https://linux.die.net/man/1/nice`. Accessed: 2020-011-13.