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

Greg Chance, Abanoub Ghobrial, Kevin McAreavey, Séverin Lemaignan, Tony Pipe, Kerstin Eder

***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, in a deterministic simulation environment, tests are rendered repeatable and yield simulation results that are trustworthy and straightforward to debug. However, game engines are seldom deterministic. This paper reviews and identifies the potential causes and effects of non-deterministic behaviours in game engines. A case study using CARLA, an open-source autonomous driving simulation environment powered by Unreal Engine, is presented to highlight 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 sufficiently high i.e. variance between repeated executions 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 engine simulations for any given system configuration.**

***Index Terms*—Autonomous Driving, Autonomous Vehicles, Determinism, Physics Engines, Verification and Validation (V&V), Simulation, Testing**

I. 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 [68], national rules [61] and road traffic laws [63] which form a body of safe and legal driving rules, termed assertions, that must not be violated.

Manuscript received ...; revised ...; accepted.... Date of publication ...; date of current version ....

Greg Chance and Abanoub Ghobrial contributed equally to this paper, Greg Chance is the corresponding author.

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.

Greg Chance (e-mail: greg.chance@bristol.ac.uk), Abanoub Ghobrial (e-mail: abanoub.ghobrial@bristol.ac.uk), Kevin McAreavey (e-mail: kevin.mcareavey@bristol.ac.uk), and Kerstin Eder (e-mail: kerstin.eder@bristol.ac.uk) are with the Trustworthy Systems Lab, Department of Computer Science, University of Bristol, Merchant Ventures Building, Woodland Road, Bristol, BS8 1UB, United Kingdom.

Séverin Lemaignan (e-mail: severin.lemaignan@brl.ac.uk) and Tony Pipe (e-mail: tony.pipe@brl.ac.uk), are with the Bristol Robotics Lab, T Block, University of the West of England, Frenchay, Coldharbour Ln, Bristol, BS34 8QZ, United Kingdom.

Digital Object Identifier ....

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 several fatalities with AVs, some of which could be attributed to insufficient verification and validation (V&V), e.g. [47]. Simulation environments offer a means to explore the vast parameter space safely and efficiently [31] without the need for millions of miles of costly on-road testing [28]. In particular, simulations can be biased to increase the frequency at which otherwise rare events occur [30]; this includes testing how the AV reacts to unexpected behaviour of the environment [24].

Increasingly, the autonomous vehicle community is adopting game engines as simulation platforms to support the development and testing of vehicle control software. CARLA [8], for instance, is an open-source simulator for autonomous driving that is implemented in the Unreal Engine [67], a real-time 3D creation environment for the gaming and film industry as well as other creative sectors [16].

State-of-the-art game engines provide a convenient option for simulation-based testing. Strong arguments exist that they offer sufficient realism [30] 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 set up and run compared 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 [64]. Finally, support for hardware-in-the-loop development or a real-time test-bed for cyber-security testing [27] may also be provided if required. Compared to the vehicle dynamics simulators and traffic-level simulators used by manufacturers [52], game engines offer a simulation solution that meets many of the requirements for the development and functional safety testing of the autonomous features 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*.

In this paper, we investigate non-determinism and how it affects simulation results using the example of CARLA, an open-source autonomous driving simulation environment based on the Unreal game engine. In our case study, scenarios between pedestrian and vehicle actors are investigated to determine the actor position variance in the simulation output for repeated simulation runs. We find that the CARLA simulator is non-deterministic. Actor path variance was found to be non-zero and, under certain conditions, the deviation from the mean was observed up to 59cm. We would consider a deviation of 1cm to be permissible for autonomous vehicle verification tasks in an urban environment. However, we found that CARLA exhibits a permissible variance when system utilisation is restricted to 75% or less and the simulation is terminated 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 suitability of a game engine for AV simulation. In particular, this can give a better understanding of the effects of non-determinism and to what extent simulation precision may impact verification results.

This paper is structured as follows. Section II defines terms used throughout the paper and identifies when determinism is needed. Section III briefly introduces how game engines work before investigating in Section IV the potential sources of non-determinism in game engines. An empirical case study of simulation variance for a number of scenarios involving pedestrian and vehicles is given in Section V including internal and external setting and system screening tests. The results from the case study are presented in Section VI. Section VII presents the step-by-step method to assess the suitability of a simulation system for AV verification in general. We conclude in Section VIII and give an outlook on future work.

II. Preliminaries

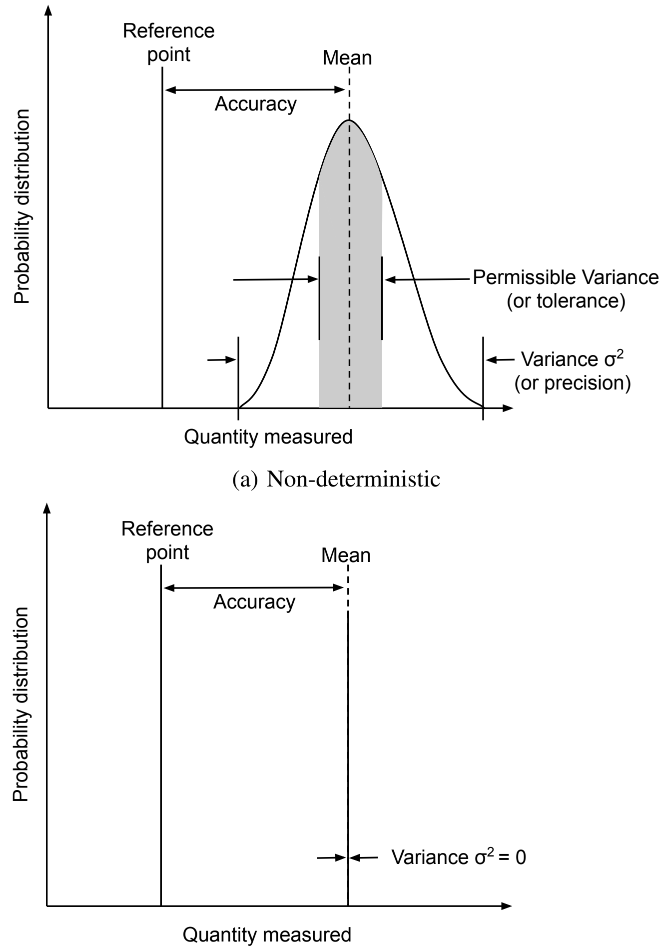
A. Definitions

Several definitions are introduced in this section. These are used in the subsequent discussion. Refer to Fig. 1 throughout this section.

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 [53]. 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 simulation can be thought of as the process of generating or producing 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, Fig. 1 (b), 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, Fig. 1 (a), then there will be a variation in the generated output data.

2) Variance, Precision & Tolerance: We adopt terminology from the mechanical engineering and statistics domains to describe when there is variation in the generated output data [3]. *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 under the same conditions and actions. *Tolerance* is defined as the permissible limit of the variance, or in short the *permissible variance*.



(a) Non-deterministic

(b) Deterministic

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

As an analogy, the simulator can be thought of as a manufacturing process that produces data. To determine the precision of this process, the output must be measured and analysed for differences when the process is repeated. Those differences describe the spread or variance in the process output. A hard limit on the variance can then be defined, Fig. 1 (a), beyond which the output fails to meet the required tolerance, e.g. the output is rejected by quality control. Real manufacturing fails to achieve absolute precision. Hence, there is a need for tolerances to be specified to account for the variance in real-world manufacturing processes.

If a simulator is deterministic then it will produce results with absolute precision or zero variance, Fig. 1 (b), and hence will be within tolerance by design. If the simulator is non-deterministic then there will be a measurable, non-zero variance in the output data.

3) Accuracy: Precision and tolerance should not 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. Therefore, we define accuracy as the difference between the true value, or reference value, and what has been achieved in the data generation process or simulation. 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 practice, full accuracy will often not be achievable due to modelling and associated computational demands of creating and executing an exact replica. In most cases it is unnecessary and some authors state that 'just the right amount of realism' is required to achieve valid simulation results [30].

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) in a 3D environment recorded at regular time intervals. This definition could be extended to include other variables. A set of simulation traces derived from the same input and starting state then forms the experimental data on which variance is calculated for a given simulation run.

5) Simulation Variance & Deviation: If the simulator is non-deterministic then how can the simulation variance be measured? This can be achieved by monitoring the values of any of the recorded output variables that should be consistent from run to run. For example, actor position variance is a distance-based metric that can be derived from simulation traces. The actor position over time, i.e. the actor path, is often used in assertion checking, e.g. to determine whether vehicles keep within lanes or whether minimum distances to other vehicles and road users are being maintained. Thus, in the case study presented in this paper, the term *simulation variance*, measured in SI unit m^2, refers to a measure of actor path variance in the simulation, assuming fixed actions. Case study results are presented using deviation (SI unit m), the square root of variance, rather than variance, as this is a more intuitive measure to comprehend when interpreting test results.

6) Scene, Scenario & Situation: We adopt the terminology defined for automated driving in [64], 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.

B. When is Determinism needed?

Determinism is a key requirement for simulation during AV development and testing. A deterministic simulation environment guarantees that tests are repeatable, i.e. repeated runs of a test, given the same initial conditions and event history, produce the same output data. Thus, a deterministic simulator has zero *variance*.

A simulator with non-zero variance can no longer be considered deterministic. Non-deterministic simulators may be sufficient for certain applications as long as their variance is permissible, i.e. *within tolerance*. Therefore, *tolerance* is the acceptable degree of variability between repeated simulations. 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, e.g. it has a non-zero variance in, for example, actor positions, then this may, in the best case, lead to intermittent assertion failures, making it difficult to reproduce, understand and remove bugs and rendering verification results unstable. In the worst case, however, bugs that could have been identified in simulation remain undetected, leading to 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 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 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 questions are: *How can one assess whether a simulation environment is deterministic?* and *How can one determine and control the simulation variance?*

III. Background

There are numerous game engines with their associated development environments that could be considered suitable for AV development, e.g. Unreal Engine [67], Unity [66], CryEngine [11]. 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 [8], AirSim [1], Apollo [2], and some have been developed for cloud-based simulation, e.g. Nvidia Drive Constellation [45].

Investigating the determinism of game engines has not attracted much research interest to date since performance is more critical for game developers than accurate and repeatable execution. Ensuring software operates deterministically is a non-trivial task. Catching intermittent failures, or flaky tests [59], in a test suite that cannot be replayed makes the debugging process equally difficult [57]. 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 [65]. A game loop is broken up into three distinct phases: processing the inputs, updating the game world (Physics Engine), and generating outputs (Rendering) [23].

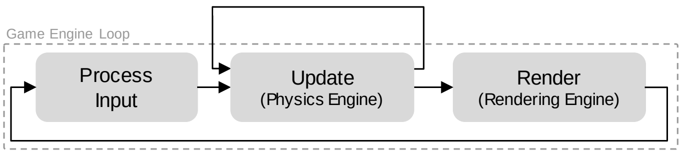


Fig. 2: Game engine loop block diagram [46].

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 update will process frames as fast as the computational processing will allow up to the maximum monitor refresh rate [34]. 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 [4].

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 [20]. 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 [60]. 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 needs a faithful representation of reality (i.e. gamer accepts low accuracy) nor require repeated actions to result in the same outcome (i.e. gamer accepts low precision). In contrast, high accuracy and precision are necessary for AV development, testing and verification.

IV. 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. [59], although the AV simulation domain introduces its own unique challenges that were not considered in that paper.

A. *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 [38, 22].

As a consequence, floating-point arithmetic is not associative. Thus, for arithmetic expressions beyond two operands, such as $x+y+z$, executing $x+(y+z)$ rather than $(x+y)+z$, can produce different results [29]. This is because rounding is being performed at different stages in the computation, depending on the respective execution order. Execution order changes can occur due to the use of a different compiler or as a consequence of parallelisation, e.g. within the runtime environment or at the hardware level. Furthermore, floating-point calculation results may differ if the execution is performed on a GPU rather than a CPU which may have different register widths [70].

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 conclude that floating-point arithmetic causes non-determinism, resulting in loss of repeatability. In fact, some authors suggest avoiding the use of floating-point representation entirely for assertion testing due to imprecision and non-deterministic behaviour [33].

In contrast, we argue that the precision issues related to floating-point operations are better described as *incorrectness* that is in fact repeatable; they do not cause non-determinism per se. It is normally reasonable to assume that the processors on which game engines run are deterministic with respect to floating-point arithmetic. Thus, given the same operands and sequence of operations, the same processor should always produce the same bit pattern. So, even if the result of a floating-point operation is incorrect due to floating-point rounding errors, it should always be equally *incorrect* for implementations that conform to the IEEE floating-point standard [25].

However, as illustrated above on the example of non-associative addition, a change of execution order can lead to sequences of floating-point operations producing different results. Thus, any uncontrolled options to modify execution order, e.g. at the compiler level, within the runtime environment or at the hardware level, can cause simulation results to differ between runs, resulting in non-zero variance and loss of determinism.

Beyond that, aggressive optimisations by the compiler can also introduce incorrectness in floating-point arithmetic [42]. However, the same executable should still return the same output for identical input.

In conclusion, floating-point arithmetic does not cause non-zero *simulation variance* for repeated simulation runs when using the same executable on the same hardware with exactly the same configuration and execution order.

B. 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 depending on the operating system's scheduler policy. A scheduler policy may be optimised in many ways such as for task throughput, deadline delivery or minimum latency [32]. In principle, 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 [69], 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 [54, 6].

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 debugging program `rr` [51] 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, thus promoting deterministic behaviour by design [57].

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*.

C. Non-Uniform Memory Access (NUMA)

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 inter-processor data transfer cost [41]. 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.

D. 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 [13]. Single bit errors may occur due to malfunctioning hardware, ionising radiation (background cosmic or environmental sources) or from electromagnetic radiation [14]. 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 [36].

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.

E. 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* [58] 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.

F. Actor Navigation

In game engines, the A* search algorithm is often used for navigation to find optimal paths for actors. The optimality and validity of A* solutions is only guaranteed if the environment is deterministic, and if not, then there are no guarantees that an execution of the same A* solution will lead to the same outcome across multiple runs.

To briefly describe the A* process, at each stage of the search, the algorithm selects one node for expansion from a set of nodes known as the *frontier* (also fringe or open set) using a heuristic to select this node. For reasons of efficiency, A* represents the frontier as a priority queue, with nodes prioritised by the heuristic. Selection then reduces to a dequeue operation on the frontier. In general the heuristic does not guarantee a uniquely optimal node in the frontier and thus ties may be encountered. The question of how these ties are broken in a priority queue is related to the broader notion of sorting stability [55]. An implementation of a priority queue is said to be stable if it preserves insertion order, otherwise it is said to be unstable. If an implementation of A* uses a stable priority queue, then it will behave as a deterministic algorithm which always returns the optimal node that was first added to the frontier. Stable priority queues are typically less computationally efficient than unstable priority queues, so the requirement on stability is often dropped in practice.

For example, one of the most common implementations of a priority queue relies on a binary heap and is unstable; it will always return the first optimal node according to its internal heap order rather than insertion order. The default priority queue implementations in the standard Java and Boost libraries are based on binary heaps [5, 48], while the Unreal documentation and source code suggests that this is also true of CARLA [17, 19, 18]. Thus, an unstable but deterministic priority queue is an unstable priority queue that preserves some order, but not necessarily insertion order. If an implementation of A* uses an unstable but deterministic priority queue, then it will behave as a deterministic algorithm which always returns the optimal node according the ordering of its priority queue.

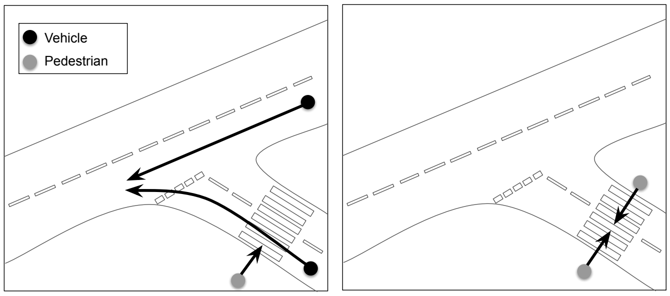
Finally, an unstable but deterministic priority queue suggests a third kind of priority queue that we can refer to as an unstable and non-deterministic priority queue; that is, a priority queue that always returns an optimal node selected at random (i.e. breaking ties arbitrarily). If an implementation of A* uses an unstable and non-deterministic priority queue, then it will behave as a non-deterministic algorithm. Overall, an implementation of A* is much more likely to use a priority queue that is unstable but deterministic rather than one that is stable, and is very unlikely to use one that is unstable and non-deterministic.

An implementation of A* will behave as a deterministic algorithm if it uses a stable priority queue or an unstable but deterministic priority queue. However, apparent determinism of A* will be sensitive to the ordering preserved by the priority queue (i.e. insertion order or e.g. heap order). If we are not aware of changes to this ordering, then a deterministic implementation of A* may appear to behave as a non-deterministic algorithm, since we will not have accounted for changes to the ordering. For example, suppose actions are iterated over in order of hash value (where order of iteration over actions often determines insertion order), but between runs there is some untracked change to the software/hardware stack that changes the calculation of the hash value, then this may change insertion order of nodes to the frontier, which could eventually materialise as a different optimal solution found by A*. This suggests that while a simulated environment may behave deterministically, factors outside the simulator may cause changes to the operation of an A* implementation, which would then materialise as non-deterministic runs in the simulator. The difference between the use of a stable priority queue or an unstable but deterministic priority is simply that it may be easier to detect such changes under a stable priority queue because insertion order is typically more meaningful with respect to the actual implementation of A*.

To summarise, documentation suggests that CARLA uses a binary heap for the priority queue [17, 19, 18] which preserves some order of the frontier nodes, but not necessarily the insertion order, and hence is unstable. Therefore, determinism of the A* algorithm for actor navigation will be sensitive to the ordering preserved by the priority queue, in this case the heap order. This also assumes the simulation environment itself is deterministic as sources of non-determinism that affect the simulator or the system more generally may cause changes to the operation of an A* implementation.

G. 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, seeds for random number generation, game engine setup and simulation specific parameters. Any implementation of the A* search algorithm for actor navigation must use a stable priority queue to ensure deterministic results. Non-uniform memory access (NUMA) should only affect interprocessor data transfer cost



(a) (b)

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

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. If this change in computational cycle gives opportunity for the execution order to be changed then this situation may lead to non-determinism.

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.

V. 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. [21] 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-determinism 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. A discussion on the internal and external settings of the simulation is included, along with system configuration and pre-screening sections.

A. 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 with the scenarios as 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 paths 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 pre-defined waypoints, that would result in either colliding with or avoiding other actors.

B. 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? The tolerance must be appropriate to allow accurate assertion checking and coverage collection in the simulation environment, but not so small such that assertion checking would 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, for 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 $\leq\pm1cm$.

In the following, 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.

C. Actor Collisions

Previous investigations into the Unreal Engine indicated that collisions between actors and solid objects, termed *blocking physics bodies* in Unreal Engine documentation [10], can lead to high simulation variance [26]. Collisions and the subsequent physics calculations that are processed, termed *event hit* callback in Unreal Engine, were identified as potentially key aspects to the investigation into simulation variance.

The tests used for the case study are listed in Table I. They cover a range of interactions between actor types. Tests 1 & 2 involve two vehicles meeting at a junction where they either do not collide (Test 1) and where they do collide (Test 2), thereby triggering an *event hit* callback in the game engine. 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 is introduced where two vehicles drive without collision and a pedestrian walks across the road at a crossing point.

Similar to vehicles, pedestrian actors navigate via a set of regularly spaced waypoints at 0.1m intervals using the A* search algorithm which is the default method to find optimal paths for the CARLA pedestrian actors [39]. There is evidence

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 I: A description of the test scenarios showing the test number, the actors included, whether a collision occurred and if so then between which actors. n, the number of repeats is set to 1000 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.

to suggest that this actor navigation in CARLA could be a source of non-deterministic simulation behaviour [9]. This behaviour is explored in Test 4 where a pedestrian collides with one of the vehicles at the crossing, triggering an *event hit* callback, see Fig 3a.

Tests 5 & 6 involve only pedestrians that either, do not collide (Test 5) and that do collide (Test 6), see Fig. 3b.

D. Evaluation Metric

For each test the position of each actor is logged at $0.1s$ intervals providing a trace of that actor's path with respect to simulation time. The logs from repeated tests are sourced to establish a value for the variance associated with each actor, a, at each time point t, giving a variance function over time for each actor, $\sigma_a^2(t)$.

Instead of using variance, herein the results are given in terms of the deviation, $\sigma_a(t)$, which indicates the dispersion of the actor path relative to the mean and is helpfully in the same units as actor position, i.e. metres (m), for ease of interpretation. The maximum variance over the entire set of n repeated simulations, i.e. the overall observed worst case, is defined as the largest variance of any actor at any time in any of the simulation runs, as given in Equation 1.

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

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

The maximum deviation, $\max \sigma$, can be analysed for 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 is sufficiently accurate for verification purposes.

E. Simulator 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 `/fp:strict` and headless mode `-nullrhi` along with running the test as a packaged release by building and cooking [49]. 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 [26].

F. External Settings

After examining controls the internal game engine, external settings 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*.

Some initial exploratory work was undertaken [26] that suggested computational resource utilisation was positively correlated with simulation variance, leading to high simulation variance when a system is under high load. This initial observation 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 to 95% (see Section VII) using reported values of the system monitors `htop` and `nvidia-smi` respectively. Resource utilisation figures reported here should be considered approximate values. Resource utilisation was capped to 75% in some parts of the results and referred to as *restricted*. This was done to limit the

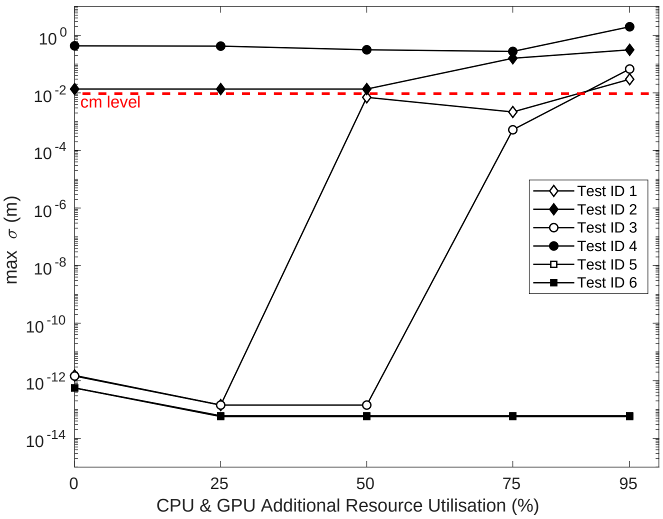


Fig. 4: Summary of results showing maximum deviation for each scenario against different resource utilisation levels. Tests 5 and 6 overlap having almost identical results. Note that the lines between data points are only a guide.

load on the test system and hence limiting simulation variance. The term *unrestricted* places no such limit on the system load.

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 [44]. 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-induced imprecision [15].

G. Experimental System Configuration and Pre-Screening

The experiments 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. The operating systems was Linux Ubuntu 18.04.4 LTS. Tests were carried out in CARLA (v0.9.6 and Unreal Engine v4.22) using synchronous mode with a fixed dt of 0.05s.

Initial testing [26] indicated an actor path deviation of 1×10^{-13}cm for 997 out of 1000 tests, with three tests reporting a deviation of over ~10cm. While executing 100 repeats may seem sufficient, this sample size may fail to observe events that occur with low probability, giving false confidence in the results. Therefore each test was repeated 1000 times to provide a sufficient sample size. A detailed guide for reproducing the experiments along with the scripts used are provided on github[1]. To eliminate some of the potential sources of non-determinism outlined in Section IV a series of screening tests and analyses were performed on our system. These were:

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

- System memory: `memtest86` [35] full suite of tests ran, all passed.
- Graphical memory: `cuda_memtest` [12], no failures on all 11 tests [56].

VI. Results and Discussion

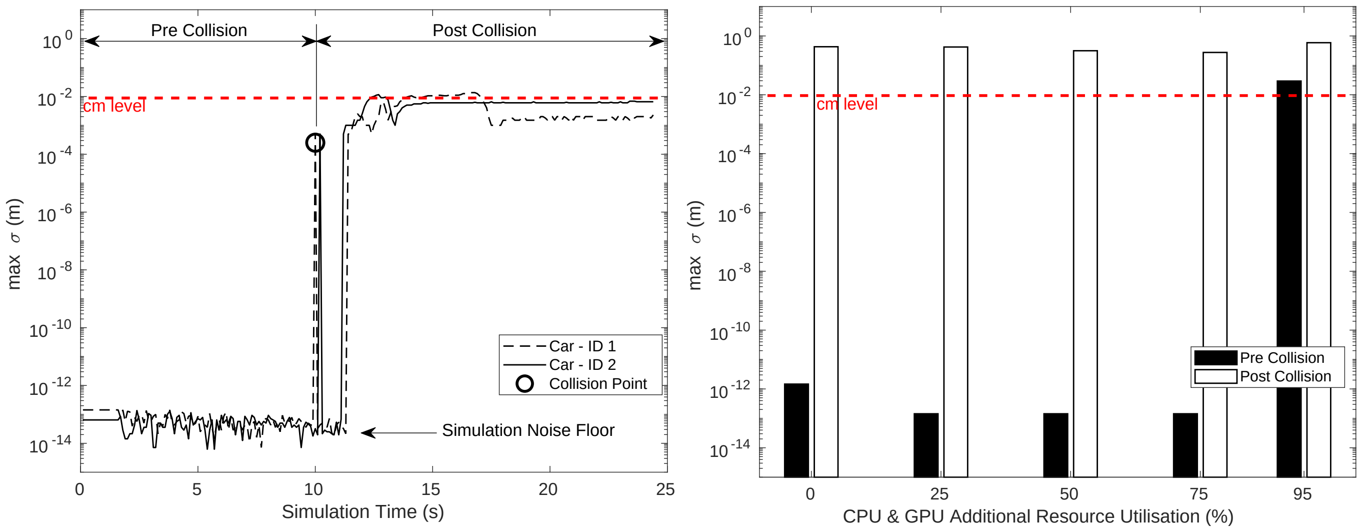
A summary of the main results are shown in Table I. In the column $\max \sigma$ (unrestricted) the value reported is the maximum deviation across all resource utilisation levels, i.e. the worst case for a given scenario. From these results it is clear that scenarios with only pedestrian actors (Tests 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 $0.56\mathrm{pm}$. However, all other scenarios involving vehicles or a mixture of actor types do not meet the required tolerance, with some deviation in actor path as large as 59cm. Clearly, such a large deviation cannot be acceptable for simulation to be considered a credible verification tool.

Resource utilisation 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, any $\max \sigma$ above the 1cm level (indicated by a dashed line) is considered non-permissible according to our specified tolerance. Note that 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.

A general pattern in the results indicates that some scenarios consistently fail to produce results within tolerance, irrespective of resource utilisation (cf. Fig. 4 Test 2 & 4 are above the dashed 1cm line), while some are consistently within tolerance (cf. Fig. 4 Test 5 & 6 both are with pedestrians only), and some cases only fail to meet the tolerance requirement at higher resource utilisation levels, i.e. above 75% resource utilisation (cf. Fig. 4 Test 1 & 3).

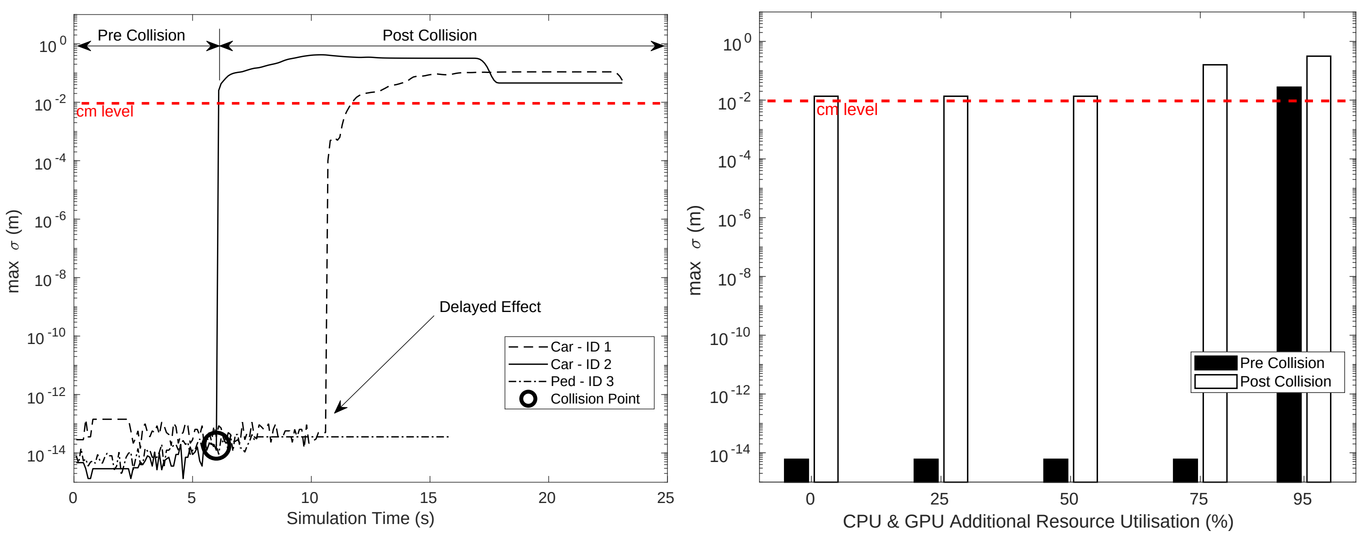
Examining specifically the results from Tests 2 & 4 as a function of simulation time reveals further information about the simulation variance before and after an actor collision. Fig. 5a shows this examination for vehicle to vehicle collisions (Test 2), where $\max \sigma$ switches from permissible prior to the vehicle collision to non-permissible post collision. The 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; 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



(a) (b)

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.



(a) (b)

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.

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; 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[2]. 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, developers and researchers 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 simulation results.

To conclude, the main findings of this case study suggest a working practice that would minimise the factors that give rise to 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 I. By applying this set of restrictions upon the simulation the maximum observed deviation across all experiments was 0.98cm which is within

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

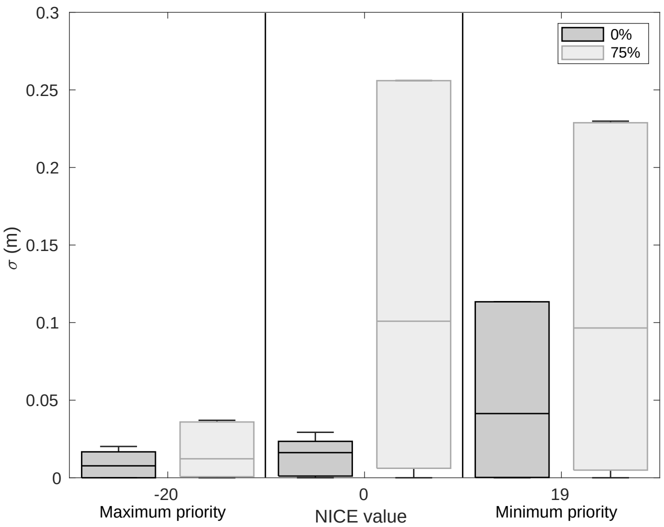


Fig. 7: Variance range of three NICE priority settings for additional CPU & GPU resource utilisation of 0% and 75%.

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.

A correlation between resource utilisation and simulation variance has been observed in these results which may be due to process scheduling. Rather than limiting utilisation of the whole system, another approach may be to promote the scheduling priority of the simulation process which is explored in Section VI-A. Furthermore, we were keen to investigate the impact of memory access and include a brief investigation of this in Section VI-B.

A. Process Scheduling Priority

An investigation into the impact of process scheduling on the simulation variance was undertaken following the observations of increased simulation variance with increased resource utilisation. The experiment was repeated ($n = 1000$) using Test 1 but altering the process scheduling priority using the program `NICE`.[3] 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 of deviation, non-outlier limits by the whiskers and a horizontal bar for the median. The figure shows that decreasing the priority of the simulator process (right hand side of the plot) has little effect on simulation variance when compared to a default NICE value of 0 (central bars in the plot). Increasing priority (left hand side of the plot) significantly reduced the variance for the 75% resource utilisation experiment, but this does not account for all the difference in the observed results. This can be seen in the maximum priority setting where the bars in the plot are not equal, indicating an additional contribution to variance not accounted for by the NICE scheduling. The remaining difference in the variance between the two resource utilisation levels may be due to the lack of absolute control that `NICE` has over process scheduling.[4]

B. Non-uniform memory access (NUMA)

An additional investigation into memory access was undertaken given the potential impact that data transfer cost and execution order may have on simulation variance, see Section IV-C. The program `numactl` [43] allows a process to run with a specified memory placement policy, essentially allowing the process to be bound to a particular CPU core. `numactl` was used to fix the simulator and test script to single cores, and a (2%) improvement in simulation variance was observed. This was considered minor comparative to the changes in simulation variance observed in other aspects of the case study, for example the pre- and post-collision between a vehicle and pedestrian saw a change of 10^{14}. Therefore `numactl` was not used for subsequent testing.

However, practitioners that are aiming to minimise simulation variance should use a technique such as this to minimise data transfer cost and potentially minimise any potential effect on process execution order that may lead to increased simulation variance.

C. Investigation Summary

These empirical investigations have highlighted the shortcomings of using a games engine for simulation based verification and suggests advice for best working practice. It was observed that resource utilisation positively correlates with simulation variance and that specific simulation events, such as vehicle collisions, can also lead to a breach in permissible tolerance. The investigation found that the effect of higher simulation variance as a result of increased resource utilisation can be reduced, but not omitted entirely, by controlling the scheduling policy. The investigation into specifying memory placement on the simulation process did improve simulation variance but only by a minor amount of 2%.

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 have derived a general methodology that practitioners can follow to find the *operational domains of permissible variance* for a game-engine-based simulation environment. This methodology is presented in the next section.

VII. Method to Determine the Variance of a Simulation

In this section a method for determining the simulation variance of actor paths and resolving the operational domains of permissible variance is presented here as a work flow,

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

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

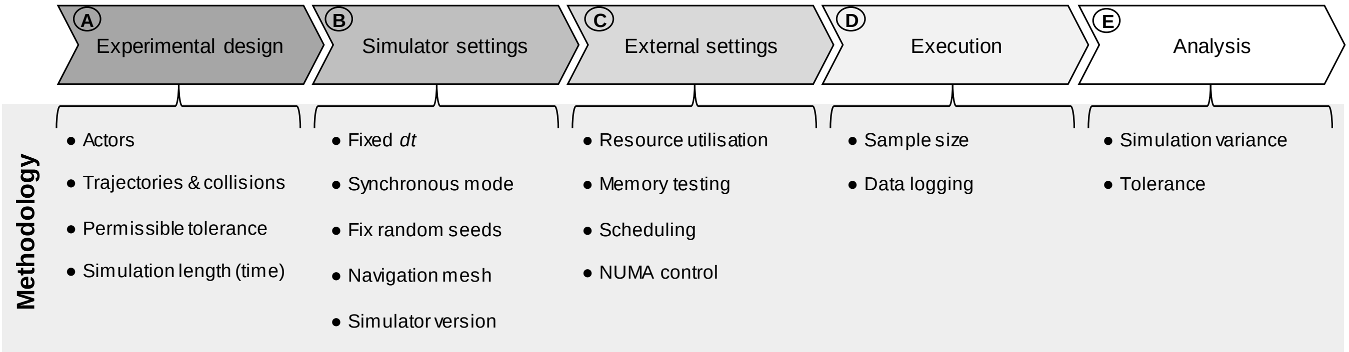


Fig. 8: Stages of the method proposed to determine the variance of a simulation.

see Fig. 8. In addition, recommendations and best practice guidelines for minimising simulation variance are suggested.

The method consists of a sequence of five stages; experimental design, simulator settings, external settings, execution and analysis. In the following, each stage is described in detail with reference to the items listed for each stage in Fig. 8.

A. Experimental Design

The experiment design is based around a series of carefully selected tests that varies one of the sources of non-determinism whilst keeping all other parameters constant. Each test is repeated n times and the analysis of the results provide a confidence with a degree of statistical certainty. By varying a single parameter and controlling all others, the simulation variance associated with each source of non-determinism can be found and addressed.

1) Actors: All actors that could be included in the simulation should be tested, including any non-standard CARLA or bespoke actors including the ego vehicle, see Section VII-C.

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; to see if deviations between runs in actor paths increases with collisions.

3) Permissible Variance: The verification engineer should set the *permissible variance*, also termed *tolerance*. The tolerance 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 and coverage collection, but not so small for assertion results to differ for repeated runs. In Section V-B a tolerance of 1cm was considered sufficient for urban scenario assertion checking. In practice, it may be necessary to determine this tolerance experimentally.

4) Simulation Time: The simulation time will depend on the actor paths and terminating conditions. The simulation time should be sufficient to record interactions between actors but not so long that the testing takes 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 in Section VI. The termination conditions of the simulation can be set by, for example, actors reaching their final trajectory waypoints.

B. 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 `setting.fixed_delta_seconds = 0.05`. In Unity, the default fixed time step is set to $0.02s$ [37].

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 [7].

The use of random numbers must be controlled through fixed seeds, resulting in pseudo-randomness. Random numbers might be used in the simulation environment 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 [26]. Therefore, ensuring a consistent version number throughout testing is also important.

C. External Settings

1) Resource Utilisation: The resources available to the simulator have been shown to have a significant effect on the variance of the path of simulated vehicles. Thus, it is important to understand at what level of resource utilisation the system running the simulation becomes susceptible to *simulation variance*.

CPU utilisation software, such as the linux `stress` tool, which is a workload generator program, can be used to spawn workers on any number of cores or virtual threads on a system. This can be used to artificially increase the load on the system. For GPU utilisation, `gpu-burn` can be employed using the `fur test`. Different resolutions and multiple instances can be used to tune graphical utilisation levels [62]. Reported values of resource utilisation can be obtained using the system monitors `htop` and `nvidia-smi` for CPU and GPU, respectively. These values should be added to the data logs. 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.

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 Section V-G. 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.

3) Scheduling: We hypothesise that thread scheduling may be a major contributor to the non-deterministic results found in the empirical study. However, gaining fine control over the scheduling policy and thread execution order is non-trivial [57].

The operating system schedules threads according to a specified scheduling policy, potentially based on equal thread priority. Thus, in such a case, all tasks non-essential to the simulator should be terminated to prevent interference with the simulation.

Assigning a higher priority to the simulator process may help to alleviate conflicting task scheduling which can be achieved by using, for example `TaskSettings.Priority` in Windows [71] or `NICE` in Linux [40].

4) NUMA Control: Control over a Non-Uniform Memory Access policy can be achieved using `numactl` for multi-core processors with shared memory. This control allows the simulator process to be fixed on a single core, reducing and unifying memory access time. Investigations in tests performed with NUMA control resulted in only minor improvements in simulation variance, see Section V-G. Using a fixed memory placement policy may assist if simulation variance is borderline to the tolerance but only of the order of a few percent from our observations.

5) Ego Vehicle Controller: An ego vehicle was not used in this study, but the impact of introducing this to the simulator can be considered here. The ego vehicle is seen as another actor in the simulation but care must be taken to ensure that any control algorithms, machine learning modules and processing pipelines are deterministic. We recommended that this be treated as a separate source of non-determinism and handled accordingly.

D. Execution

1) Sample Size: It is recommended that the chosen sample size, i.e. the number of repeated tests, is determined empirically. We recommend monitoring variance while increasing the sample size in orders of magnitude until there is confidence that $\max\sigma$ will not exceed the permissible variance for the verification process. Note that we suggest monitoring the maximum value of σ, not the average, because if even a single simulation run is outside of the permissible variance this may lead to false confidence in the verification result.

2) Data Logging: Unique identifiers should be assigned to each experiment, each repeat and each individual actor. The time-stamped actor positions should then be recorded at fixed time intervals throughout the simulation in order to determine the variance in actor path. Additional information should also be logged such as the CPU and GPU utilisation levels and engine specific metrics such as game loop latency.

E. Analysis

For each experiment, 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 to reach the domain of permissible variance within the simulation environment.

VIII. Conclusions & Future Work

Game engines offer simulation environments that are used for the development and verification of autonomous driving functions. Determinism of a simulator is required to achieve repeatability, which is essential to find and fix software bugs efficiently, and to ensure simulation results are trustworthy. If a simulator is non-deterministic then practitioners should at least be aware of, and know how to find, the operational domains where *simulation variance* is tolerable.

An investigation into the CARLA simulator revealed a significant simulation variance for repeated tests with the same initial conditions and event history, indicating non-determinism of the simulation. We then researched, identified and discussed potential sources for non-determinism in this context. A systematic case study of the CARLA simulator uncovered the actual factors that contribute towards greater simulation variance, giving rise to non-deterministic execution. In particular, actor collisions and system-level resource utilisation were identified as key contributors. To meet the recommended simulation variance for urban autonomous vehicle verification demonstrated in our case study, we recommend being aware of actor collisions and resource utilisation levels during simulation execution. Alternatively there are commercially available driving simulators that claim to be fully deterministic, e.g. RFPro [50], that may be more suitable if using a game engine does not provide a simulation variance sufficient for your verification requirements.

A general method to assess the actor path variance of a game-engine-based simulation environment was then proposed. The method can be used to find the *domains of permissible variance* of a simulation environment for a given system configuration. This can give AV developers and verification engineers increased confidence in the simulation results and reduce debug time. As future work, the method can be extended to other simulation platforms and to criteria other than the actor path, e.g. actor orientation and any status indicators that may be of interest, also including actions, sequences and timings that may be useful for verification purposes.

An ambitious avenue for future work is the development of a deterministic simulator for AV development and verification. This requires controlling all potential sources of non-determinism, including randomness and scheduling, very similar to the development of the record-and-reply debugger `rr` [51], originally developed to catch low-frequency non-deterministically failing tests at Mozilla [57].

IX. Acknowledgement

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. Special thanks to David May, Roger Shepherd, CFMS, Auroch Digital and Fortelix for their discussions and support on this topic.

References

[1] *AirSim drones simulator*. https://microsoft.github.io/AirSim/. Accessed: 2020-011-13.

[2] *Apollo autonomous driving soultion*. http://apollo.auto/. Accessed: 2020-011-13.

[3] T. Atkins and M. Escudier. *A Dictionary of Mechanical Engineering*. Oxford University Press, 2013.

[4] J. Austin. *Fix Your Unity Timestep*. https://johnaustin.io/articles/2019/fix-your-unity-timestep. [Accessed: 04-03-2020].

[5] *Boost C++ libraries, Basic priority queue interface*. https://www.boost.org/doc/libs/1_50_0/doc/html/heap/concepts.html. Accessed: 2021-04-14.

[6] 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". In: *Proceedings - 2017 IEEE International Conference on Big Data, Big Data 2017* 2018-January (2017), pp. 1123–1132.

[7] *CARLA: Configuring the simulation*. [Accessed: 04-03-2020]. URL: https://carla.readthedocs.io/en/0.8.4/configuring_the_simulation.

[8] *CARLA: Open source simulator for autonomous driving research*. http://carla.org/. Accessed: 2020-011-13.

[9] F. Codevilla. *CARLA 0.8.2 Driving Benchmark*. http://carla.org/2018/04/23/release-0.8.2/. [Accessed: 26-03-2019].

[10] *Collision Overview - Unreal Engine 4*. https://docs.unrealengine.com/en-US/Engine/Physics/Collision/Overview/index.html. Accessed: 2020-011-13.

[11] *Cry Engine*. https://www.cryengine.com/. Accessed: 2020-011-13.

[12] *Cuda memtest - Tests GPU memory for hardware errors and soft errors using CUDA*. https://github.com/ComputationalRadiationPhysics/cuda_memtest. Accessed: 2020-011-13.

[13] T. J. Dell. "A white paper on the benefits of chipkill-correct ECC for PC server main memory". In: *IBM Microelectronics Division* (1997), pp. 1–23.

[14] P. E. Dodd and L. W. Massengill. "Basic mechanisms and modeling of single-event upset in digital microelectronics". In: *IEEE Transactions on nuclear Science* 50.3 (2003), pp. 583–602.

[15] *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-a-program. Accessed: 2020-011-13.

[16] 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)*. 2017, pp. 1–16.

[17] U. Engine. *FGraphAStar, Generic graph A* implementation*. https://docs.unrealengine.com/en-US/API/Runtime/AIModule/FGraphAStar/index.html. Accessed: 2021-04-14.

[18] U. Engine. *Github GraphAStar.h*. https://github.com/EpicGames/UnrealEngine/blob/release/Engine/Source/Runtime/AIModule/Public/GraphAStar.h. Accessed: 2021-04-14.

[19] U. Engine. *TArray: Arrays in Unreal Engine*. https://docs.unrealengine.com/en-US/ProgrammingAndScripting/ProgrammingWithCPP/UnrealArchitecture/TArrays/index.html. Accessed: 2021-04-14.

[20] G. Fiedler. "Fix Your Timestep". In: (2004). Accessed: 2019-12-18.

[21] 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. 2015, pp. 55–65.

[22] D. Goldberg. "What every computer scientist should know about floating-point arithmetic". In: *ACM Computing Surveys (CSUR)* 23.1 (1991), pp. 5–48.

[23] J. Gregory. *Game Engine Architecture*. 2nd ed. CRC Press, 2017. Chap. 7.

[24] C. Hutchison et al. "Robustness Testing of Autonomy Software". In: *International Conference on Software Engineering Software Engineering in Practice Track* (2018).

[25] "IEEE Standard for Floating-Point Arithmetic". In: *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (July 2019), pp. 1–84.

[26] *Investigating Unreal Engine for deterministic behaviour*. [Accessed: 06-03-2020]. URL: https://github.com/TSL-UOB/CAV-Determinism/tree/master/UnrealEngineTests.

[27] A. Y. Javaid, W. Sun, and M. Alam. "UAVSim A simulation testbed for unmanned aerial vehicle network cyber security analysis". In: *2013 IEEE Globecom Workshops, GC Wkshps 2013* (2013), pp. 1432–1436.

[28] N. Kalra and S. M. Paddock. "Driving to safety: How many miles of driving would it take to demonstrate autonomous vehicle reliability?" In: *Transportation Research Part A Policy and Practice* 94 (2016), pp. 182–193.

[29] N. Kapre and A. DeHon. “Optimistic parallelization of floating-point accumulation”. In: *Proceedings - Symposium on Computer Arithmetic* (2007), pp. 205–213.

[30] P. Koopman and M. Wagner. “Toward a Framework for Highly Automated Vehicle Safety Validation”. In: *SAE Technical Paper Series* 1 (2018), pp. 1–13.

[31] 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. Accessed: 2019-11-26. 2019.

[32] C. L. Liu and J. W. Layland. “Scheduling algorithms for multiprogramming in a hard-real-time environment”. In: *Journal of the ACM (JACM)* 20.1 (1973), pp. 46–61.

[33] 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. Hong Kong, China: Association for Computing Machinery, 2014, pp. 643–653.

[34] T. d. Margerie. *Precise frame rates in Unity*. [Accessed: 24-06-2020]. URL: https://blogs.unity3d.com/2019/06/03/precise-framerates-in-unity/.

[35] *MemTest86 - The standard for memory diagnostics*. https://www.memtest86.com/. Accessed: 2020-011-13.

[36] N. Mielke et al. “Bit error rate in NAND flash memories”. In: *2008 IEEE International Reliability Physics Symposium*. IEEE. 2008, pp. 9–19.

[37] *MonoBehaviour - Unity Game Engine*. https://docs.unity3d.com/ScriptReference/MonoBehaviour.FixedUpdate.html. Accessed: 2020-011-13.

[38] J.-M. Muller et al. *Handbook of Floating-Point Arithmetic*. 2nd ed. Springer International Publishing, 2018.

[39] P. L. Newton and J. Feng. *Unreal Engine 4 AI Programming Essentials*. https://www.oreilly.com/library/view/unreal-engine-4/9781784393120/ch04s03.html. O’Reilly, 2016.

[40] *NICE - run a program with modified scheduling priority*. https://linux.die.net/man/1/nice. Accessed: 2020-011-13.

[41] J. Nieplocha, R. J. Harrison, and R. J. Littlefield. “Global arrays: A nonuniform memory access programming model for high-performance computers”. In: *The Journal of Supercomputing* 10.2 (1996), pp. 169–189.

[42] 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. Santa Barbara, CA, USA: Association for Computing Machinery, 2016, pp. 24–29.

[43] *Numactl - Control NUMA policy for processes or shared memory*. https://linux.die.net/man/8/numactl. Accessed: 2020-011-13.

[44] *NumPy Variance*. https://numpy.org/doc/stable/reference/generated/numpy.var.html. Accessed: 2020-09-03. NumPy, 2020.

[45] Nvidia. *Nvidia drive constellation*. [Accessed: 24-06-2020]. URL: https://www.nvidia.com/en-gb/self-driving-cars/drive-constellation/.

[46] R. Nystrom. *Game Programming Patterns*. 1st ed. Genever Benning, 2011.

[47] “Preliminary Report Highway HWY18MH010”. In: *National Transportation Safety Board* (2018).

[48] *PriorityQueue (Java SE 11 and JDK 11)*. https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/PriorityQueue.html. Accessed: 2021-04-15.

[49] *Releasing Your Project - Unreal Engine 4*. https://docs.unrealengine.com/en-US/Engine/Deployment/Releasing/. Accessed: 2020-011-13.

[50] rFpro. *Add traffic and pedestrians to your testing*. https://www.rfpro.com/driving-simulation/traffic-and-pedestrians/. Accessed: 2021-04-20.

[51] *RR lightweight tool for recording*. https://github.com/mozilla/rr. Accessed: 2020-011-13.

[52] Z. Saigol and A. Peters. “Verifying automated driving systems in simulation framework and challenges”. In: *25th ITS World Congress, Copenhagen* (2018).

[53] J. Schumann, P. Gupta, and Y. Lui. “Application of Neural Networks in High Assurance Systems: A Survey”. In: *Applications of Neural Networks in High Assurance Systems. Springer, Berlin, Heidelberg,* (2010), pp. 1–19.

[54] D. Sculley et al. “Hidden technical debt in machine learning systems”. In: *Advances in Neural Information Processing Systems* 2015-January (2015), pp. 2503–2511.

[55] R. Sedgewick and K. Wayne. *Algorithms, 4th Edition*. Addison-Wesley, 2011.

[56] 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.

[57] C. Staff. “To Catch a Failure: The Record-and-Replay Approach to Debugging”. In: *Commun. ACM* 63.8 (July 2020), pp. 34–40.

[58] *Stat Commands*. https://docs.unrealengine.com/en-US/Engine/Performance/StatCommands/. Accessed: 2020-011-13.

[59] 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. Virtual Event, USA: Association for Computing Machinery, 2020, pp. 337–348.

[60] *Substepping UE4*. https://docs.unrealengine.com/en-US/Engine/Physics/Substepping/. Accessed: 2020-011-13.

[61] *The Highway Code*. https://www.gov.uk/guidance/the-highway-code. Accessed: 2019-11-25. Department of Transport, UK, 2015.

[62] V. Timonen. *GPU burn*. [Accessed: 17-12-2019]. URL: https://github.com/wilicc/gpu-burn.

[63] *UK Road Traffic Act 1988*. http://www.legislation.gov.uk/ukpga/1988/52/contents. Accessed: 2019-10-03.

[64] S. Ulbrich, T. Menzel, A. Reschka, F. Schuldt, and M. Maurer. “Defining and Substantiating the Terms Scene, Situation, and Scenario for Automated Driving”. In:

IEEE Conference on Intelligent Transportation Systems, Proceedings, ITSC 2015-Oct (2015), pp. 982–988.

[65] Unity. *Order of Execution for Event Functions*. [Accessed: 24-06-2020]. URL: https://docs.unity3d.com/Manual/ExecutionOrder.html.

[66] *Unity Game Engine*. https://unity.com/. Accessed: 2020-011-13.

[67] *Unreal Engine 4*. https://www.unrealengine.com/. Accessed: 2020-011-13.

[68] *Vienna Convention on Road Traffic*. https://treaties.un.org. Accessed: 2019-10-03.

[69] *What is multithreading?* https://docs.unity3d.com/Manual/JobSystemMultithreading.html. Accessed: 2020-011-13.

[70] N. Whitehead and A. Fit-Florea. *Precision & Performance: Floating Point and IEEE 754 Compliance for NVIDIA GPUs*. https://developer.nvidia.com. 2011.

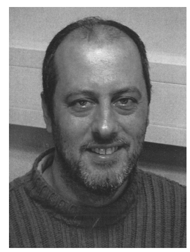
[71] *Windows Task Settings Priority property*. https://docs.microsoft.com/en-us/windows/win32/taskschd/tasksettings-priority. Accessed: 2020-011-13.

Greg Chance received the B.Sc. (Hons.) and Ph.D. degrees in physics from the University of Bath, Bath, U.K., in 2001 and 2005, respectively. He was previously at the Bristol Robotics Lab and is currently a Senior Research Associate with the Trustworthy Systems Laboratory, University of Bristol, Bristol, U.K. His current research interests are simulation-based verification for autonomous systems and cybersecurity. He has ten years industrial experience researching for Oxford Instruments, U.K., and BAE Systems, U.K. Dr Chance is a Chartered Engineer and a member of the Institute of Physics and holds a patent for a novel plasma control method.

Abanoub Ghobrial received the MEng degree in mechanical engineering from the University of Bristol, Bristol, U.K., in 2018. He is currently pursuing the Ph.D. degree in computer science at the University of Bristol and is a part-time Research Associate with the Trustworthy Systems Lab, Bristol, U.K. From 2018 to 2020, he was a full-time Research Associate with the Trustworthy Systems Lab. His current research interests are techniques to allow self-managing of autonomous safety-critical systems via continual learning during operation; and the development of simulation-based verification techniques for autonomous systems.

Kevin McAreavey received a BSc (Hons) in computer science from Queen's University Belfast (QUB) in 2010 followed by a PhD in AI in 2014. He is currently a Senior Research Associate with the Intelligent Systems Laboratory at the University of Bristol (UOB), working on explainable AI for AI-enabled domestic life. Previously he has worked with the Trustworthy Systems Laboratory at UOB and as a Research Fellow at QUB. His research interests are broadly in the areas of knowledge representation and reasoning, agents and multi-agent systems, and automated planning.

Séverin Lemaignan is Associate Professor in Social Robotics and AI at the Bristol Robotics Laboratory, University of the West of England, Bristol. He obtained a joint PhD in Cognitive Robotics from the CNRS/LAAS (France) and the Technical University of Munich (Germany) in 2012. He then joined the EPFL (Switzerland) and Plymouth University (UK) as post-doc, then lecturer in Robotics until 2018, when he joined the Bristol Robotics Lab. His research interest primarily concerns the socio-cognitive aspects of human-robot interaction and he has been focusing his recent experimental work on child-robot interaction and human-in-the-loop machine learning for social robots. Pr. Séverin Lemaignan has been involved in several European projects related to social and cognitive robotics (CHRIS, DREAM, L2TOR) and was awarded in 2015 a EU H2020 Marie Skłodowska-Curie Individual Fellowship for his project on robots and theory of mind.



Tony Pipe received his BSc (Hons.) degree in Electronic Engineering from Warwick in 1979 and, after an eight-year period in industry that included running a small company, received his PhD in Robotics from the University of the West of England (UWE) in 1997. Since 2010 he has been a Professor of Robotics and Autonomous Systems at UWE. He was a founding member of the Bristol Robotics Laboratory (BRL), and acted as one of its two Deputy Directors from its inception in 2006 until 2020. His research interests focus on safe and correct operation whilst in close-proximity to human beings. He has co-authored over 200 international refereed publications, supervised 35 PhDs, and created a BRL income of £10M since 2013. He has acted as BRL-UWE PI on the EPSRC grant 'Trustworthy Robotic Assistants', the Programme/Hub grants 'Robotics for Nuclear Environments' and 'National Centre for Nuclear Robotics', and the Centre for Doctoral Training in Robotics and Autonomous Systems. Prof. Pipe has been involved with Connected Autonomous Vehicle research and development since 2015 via the Innovate UK/CCAV funded projects VENTURER, FLOURISH, MultiCAV, CAPRI, ROBOPILOT and CAV-Forth.

Kerstin Eder is Professor of Computer Science and heads the Trustworthy Systems Laboratory at the University of Bristol. She also leads the Verification and Validation for Safety in Robots research theme at the Bristol Robotics Laboratory. She has gained extensive experience of verifying complex micro-electronic designs while working with leading semi-conductor design and Electronic Design Automation companies worldwide. Her research is focused on specification, verification and analysis techniques to verify or explore a system's behaviour in terms of functional correctness, safety, security, performance and energy efficiency. Prof. Eder's most recent contributions include agent-based testing, how to use assertions and theorem proving to verify control system designs, energy modeling of software and static analysis to predict energy consumption of programs. She holds a PhD in Computational Logic and an MSc in Artificial Intelligence from the University of Bristol, UK, as well as an MEng in Informatics (Dipl. Inf.) from the Technical University Dresden, Germany. In 2007 she was awarded an "Excellence in Engineering" prize from the Royal Academy of Engineering.