Automatic Fuzz Testing and Tuning Tools for Software Blueprints

Ciprian Paduraru[1,2], Rares Cristea[1,2] and Alin Stefanescu[1,2]
[1] *Department of Computer Science, University of Bucharest, Romania*
[2] *Research Institute of the University of Bucharest, Romania*

Keywords: Guided Fuzzing, No or Low-Code Paradigm, Blueprints, Simulation Applications, Digital Twins, Functional Tests.

Abstract: The increasingly popular no- or low-code paradigm is based on functional blocks connected on a graphical interface that is accessible to many stakeholders in an application. Areas such as machine learning, DevOps, digital twins, simulations, and video games use this technique to facilitate communication between stakeholders regarding the business logic. However, the testing methods for such interfaces that connect blocks of code through visual programming are not well studied. In this paper, we address this research gap by taking an example from a niche domain that nevertheless allows for full generalization to other types of applications. Our open-source tool and proposed methods are reusing existing software testing techniques, mainly those based on fuzzing methods, and show how they can be applied to test applications defined as visual interaction blocks. Specifically for simulation applications, but not limited to them, the automated fuzz testing processes can serve two main purposes: (a) automatically generate tests triggered by new stakeholder changes and (b) support tuning of different parameters with shorter processing times. We present a comprehensive motivation plan and high-level methods that could help industry reduce the cost of testing, designing, and tuning parameters, as well as a preliminary evaluation.

1 INTRODUCTION

Low-code or no-code application development is defined as a methodology for developing applications by connecting blocks of operations (implemented using classical programming methods) and data at a high level of granularity using a mostly graphical interface. The methodology is well documented by practitioners in industry [1] and in the software engineering literature (Luo et al., 2021), (Bucaioni et al., 2022), (Waszkowski, 2019). Its advantages can be considered from several points of view. First, it allows faster development of prototypes for new concepts and applications. This is an important aspect, as flexibility and speed in developing prototypes to full deployment are critical for companies. Multiple stakeholders can participate in product development, testing, and evaluation because the process of linking visual blocks is much easier to understand for people who have no knowledge of software engineering. Second, architecture and orchestration within an application can be better explained at a very granular level and in a graphical format. This leads to higher product quality and easier maintainability. Last but not least, low-level operations and legacy source code can be hidden, which can help focus more on the high-level processes rather than the details.

From our study, we concluded that there is currently a large gap in automated end-to-end testing of such components. In this regard, we take a step forward by focusing on the development of a framework capable of testing and optimizing the no-/low-level code approach for a specific domain: simulation applications. The applications that fall into this category are diverse: entertainment, video games, simulators such as car driving, airplanes, digital skills, training of various professions such as medicine, army, pilots, etc. These are usually implemented on a set of libraries known nowadays as *engines*, e.g., Unreal Engine[2], CryEngine[3], Unity[4]. The motivation to apply our proposed methods to this domain stems from the fact that no-/low-code methods have been

[1] https://powerapps.microsoft.com/en-us/low-code-no-code-development-platforms and https://www.ibm.com/cloud/blog/low-code-vs-no-code

[2] https://docs.unrealengine.com

[3] https://www.cryengine.com

[4] https://unity.com

Paduraru, C., Cristea, R. and Stefanescu, A.
Automatic Fuzz Testing and Tuning Tools for Software Blueprints.
DOI: 10.5220/0012121900003538
In *Proceedings of the 18th International Conference on Software Technologies (ICSOFT 2023)*, pages 151-162
ISBN: 978-989-758-665-1; ISSN: 2184-2833

used in this sector and its tools for years. The notion of a graphical representation that orchestrates inputs, data, operation blocks, and produces outputs is variously named *Blueprints*, *Schematics*, or *Visual Scripting* in today's commercial engines. In this paper we use the first term, i.e., *Blueprint*. As indicated by our surveys, their primary motivation was to encourage multi-stakeholder development, maintenance, and testing of applications. Another important motivation was to decouple the low-level parts of the source code from the high-level logic, so that the application as a whole would be much easier to understand, optimize, and evolve over the years.

Software Blueprints are an important part in the process of product development (Godoy et al., 2019), (Noll et al., 2016). Their use can be important in both forward and reverse engineering processes. Blueprints are used in popular applications such as simulation and game development environments (as mentioned earlier), software in web development such as the Ember framework[5], Flask and Python collaboration[6]. The purpose of their use varies from different perspectives. For example, in simulation app development, their role is to involve stakeholders other than programmers in the development of software products through a reusable library of implemented objects and functionalities. In the area of web development, they are typically used to generate code at runtime, i.e., blueprints function more as templates that are used as code generators. A well-known method for defining blueprints in software development methodology is the use of UML diagrams (Bergmayr et al., 2014), but more recently they can also be defined by API functions, as in the case of web development frameworks, or by visual scripting in simulation engines.

Contributions. To our knowledge, our open source framework, available at https://github.com/AGAPIA/EBLT, is the first to address automated testing and tuning of processes defined by graphically defined blueprints. Some new features are given below:

- A fully functional open source solution that can automatically create functional tests without requiring any human effort. Any other human help is optional and can only guide the fuzzer to get faster and more meaningful feedback. The tool is evaluated using examples created in one of the most popular engines on the market, the Unreal Engine. It was architecturally designed to be completely decoupled from individual projects.
- Identification of real industry problems, potential use cases for scenario testing and implementation to solve them with minimal developer effort. Considerations were made after discussions and collaboration with our industry partners.
- An abstraction of no-/low-code (blueprints) method definitions that can be extended to other application types, not just simulations.
- Support for tuning simulation applications (automatically determining parameters and simulation contexts subject to some given constraints) by test methods and agents. This would replace what tends to be human work with machine work.
- A link between blueprint-based testing methods and previous work in the general software testing literature, such as fuzzing methods, which form the core of our proposed solution. More specifically, we currently use whitebox fuzzing methods (Godefroid et al., 2012) along some various instructions and strategies to prioritize tests.

The paper is organised as follows. The next section shows how blueprints are currently used in the simulation software industry. Section 3 describes related work in the literature. Our proposed and implemented methods are presented in Section 4. A collection of preliminary evaluation results and discussions are provided in Section 5. Finally, conclusions and future work are constitute the last section.

2 BLUEPRINTS IN SIMULATION SOFTWARE DEVELOPMENT - AN INTRODUCTION

The basic idea of blueprints is to provide the same semantic capabilities as the concept of object-oriented programming in common programming languages, but in a graphical representation that can be understood and authored by various stakeholders in an application, not just software engineers.

In engines commonly used in industry today, blueprints are defined by a visual scripting tool that allows stakeholders to connect code and data blocks through a node-based interface. The data and code blocks can either be implemented in source code by software engineers and deployed as a reusable library for different projects, or created hierarchically in the visual tool by other stakeholders such as designers, quality assurance engineers, managers, etc. The typical workflow is that programmers implement a basic block component that can be further extended by other stakeholders

[5] https://cli.emberjs.com/release/advanced-use/blueprints

[6] https://flask.palletsprojects.com/en/2.0.x/tutorial/views

Since these simulation engines share strong similarities in the visual scripting aspects of the blueprints, from their graphical representation to their backend implementation, we have chosen Unreal Engine 5 for presentation purposes, since it is currently one of the most popular engines used in both industry and academia, and also has the advantage of being fully open source, which supports our current development and experimentation. Note that our proposed methods, architectures and source code abstractions are also suitable for the interfaces of the other simulation engines mentioned.

2.1 Authoring Blueprints

A blueprint, Fig. 1, 2, is similar to the concept of a class in object-oriented programming (OOP) in that it contains the following features:

- Variables (members) of any kind, either default or user-defined by users.
- Internal (possibly hidden) functionality, i.e., encapsulation.
- Communication with other blueprints or components of the system is possible as follows:
  - Interfaces for function calls.
  - Event-based systems, i.e. similar to the observer pattern (Zhang et al., 2017).
  - References between blueprints.
- Blueprints can be derived to create hierarchies as in OOP, register functions to serve as constructors and destructors, and they can define virtual functions.

One of the advantages of blueprints is that they can present details about a class or functionality in a graphical form that is easier even for software engineers to understand, not to mention that they allow various stakeholders without technical expertise to approach, build, and test these types of systems. As shown in the examples in Fig. 1, the connected white pins represent the connection flows (execution flows) between nodes. When a node is executed, the data inputs associated with it are retrieved, populated, and then sent to the node functionality (which can either be part of another blueprint, an internal blueprint, or source code defined elsewhere). In this way, blueprints can define complete logical flows that link functions and data for development processes. Simulation engine tools typically include tools to support debugging, similar to common programming language IDEs, definition browsing, reference and connection visualization.

2.2 Backend Support

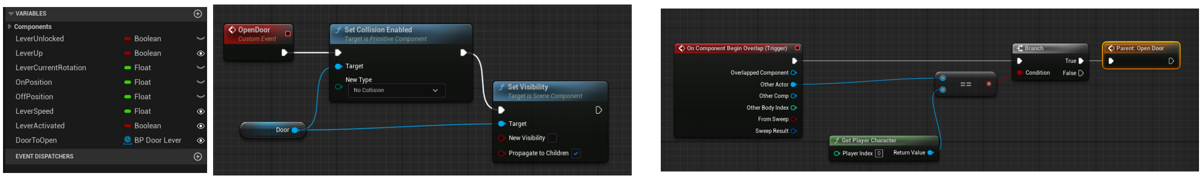
Blueprints typically have a base of objects that are provided in the form of a library of functionality. This library is reusable between projects, and many independent developers have created such objects for specific audiences (e.g., artificial intelligence, point-cloud visualization, motion capture components, architecture or film industries, etc.) and then uploaded them to a public marketplace repository[7]. These are implemented in common programming languages such as *C++*, *C#*, or *Python*. The idea is that by supporting reflection, developers can mark which of their implementations, objects, function calls, and events from the source code are publicly available to blueprints.

3 RELATED WORK

As noted in the recent survey presented in (Politowski et al., 2021), most simulation applications and game developers still rely on human efforts to test games by performing manual playtesting However, parts of the systems can be automated, as is the case in (Paduraru et al., 2022), (Paduraru et al., 2021a) which use artificial intelligence, computer vision, and behavior-driven development (BDD) (Irshad et al., 2021) to solve some of the problems in automating game testing. Modern simulation engines have built-in capabilities for creating functional tests at some level, either through code or by inserting nodes into blueprint definitions that link the sequence of actions to their expected outcomes. The literature has explored support in engines such as Unreal (Yang et al., 2019), or Unity (Pasternak et al., 2018). However, there is no support for automated blueprint testing without human effort, which is the novelty of our paper. Our plan is to reuse existing work on automated agents capable of exploring environments from 2D to 3D representation and support for guided fuzzing, and then apply these methods to our goal of automating blueprint testability with minimal user intervention. Below, we mention related work from the two areas that inspired some of our component development.

Automated Game Testing Agents. Agents for driving automated tests that simulate human behavior in simulation environments have been well studied in the literature. The methods used by authors vary widely, and each has been shown to be successful in niche games. For example, the work in (Ariyurek et al., 2021) and (Paduraru and Paduraru, 2019) uses techniques such as Monte Carlo Tree Search (MCTS)

[7] https://www.unrealengine.com/marketplace



(a) (b) (c)

Figure 1: Example of how blueprints are defined and used in the Unreal Engine 5 tool. For each node, the inputs are shown on the left and the outputs on the right. The white connections between the arrows show the connections of the nodes, i.e., the concrete function flow. The other types of edges represent reading/writing data at the inputs or outputs of the nodes. (a) Member variables definition inside a blueprint, containing names, variable type and visibility (public or private). (b) Example of an event listening and response in blueprints (c) blueprint functionality to respond to a collision event and trigger environmental changes - opening a door in this example.

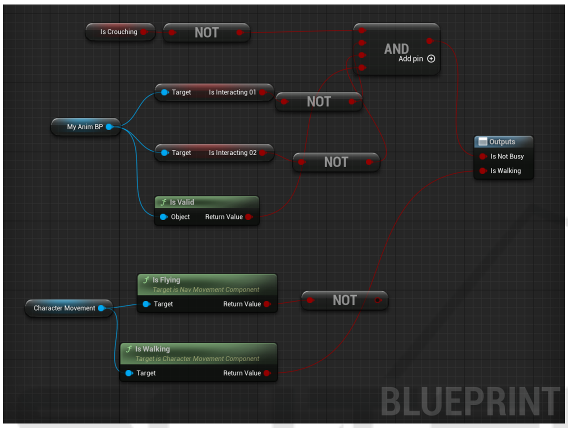


Figure 2: A blueprint example showing communication between many other blueprints, similar to object-oriented systems where references to other objects are coordinated to obtain different functionalities. In this specific case, the input comes from two different blueprints, i.e. an animation blueprint (*My AnimBlueprint*), a character movement component (*Character Movement*), and a member variable called *Is Crouching*. After some tests consisting of Boolean logic composed of the result of function calls (the green functions, *Is Valid*, *Is Flying*, *Is Walking*,...), and by reading some member variables defined in other Blueprints (*Is Interacting 01*, *Is Interacting 02*), a final result output is obtained, which decides for external systems whether the character is currently performing an action or not. The blueprint itself can be considered a high-level utility class/function.

and behavior trees to create logical agents capable of testing 2D or isometric types of games with limited ranges of motion and environments.

In recent publications, Deep Reinforcement Learning (DRL) is used to accelerate the results obtained by the test agents. For example, the work in (Zheng et al., 2019) uses DRL and evolutionary methods to optimize a set of strategies for debugging video games by training agents in two different perspectives: (a) winning the game, (b) exploring the states of the environment as much as possible. In this combination, the authors prove that for certain game genres, it is possible to obtain both state and code coverage detection. Using the same basic method, other works exploit the idea of training agents by using reward functions to punish their actions when they deviate too much from human behaviors, as shown in (Gudmundsson et al., 2018), (Tastan and Sukthankar, 2011), and (Glavin and Madden, 2015). A novelty in the same area, presented in (Bergdahl et al., 2020), (Gordillo et al., 2021), (Gisslén et al., 2021), extends the previous ideas by attempting to create a heatmap of the situations analyzed at each point in the testing process. Thus, the agent focuses on increasing not only code coverage, but also physical states in the 3D target environment. As an aside, Unity supports an out-of-the-box solution for creating agents based on DRL (Juliani et al., 2018). By using a modular architecture, users of our framework can switch between these implementations depending on their needs and available resources.

Guided Whitebox Fuzzing Techniques and Best Practices. The type of fuzzing supported in simulation engines includes the *whitebox testing* (Godefroid et al., 2012) category, as they can provide valuable information about the state covered by the inputs generated by the fuzzer. This is an important aspect to consider, as it allows the reuse of methods that allow orientation. Guided fuzzing has been studied in the literature for testing general software components and has proven successful in many applications. The methods used to guide the fuzzer can vary depending on the purpose of the test and available computational resources. For example, fast testing methods are based on the use of heuristics and genetic algorithms ((AFL,), (Paduraru et al., 2017)). Symbolic execution methods that use DRL techniques are also explored in (Böttinger et al., 2018), (Koo et al., 2019), (Paduraru et al., 2020), (Paduraru et al., 2021b). All these types of guidance methods can be reused in our framework as it uses a plugin-enabled architecture.

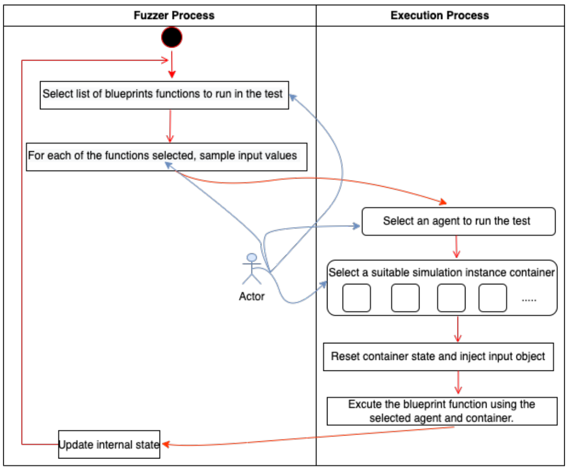

Fuzzer Process
Execution Process
Select list of blueprints functions to run in the test
For each of the functions selected, sample input values
Select an agent to run the test
Select a suitable simulation instance container
.....
Actor
Reset container state and inject input object
Excute the blueprint function using the selected agent and container.
Update internal state

Figure 3: Interaction of the two main components, Fuzzer and Executor to coordinate the automated testing processes within our framework. The blue arrows starting from the human actor represent the human-in-the-loop concept, i.e., activities where the human can play a role in defining different behaviors to guide the testing efforts. For example, humans can help in selecting the agent for test execution, configuring clusters in which the test process is executed, and by providing hints about the input variables and their types/ranges using our annotation support.

4 METHODS

The overall flow of the testing process is shown in Fig. 3. There are two main components involved in the architecture:

- Fuzzer: selects the functions and inputs for execution. Guidance methods are also included here.
- Executor: selects the agent to be used for execution with the input values suggested by the fuzzer, and then executes the simulation environment in one of the available clusters. The feedback from the testing process is sent back to the fuzzer so that it can properly guide the next steps. The next section describes the implementation in detail.

4.1 Abstractization of Blueprints

Since the layouts and definitions in the graphical format of blueprints (or visual scripting, as they are called in some of the commercial solutions used, such as Unity) may be different, we first create an abstraction of blueprints that acts as a proxy between the simulation engines and our plugin. This way, any user or custom simulation engine can reuse our solution.

Let us first consider a set of blueprints defined in the framework's shared library, N^{lib}, which can be immediately reused by any project. The set of blueprints that represent the specifics of a project is denoted by N^{bp}. Thus, the total set of available blueprints is $N = N^{lib} \cup N^{bp}$. Blueprints themselves may contain other blueprints that construct complex data flows, but in practice these hierarchies are manageable from the point of view of definition (creation) using the graphical editor provided. In this context, i.e., hypergraphs, we consider each blueprint as a node with internal data, input and output interfaces. Formally, each node $N_i \in N$ contains the following:

- Data: consisting of member variables, as in the example in Fig. 1a shown. The set of member variables is denoted as $Vars(N_i) = \{m_{i_0}, m_{i_1}, \dots\}$. Each variable has two properties:
  - $VarType(m_{i_j}) \in$ *Boolean, Int, Float, Vector, Matrix, Actor, Arrays, or other user defined type*
  - $Visibility(m_{i_j}) \in \{Private, Public\}$.
- Functionality: Each blueprint definition contains a set of functions $Func(N_i) = \{F_{i_0}, F_{i_1}, \dots, F_{|Func(N_i)-1|}\}$, which can be interpreted as structured flow subcomponents. Functions are represented using an operations graph, as shown in Fig. 1b, 1c, and 2. Their role is to specify the communication flows within a blueprint (node) and to connect data, operations, and nodes in an organized graphical format.

  It should be noted that stateless utility type functions can also be defined in the shared blocks library by software engineers and written in source code. The input pins $Inputs(F)$ and the output pins $Output(F)$ of a function are comparable to the arguments of a function in common programming languages. Examples of input pins are shown on the left side of the function *Set Collision Enabled* in Fig. 1b, while output pins can be seen in Fig. 1c, in the event node *On Component Begin Overlap*. Note that a blueprint function definition can have both input and output pins, cf. Fig. 2.

4.2 Testing and Tuning Blueprints With Fuzzing

The inputs of any function F in a blueprint N_i can be subjected to the component *Fuzzer*. As described in Section 4.3, each function F can optionally be given a schema for its inputs, $Inputs(F)$, which contains what is testable and what is not, the mocking components, value ranges, and other valuable information to better coordinate the fuzzing process. We denote by $T_{BP} = \{N_{t_0}, N_{t_1}, \dots\}$ the set of blueprints to be tested in a time step and by $T_{func}(N_{t_i})$ the set of desired functions to be tested within each blueprint. As mentioned earlier, each $F \in T_{func}(N_{t_i})$ can optionally contain a scheme defined as $Schema(F)$. After applying and evaluating fuzzing methods according to the latest software testing methods, the goal of blueprints testing could be defined as follows:

- Achieve the best possible coverage in terms of nodes and edges within a blueprint.
- Improve state value coverage, i.e., try to execute nodes with different sets of input values as many times as possible in the allowed test time window.

Listing 1: The blueprint fuzzing process pseudo code.

```
1 func RunTests()
2   $T_{Funcs}$ = Fuzzer.Sample()
3   for each F in $T_{Funcs}$ do
4     input = Fuzzer.GetInput(F, Schema(F))
5     ExecutionModule.InjectInput(input,F.ParentBP)
6     feedback = ExecutionModule.Run()
7     Fuzzer.UpdateState(feedback)
8   endfor
9 endfunc
```

The high-level pseudocode of a test step is shown in Listing. 1 in the *RunTests* function. Detailed strategies are discussed in more detail in Section 4.4. First, in line 2, the pseudocode collects the list of blueprints and functions to be tested with the current evaluation strategy. For each of these functions, the fuzzer component retrieves an input object for its input pins in line 4 using the internal strategy and the given schema (if any). This input object is applied to the simulation engine's memory context and then executed in lines 5-6. The feedback after the execution process contains which nodes were touched in the last execution step and against which new value ranges. This information is sent back to the fuzzer component to properly guide the testing efforts in the next time steps, line 7. The presented algorithm is parallelized in practice, as different sampled inputs can be executed on individual computers to obtain faster feedback. Note that, according to our evaluation, this is the most expensive process in testing simulation environments.

4.3 Data Schema and Mocking

The data types supported by our current implementation are user customizable. The $Object$ type is the base class for each of these types. The default types of the framework are: *Types* ∈ *Int, Float, Vector2D, Vector3D , Matrices, String, Object*. For each input of a testable function, the user can optionally specify a schema in a JSON file that contains annotations for data types to control the work of Fuzzer in an efficient way. The annotations for each variable of the base type, i.e. with the exception of the $Object$ type, can be specified as follows:

- A concrete set of values to be used, e.g., for a *Vector3D* type: $\{(X^0,Y^0,Z^0),(X^1,Y^1,Z^1),\ldots\}$, denoting potentially different valid spawning positions of a character in the environment (X,Y,Z) given in 3D space.
- A concrete set of value ranges to be used, e.g., for the same vector type: $\{([X_s^0,X_e^0],[Y_s^0,Y_e^0],[Z_s^0,Z_e^0]), ([X_s^1,X_e^1],[Y_s^1,Y_e^1],[Z_s^1,Z_e^1]),\ldots\}$, meaning valid 3D cube positions in the environment for a vector type.
- For string types only, a set of compatible regex patterns. E.g., an acceptance criterion for email addresses or usernames.

For generic *Object* types, the annotations can be specified as follows:

- A concrete set of objects $\{O_0,O_1,\ldots\}$ in the environment that can be used for this value. E.g., the variable is a placeholder that can be filled at test time with one of the specified vehicles in the world.
- A concrete set of classes $\{C_0,C_1,\ldots\}$ that can be created at test time to fill the concrete value of the variable. E.g., a vehicle class that can be filled in. With this method, the framework searches for all compatible objects of the compatible class along the hierarchy and selects one of them.

This solution also solves the **mocking** concept in the case of test blueprints, for example, by selecting a concrete set of mocking objects or values for functionalities that refer to external components independent of the test purpose.

4.4 Strategies for Testing

The first part of this section discusses how the fuzzer component selects the blueprints and functions to be evaluated in the next test run according to the goals set by the user, cf. Listing 1. The second part shows how the actual input object to be injected into the execution module is sampled.

Blueprints and Functions Prioritization. For the selection of blueprints to be tested within a test run, as shown in Listing 1 line 2, the framework considers two possible strategies One is the manual one, where the list of blueprint functions to be tested and their priorities are proposed: $T_{func} = \{(F_{t_0},p_{t_0}),(F_{t_1},p_{t_1}),\ldots(F_{any},p_{any})\}$. In this setup, F_{any} plays the role of a wildcard, i.e., the algorithm has the possibility to select any blueprint function other than the concretely specified one for testing with the specified probability. This type of suggested selection can be used for many common use cases in practice, e.g., when performing smoke tests for some components or cluster tests for different parts of an implementation. It is also possible to make an automatic selection if no list is specified. By default, the framework selects the most recently added or changed blueprint features between test runs with higher priority. With this setup, a process observer regularly analyzes the changes to the blueprint files via the source code repository.

Guided Input Generation. The first trivial option is to randomly generate the input to a function according to its schema, if the user has specified one. However, to achieve coverage of blueprints nodes, we propose an option that suggests to the fuzzer a prioritized list of nodes to be reached in the test process, either internal or output nodes, L_{F_i} for each of the functions. In practice, this could represent the effort of the human in the loop directing the test to low-level areas that need more testing than others, are prone to bugs, etc. The list is optional, but will be prioritized if specified. To solve the prioritization of node execution, a modified breadth-first search algorithm is used to create an inverted list of input pins associated with each target node. Thus, with the set L_{F_i} of priority nodes, the fuzzer knows the list of input pins in the function F_i that are connected to them and could affect their execution $Inputs_{selected}(F_i) \in Inputs_{F_i}$. This semi-random strategy can be further improved by symbolic execution, either online or offline, as suggested in (Godefroid et al., 2012), (Paduraru et al., 2021b), or (Böttinger et al., 2018). This is possible because after the *ExecutionModule* evaluates the simulation against a certain input, a path of nodes can be retrieved as feedback. Since this path may contain certain branch points, it would be wiser to use the symbolic solution for the other branches that were not seen before. We have saved this idea for the next versions of our tool.

Input Values - Sampling Strategies. For sampling concrete input values to be filled in at test time, the user-specified annotation may additionally contain a priority for each value in the set, regardless of the type used, i.e., this is suitable for concrete values, ranges, classes of objects, or concrete objects. In practice, this priority could mean that a particular concrete value or range is more likely to reveal errors. Common examples in simulation engines are parts of the map or new classes of entities that have recently been added. At runtime, priorities are normalized, meaning that each value has a final priority between $0-1$ and the sum of all values is 1. By default, if not specified, all values have the same probability. Listing 2 shows the underlying sampling mechanism used by the fuzzer component to obtain the values of the variables used in Listing 1, line 4. As shown in lines 3 and 9 for both cases, i.e., generic objects and base types, the sample values are extracted from the given set of concrete values according to their priorities (function *ProbRandom*). In the case of an object where a set of classes is specified (line 6), a compatible class is selected and then created at runtime using the engine's internal functionality. Similarly, for base type objects (line 12): If a range of ranges is specified, the algorithm first takes a probability sample from one of the ranges and finally a uniform sample within that range.

Listing 2: Algorithm used by the fuzzer to sample concrete values according to the rules of annotation.

```
 1 func GetValue(var : type)
 2   if var.Type is Object:
 3     if var.annot.isConcrete:
 4       return ProbRandom(var.annot.set)
 5     else:
 6       C=ProbRandom(var.annot.set)
 7       return Engine.Create(C)
 8   else: // a base type
 9     if var.annot.isConcrete:
10       return ProbRandom(var.annot.set)
11     else:
12       range=ProbRandom(var.annot.set)
13       return RandInRange(range)
14   endif
15 endfunc
```

Sampling Notes for Tuning Process and Value Samples. While testing may uncover problems that can be attributed to various causes, the situation may be different when performing tuning processes. Usually, the designer knows exactly the parameters to be tested, the target and their ranges. Therefore, in practice, a different annotation should be used between testing and tuning processes.

4.5 Functional Tests, Expectations, and Behavior Driven Methodology

To define functional tests and their expectations as generic and reusable as possible at this level, the proposed framework uses the behavior-driven development testing (BDD) methodology (Irshad et al., 2021). The framework defines a base class called *TemplateTestDef* that can be overridden either in source code or in blueprints. The base class is used to control the message flow between the $Fuzzer\,process$ component and this test, and to enforce the definitions of the three main steps of a BDD test:

- Set the context of the test execution, i.e., the *Given* clause. For example, a test could load a specific level, create a character, and launch an agent to perform a specific action on that level.
- Specify the trigger when the test should be executed, i.e. the $When$ clause. An example could be that the character enters a certain area.
- Set the expectations for the test, i.e., the $Then$ clause. Continuing the previous example, one type of expectation might be to trigger a doorway.

These three clauses can also support logical operations, persistence, and state changes, and can be defined in both the source code and the blueprint. On a technical level, the framework provides all environment inputs and states to the user as dictionaries, allowing them to decide what to use in their desired test context. Practically, this means that a person with no programming knowledge can use blueprints to run functional tests from start to finish. A test blueprint could be used as a template for testing against a data-driven set of inputs and expected outputs defined in a data table. More concrete examples and already implemented components on this topic can be found in (Paduraru et al., 2022).

Persistent Functional Testing. An important feature of simulation applications is that when functional testing methods are used, most tests must be run for multiple time steps before a decision about correctness can be made (e.g., a path finding process leading from point A to point B in the environment). To support these specifics, the proposed framework adds the following support to the annotations:

1. Indication of which output variables can be continuously tested at regular intervals to provide early detection of any success or failure.
2. Events listened, which can support the correctness of states checking according to an observer pattern method (Piveta and Zancanella, 2003).

An example of the definitions of inputs and outputs and their annotations can be seen in Listing 3. The *inputs* field describes the annotation support for the variables present on the input interface of the blueprint, while *expectedOutputs* is used to define the expectation conditions and/or parameter ranges for the given inputs. The types of the variables are taken from the graphical definition of the blueprint or from their source code definition. The values of the variables can be specified either as a series of discrete values separated by the # character (lines 4, 8), or as a range (lines 3, 6). In the case of Vector3D representations, as in the case of the line 6, the range represents a 3D box in which the character could be positioned at the time of its instantiation. Entities can be defined by their names, as shown in line 7, where the target of the character is randomly chosen between two options. The output variables are used to indicate the result of the test, either successful or failed, along with a diagnostic code. We distinguish between the two options by using some markers. For example, the user can specify either a *fail* code (line 15), which in this case specifies the expected time interval that the character will take to get to the destination, or a *succeed* code (line 24), which in this case specifies an event that means that the operation will be performed in a different way and the test will be considered successful). Also, output variables can be tested only at the end of the test or continuously during its runtime, e.g., line 13. As shown, this variable is sampled and tested on every 60 frame to avoid unnecessary overhead.

Listing 3: An example annotation included in a JSON file for the pathfinding blueprint logic of a character named *PathfindingTestBP*.

```
 1 "PathfindingTestBP_annotations": {
 2   "inputs" : {
 3     "CharacterScale": "[min=0.8, max=1.2]",
 4     "sprintSpeed":"{100 # 200 # 300}",
 5     "StartLocation": "{min=(-23.0,-19.0,53.14),
 6              max=(-17.2,-18.2,41.5)}",
 7      "DestLocation_asEntity": "{P2 # P3}",
 8      "StartRotation" : "{(0,-90,0) # (0,0,0)}"
 9    },
10
11   "expectedOutputs" : {
12      "timeToPathLimit" : {
13           "type" : "continuous-60",
14           "value" : "[min=0.0, max=10]",
15           "failCode" : "1"
16        },
17      "movingIdleLimit" : {
18            "type" : "continuous-10",
19            "value" : "[min=0.0, max=3]",
20            "failCode" : "2"
21        },
22       "characterTeleported":  {
23            "type: event",
24            "successCode:"0"}
25 }}
```

4.6 Tuning Support

Guided fuzzing processes can be used to take a sample of inputs from a space of parameters and then compute a fitness score or penalty with respect to various outputs expected after evaluating the input. As a concrete real-world example, a fuzzer in a video game might choose between different abilities of a character to classify it into certain difficulty levels. The use of a fuzzer is desirable in several ways. First, it allows finding different behaviors (i.e., a set of parameters) of a character that lead to the same desired result with respect to the computed metrics without human effort. Second, it allows machines to find these behaviors themselves, sometimes exposing potential vulnerabilities that were not planned or conceived. Listing 4 shows a concrete example of a tuning annotation from our repository. The user is expected to write the input parameters and the results to be evaluated in the space. The idea behind the fitness/penalization function in the default provided support is to reach an ideal value and penalize deviations from it according to some rules. In our current implementation, a linear 9 and a Gaussian penalty 15 are provided. However, the user can extend the system and write their own fitness functions.

Listing 4: An example of an annotation specified in a JSON file to automatically find parameters for an intermediate difficulty class in a video game. In this example the medium difficulty character class is expected to beat the average human user in a given time interval and with a limited amount of damage.

```
"EBLTTest_Ex1.RPGTemplate_AI_mediumDiff": {
   "inputs" : {
      "HitRate": "[min=0.5, max=2]",
      "potions": "[min=0, max=3]",
      "jumpSpaces": "[min=0, max=3]"
    },
    "expectedOutputs" : {
       "timeToBeatEasyAI" : {
       "type" : "tuning-linear",
       "value" : "[ideal=180, std=600]",
       "penalization": "[min=0.0,  max=1.0]"
       },
   }
    "damageTaken" : {
        "type" : "tuning-gaussian",
        "value" : "[ideal=20, std=50]",
        "penalization" : "[min=0.0, max=1.0]"
 }}
```

4.7 Discussion About Testing Agents

As discussed in Section 3, there are several types of agents for testing simulation environments addressing various use cases such as closed spaces or open-world simulation environments, each with their strengths and drawbacks in different cases. In addition, it is common in this domain to test applications against scripted AI behaviors intended for specific tasks. To address these requirements, the framework's architecture completely separates the agents that control the testing and tuning processes from the processes themselves (Fig. 3). This provides a pluginable architecture so that the framework provide transparent interfaces for communicating with the agents deployed by the user. This also helps from the point of view that agents designed for different purposes (e.g., those that test more normal user behavior, others that test abnormal users or exploration of large space environments) can be used in different execution contexts using the same source code and blueprint definitions.

5 EVALUATION

The framework presented in this paper, cf. our open source repository, has been tested in the Unreal Engine 5 engine with several open source available demo projects: an educational tutorial project (*Hour of Code*), a city traffic simulator (*City Sample*), and an RPG game template. The diversity of the projects shows that the current implementation works as intended, i.e., as a plugin that is not associated with a concrete implementation. Because of the abstraction and interfaces provided, the core of the implementation can be reused for other simulation engines or applications developed with no/low code paradigms. Most work on testing such tools deals with quantitative evaluation by having the implemented test agents act in the environments to find hidden bugs. The metrics used are usually the source code, the environment and the state coverage. This type of evaluation is not very useful in our case, and we can only say that our methods can fully reuse this type of agents, as mentioned in Section 4.7.

Expectations. Ideally, the evaluation process of the proposed framework should consider if and how it can save the cost and increase the testability of the projects with less human supervision or effort. As described in this section, we found that:

- The testability of software projects can be significantly increased by several factors. First, more tests can be defined by promoting simplicity of definitions and by involving more stakeholders (with or without software engineering skills) in creating the tests. Second, more testable content is automatically added to a project's source code.
- Costs savings are expected from several perspectives. First, new or changed features are automatically discovered, prioritized and tested. Users can optionally contribute by providing only schemas or guided high-level prioritization of blueprints to be tested to support the effort and get faster feedback. But overall, the testing of blueprints functionalities is automatic. Second, it should be noted that source code written by software engineers can be made available for testing only by adding a node that calls the part of the code to be tested in a blueprint. This is an important point to consider as non-technical people can add testability to projects and guide them. Third, tuning various behaviors and parameters in simulation applications takes a lot of development time. As shown in Section 4.6, the proposed automated methods can replace human labor with machine efforts.

5.1 Usefulness of the Tool in Relation to Typical Problems

To outline the usefulness of the proposed framework, we follow (Zheng et al., 2019)'s classification and distribution of observed problems in game development (from our point of view, this is transferable to simulation applications in general). E.g., of issues that are described in the author's work: crashes, getting stuck in certain states, logical issues, game/simulation balance issues, load/stress testing.

5.2 Qualitative Evaluation

We tested our prototype tool for 9 months (at the time of writing our paper) at one of our industry partners with a group consisting of: 4 software engineers, 3 game designers, 7 animators, 5 technical artists, and 2 producers (management group). The group of software engineers developed new features to connect different parts of the engine or game logic. They were also very interested in using the functional tests for stress/load testing. The other stakeholders either wrote functional tests or provided general support in evaluating the quality of their own work or the project in general, i.e., animations, rendered art, stability, or generated game balance/exploits. The initial set of blueprints within the prototype was 945 divided among several categories (see Fig. 1a). Of these, the group supported annotations for 75% of the blueprints, with a detailed data schema defined for 84 deemed important to the project. By using the tool we proposed, the users did not have to perform any additional complex manual work. The three important research question we wanted to evaluate from a qualitative evaluation is given below.

RQ1: Does The Testing Tool Help Increase the Percentage of Testable Content in a Project? To answer this question, we evaluated the percentage of newly introduced features that contained tests after their submission in the 9 months of using the framework. This percentage was compared to the previous 9 months before the introduction of the proposed tool. Table 1 shows that the percentage of components tested increased significantly, especially for features introduced by stakeholders other than software engineers (e.g., animations, UI art, sound). The table also shows that in these same areas, there was a great interest in providing user guidance through our annotation support for features that have concrete input/output ranges of values or context. It is also worth noting that software engineers and vendors implemented significantly more load/stress testing and auto-tuning for the project as a result of our support (more visual details in our supplemental material). We attribute the demonstrated improvements mainly to the fact that by using our framework, more stakeholders had the opportunity to assess the quality of their own work and the project as a whole.

RQ2: Does the Proposed Testing Tool Help a Project Achieve Higher Code Coverage (Branch Coverage)? While the first study showed a comparative test at the feature level, we also want to see the delta in code coverage, i.e., how much the percentage of code coverage of newly introduced code for feature implementations changed in the analyzed time windows before and after the tool was introduced. As Table 2 shows, there is a similar trend as above, which was to be expected since blueprints connect blocks that are actually implemented in the source code. Note that before the introduction of tool support, tests were mainly written by software engineers using classical unit or functional tests. The improvements observed this time are not only due to the fact that more stakeholders are supported in writing tests, but also because the tests can be automatically created by the tool even if there are no annotations. Thus, the blocks become testable by simply connecting them in the Blueprint Designer.

Table 1: The first two columns contain the percentage of testable new features introduced in a 9-month period before and after the introduction of our test tool. The third column contains the percentage of tests that have annotations. The breakdown is by project area, but an overall view of the entire project can also be found in the last row.

Area	Before	After	Annotations
	proposed tool		percent
Gameplay	58%	79%	80%
Animations	41%	84%	11%
Physics	78%	83%	18%
Rendering	46%	51%	80%
Sound	39%	82%	90%
UI/art	29%	75%	100%
Load/Stress, Performance, Tuning	61%	90%	100%
Overall	59%	78%	75%

Table 2: Code coverage as a percentage of newly introduced source code in the 9-month analysis periods before and after the introduction of the test tool. Previous methods used unit and functional testing with existing tools available in their own game engines.

Area	Before	After
Overall	39%	53%

RQ3: What is the Subjective Opinion of the People Involved in the Experiment (User Study)?To evaluate this goal, we asked each person involved in testing the tool for feedback on what they think about the no-/low-code paradigm in general and how our tool can help in testing a project that uses it. Due to space constraints, we are including the responses we received in our repository. To summarize the conclusion of everyone involved in the experiments: A visual representa-

tion of the architecture and logic can greatly facilitate the testing process. The input/output variables and the connections between components become more visible, while the proposed annotation and testing support can help to identify problems faster and facilitate the maintenance of the project between cycles.

5.3 Discussion on the Efficiency of Using Existing Work For Fuzzing and Symbolic Execution Methods

As mentioned in Section 4.4, symbolic and concolic execution can be used in conjunction with the proposed tools by incorporating an external agent into the architecture shown in Fig. 3. This is another avenue that we will explore further in our future work. In an initial assessment of the work in progress, we found that the use of unprocessed source code methods, as used in previous work, is generally impractical for simulation software, except in short and highly targeted test scenarios.

A concrete example might be a common pathfinding case in which a character C must find and follow a path between a point A and B in the environment. The character's collision capsule $CCP(C)$ would be tested against nearby meshes in the environment located between A and B at each frame. At any time, the mesh sizes of the environment or of $CCP(C)$ can be changed by a symbolic execution method to take different execution paths, either at runtime of the path or at the execution of the move itself. It would be an impractical waste of resources and feedback to complete tests in a short time while covering only a minimal number of states or code. This is one of the reasons why in this work we have presented and implemented the standard version of the agent that tests the environment based on guided fuzzing methods. However, for future work, we have ideas for distinguishing the space in which symbolic execution should be used and for freezing some states of the environment.

Annotations can be used to drive the fuzzing processes without having to include the BDD or functional test templates. In general, the fuzzing processes cannot be used directly to specify expected outputs. However, expected value ranges can be defined in many situations. For example, consider the common industry requirement that the average number of frames per second should not fall below 30 fps, no matter what values the fuzzer produces. It is also clear from our experiments that a suitable input instruction can significantly reduce the computational cost of achieving the required coverage in a given time.

6 CONCLUSION AND FUTURE WORK

This paper presents a method for testing software that builds on the no-/low-code paradigm and focuses on simulation engines as a concrete implementation and testing platform. While different application domains and underlying components can play an important role in testing the visually connected blocks (blueprints) in the targeted paradigm, we believe that our abstraction can at least help generalize to other types of applications as well. Through the use of our framework and implemented tools, we have shown how most of the common testing and tuning use cases in the industry can be addressed. In addition, we plan to explore other methods to practically integrate existing work in the literature on symbolic/concolic execution into our methods. Possible ideas on this point include freezing states and discretizing time steps when invoking the symbolic solver. We plan to work closely with industry partners to evaluate the methods in depth to determine the importance of long-term use of blueprints on different types of projects. We also imagine with

ACKNOWLEDGEMENTS

This research was supported by European Union's Horizon Europe research and innovation programme under grant agreement no. 101070455, project DYNABIC. We would also like to thank our game development industry partners from Amber, Ubisoft, and Electronic Arts for their support.

REFERENCES

AFL. http://lcamtuf.coredump.cx/afl.

Ariyurek, S., Betin-Can, A., and Surer, E. (2021). Automated video game testing using synthetic and human-like agents. *IEEE Trans on Games*, 13(1):50–67.

Bergdahl, J., Gordillo, C., Tollmar, K., and Gisslén, L. (2020). Augmenting automated game testing with deep reinforcement learning. In *Proc. of CoG'20*, pages 600–603. IEEE.

Bergmayr, A., Troya Castilla, J., Neubauer, P., Wimmer, M., and Kappel, G. (2014). Uml-based cloud application modeling with libraries, profiles, and templates. In *CloudMDE workshop at 17th International Conference on Model Driven Engineering Languages and Systems, p56-65.*

Böttinger, K., Godefroid, P., and Singh, R. (2018). Deep reinforcement fuzzing. In *SP'18 Workshops*, pages 116–122. IEEE.

Bucaioni, A., Cicchetti, A., and Ciccozzi, F. (2022). Modelling in low-code development: a multi-vocal systematic review. *Software and Systems Modeling*, 21.

Gisslén, L., Eakins, A., Gordillo, C., Bergdahl, J., and Tollmar, K. (2021). Adversarial reinforcement learning for procedural content generation. In *Proc. of CoG'21*, pages 1–8. IEEE.

Glavin, F. G. and Madden, M. G. (2015). Adaptive shooting for bots in first person shooter games using reinforcement learning. *IEEE Trans. Comput. Intell. AI Games*, 7(2):180–192.

Godefroid, P., Levin, M. Y., and Molnar, D. (2012). SAGE: Whitebox fuzzing for security testing. *Queue*, 10(1):20:20–20:27.

Godoy, C. P., Cruz, A. F., Silva, E. P., Santos, L. M., Zerbini, R. S., and Pahins, C. A. L. (2019). Blueprint model: A new approach to scrum agile methodology. In *14th International Conference on Global Software Engineering (ICGSE)*, pages 95–99.

Gordillo, C., Bergdahl, J., Tollmar, K., and Gisslén, L. (2021). Improving playtesting coverage via curiosity driven reinforcement learning agents. In *Proc. of CoG'21*, pages 1–8. IEEE.

Gudmundsson, S. F. et al. (2018). Human-like playtesting with deep learning. In *Proc. of CIG'18*, pages 1–8. IEEE.

Irshad, M., Britto, R., and Petersen, K. (2021). Adapting behavior driven development (bdd) for large-scale software systems. *Journal of Systems and Software*, 177:110944.

Juliani, A., Berges, V.-P., Teng, E., Cohen, A., Harper, J., Elion, C., Goy, C., Gao, Y., Henry, H., Mattar, M., and Lange, D. (2018). Unity: A general platform for intelligent agents.

Koo, J., Saumya, C., Kulkarni, M., and Bagchi, S. (2019). PySE: Automatic worst-case test generation by reinforcement learning. In *ICST'19*, pages 136–147. IEEE.

Luo, Y., Liang, P., Wang, C., Shahin, M., and Zhan, J. (2021). Characteristics and challenges of low-code development: The practitioners' perspective. In *Proceedings of the 15th ACM International Symposium on Empirical Software Engineering and Measurement (ESEM)*.

Noll, J., Beecham, S., Richardson, I., and Canna, C. N. (2016). A global teaming model for global software development governance: A case study. In *11th International Conference on Global Software Engineering (ICGSE)*, pages 179–188.

Paduraru, C., Melemciuc, M., and Stefanescu, A. (2017). A distributed implementation using Apache Spark of a genetic algorithm applied to test data generation. In *GECCO'17 Workshops*, pages 1857–1863. ACM.

Paduraru, C. and Paduraru, M. (2019). Automatic difficulty management and testing in games using a framework based on behavior trees and genetic algorithms. In Pang, J. and Sun, J., editors, *24th International Conference on Engineering of Complex Computer Systems*, pages 170–179. IEEE.

Paduraru, C., Paduraru, M., and Stefanescu, A. (2020). Optimizing decision making in concolic execution using reinforcement learning. In *ICST'20 Workshops*, pages 52–61. IEEE.

Paduraru, C., Paduraru, M., and Stefanescu, A. (2021a). Automated game testing using computer vision methods. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*, pages 65–72.

Paduraru, C., Paduraru, M., and Stefanescu, A. (2021b). RiverFuzzRL - an open-source tool to experiment with reinforcement learning for fuzzing. In *14th IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 430–435.

Paduraru, C., Paduraru, M., and Stefanescu, A. (2022). Rivergame - a game testing tool using artificial intelligence. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 422–432.

Pasternak, M., Kahani, N., Bagherzadeh, M., Dingel, J., and Cordy, J. R. (2018). Simgen: A tool for generating simulations and visualizations of embedded systems on the unity game engine. In *ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, MODELS '18, page 42–46.

Piveta, E. K. and Zancanella, L. C. (2003). Observer pattern using aspect-oriented programming. *Scientific Literature Digital Library*.

Politowski, C., Petrillo, F., and Guéhéneuc, Y.-G. (2021). A survey of video game testing. In *2021 IEEE/ACM International Conference on Automation of Software Test (AST)*, pages 90–99.

Tastan, B. and Sukthankar, G. (2011). Learning policies for first person shooter games using inverse reinforcement learning. In *Proc. of AIIDE'11*, pages 85—90. AAAI.

Waszkowski, R. (2019). Low-code platform for automating business processes in manufacturing. *IFAC-PapersOnLine*, 52(10):376–381. IMS 2019.

Yang, X., Zou, D., Pei, L., Sartori, D., and Yu, W. (2019). An efficient simulation platform for testing and validating autonomous navigation algorithms for multirotor uavs based on unreal engine. In *China Satellite Navigation Conference*, pages 527–539. Springer.

Zhang, H., Li, W., Ding, H., Yi, C., and Wan, X. (2017). Observer-pattern modeling and nonlinear modal analysis of two-stage boost inverter. *IEEE Transactions on Power Electronics*, 33(8):6822–6836.

Zheng, Y., Xie, X., Su, T., Ma, L., Hao, J., Meng, Z., Liu, Y., Shen, R., Chen, Y., and Fan, C. (2019). Wuji: Automatic online combat game testing using evolutionary deep reinforcement learning. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 772–784.