SimGen: A Tool for Generating Simulations and Visualizations of Embedded Systems on the Unity Game Engine

Michal Pasternak
Queen's University
Kingston, Ontario, Canada
9mzp@queensu.ca

Nafiseh Kahani
Queen's University
Kingston, Ontario, Canada
kahani@cs.queensu.ca

Mojtaba Bagherzadeh
Queen's University
Kingston, Ontario, Canada
mojtaba@cs.queensu.ca

Juergen Dingel
Queen's University
Kingston, Ontario, Canada
dingel@cs.queensu.ca

James R. Cordy
Queen's University
Kingston, Ontario, Canada
cordy@cs.queensu.ca

ABSTRACT

This paper provides an overview of SimGen, a prototyping tool which aids in the creation of 3D simulations for embedded systems testing. SimGen relies on a domain specific language to describe the components of the simulation and the communication protocol used to interface with them. A game engine called Unity is used to power the 3D environment in which the simulation takes place. The prototyping tool generates the necessary scripts to fully build the simulation within the Unity environment. By using the power of the Unity environment with the simplicity of the domain specific language, users are able to create simulations customized to their testing needs. The communication protocol between individual objects and the users application is defined by the user in the DSL and runs on TCP sockets for a general and easy to implement connection.

Demonstration video: https://youtu.be/4ROt2N6i6KA

Download link: https://github.com/PasternakMichal/SimGen.git

CCS CONCEPTS

• **Computing methodologies → Simulation support systems**; Simulation by animation; • **Computer systems organization →** *Embedded systems*;

KEYWORDS

MDE, Simulation, Unity, DSL

ACM Reference format:
Michal Pasternak, Nafiseh Kahani, Mojtaba Bagherzadeh, Juergen Dingel, and James R. Cordy. 2018. SimGen: A Tool for Generating Simulations and Visualizations of Embedded Systems on the Unity Game Engine. In *Proceedings of ACM/IEEE 21th International Conference on Model Driven Engineering Languages and Systems, Copenhagen, Denmark, October 14–19, 2018 (MODELS '18 Companion),* 5 pages.
https://doi.org/10.1145/3270112.3270135

1 INTRODUCTION

Simulation and visualization are some of the main methods for testing and validation in the context of development of Real-time Embedded Systems (RTE), especially when testing with real hardware is costly or dangerous [1, 2]. A variety of simulators currently exist. They can be classified as custom designed specific simulators, and general use simulation systems which cover a broad list of domains (such as Simics [3] and Modelica based tools [4]). Applying simulation for certain tasks, especially for academic purposes is challenging due to: (1) many existing simulators are commercial, closed source, and expensive (2) learning to use and configure the simulator for a specific use case is a cumbersome and time-consuming task.

Unity is one of the most popular game engines used today. It provides an integrated development environment in which users can create their game by editing the scene with the mouse, or through C# scripts. Unity applications have been downloaded onto over 3 billion devices worldwide and applications made with the Unity engine have been downloaded over 24 billion times in the last 12 months [5]. It contains a full 3D physics engine, and the ability to easily import and create 3D models. This makes it a good fit for developing a simulation environment, and it recently has gained attention among users who develop simulations and other non-game related applications [6][7]. The use of Unity allows for reusing previously made content, and modifying game resources for the purpose of simulation.

In this work, Model Driven Engineering (MDE) techniques are used to create a tool which generates scripts for a framework in Unity for the purpose of creating custom simulations. The tool, called SimGen is composed of a DSL, code generator and the Unity framework. Using SimGen game designers and RTE system developers can work together to buildup a library of resources that can be used in simulations. RTE developers can then customize a final simulator by specifying the objects and setting their configurations within the DSL. This paper demonstrates how SimGen can be applied to use existing resources to create a simulation for testing control logic.

The remainder of the paper is organized as follows: Section 2 provides an overview of related work. Section 3 provides an overview of the SimGen tool. Section 4 shows how the tool can be

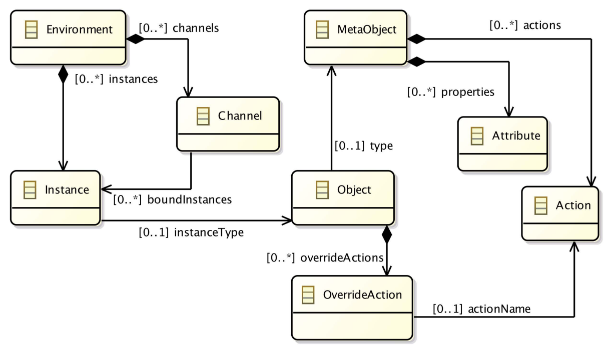

Figure 1: Abstract Syntax of the DSL

used to test a rover control script. Section 5 shows the integration of traditional MDE tools with the simulation environment.

2 RELATED WORK

Simulation and visualization has slowly begun to be explored as a useful tool when using MDE techniques. For example, the authors in [8] found that the use of 3D environments is useful to visualize the sensor layout on self driving vehicles they were performing studies on. MDE approaches to real design problems include some level of simulation, as described in [9] during development of onboard air traffic control systems.

Simulation has so far been focused on visualization of model execution without simulating or visualizing the physical system within a 3D environment. This is evident in tools such as Moka [10] for Papyrus or the model simulator for the Topcased toolkit [11]. Many tools for testing embedded systems do so without any 3D visualization, such as the UPPAAL-TRON tool used in [12]. Our work is different in that it simulates the physical embedded system entirely, and not only the control code or model.

Simulation of embedded systems such as vehicles is common and many simulators exist as shown in the survey [13], however most of these do not allow for easy out-of-the-box integration with any MDE tool. Commercial software with links to MDE tools is often specific to one industry, one such example is the preScan tool for the automobile industry [14]. Specific MDE simulation tools which have 3D environments such as MapleSim, Dymola and Simulink 3D provide a simulation environment which lacks the refinement and visual aesthetics of a commercial game engine such as Unity. The work presented in [15] on run time monitoring of a rover proposes visualization and simulation of the hardware in 3D as future work. Our work provides a tool for easy generation of a simulator which could fulfill this need. The benefit for custom simulators tailored to the users needs is made clear in the works described in [12], [9] and [16] among others. Visualization as part of simulation is an important aspect as described in [17], however it has for the most part been ignored by the MDE community.

3 SIMGEN OVERVIEW

The SimGen tool consists of two major components which work together to produce a 3D simulator. The tool consists of a framework within the simulation environment (Unity) and a DSL to configure what gets included in the framework. As shown in Figure 2, the user begins by using the DSL to specify which 3D objects they would like to include in the simulation, as well as the actions that are taken by these objects after receiving a specific message. A code generator translates this configuration to generate Unity scripts which control the simulation, as well as documentation describing the communication protocol. These scripts are then added to the Unity framework, and upon compilation create the desired simulator. The remainder of this section details the components of SimGen.

3.1 Unity Framework

The Unity framework component of the prototyping tool primarily consist of two parts. The first is a resource folder which contains a collection of 3D objects with scripts and various Unity specific physics components attached to them. These objects are known as 'prefabs'. All the resources are accessible to the framework, as such any referenced prefab must be present within the folder.

The second part of the framework is referred to in Unity terms as the 'scene'. The scene is what is initially loaded when the compiled simulation is launched [18]. The scene used by the framework consists of nothing but an empty object that references the origin coordinates of the simulated environment and a script named 'starter' that is executed on launch. Once the generated scripts are included and the simulation is compiled, the simulation is built at runtime rather than being saved in the scene. Consequently, the user of the tool only needs to use the DSL to edit the simulation properties. As the scene is generated according to the scripts the user does not have to do any work within the Unity environment. If the user wants to manually add any component into the simulation to extend it, he can optionally do so by adding objects into the scene before compilation.

3.2 SimGen Syntax

The domain specific language is implemented in Xtext on the Eclipse platform. The description of the simulation to be generated expressed in this language. Figure 1 shows the high-level components of the language. The definition of a simulator is a combination of MetaObjects, Objects, and an Environment.

3.2.1 MetaObjects.
A MetaObject is a description of a Unity prefab within the domain specific language. It contains the properties of the prefab which are available to the user application. A MetaObject must have the same name as the prefab that it represents in the Unity simulation. It is composed of a list of 'properties' and 'actions'.

A property in the MetaObject has a counterpart variable within the Unity prefab. For the generator to be able to successfully translate the property to the correct value in the prefab, its mapping must be recorded in the 'prototype config' file, which is explained in Section 3.3 and Section 4.4.

Actions are described in the Objects subsection.

3.2.2 Objects.
An object contains a configuration as well as a list of actions which is specific to that object. In the configuration, any variable or property that is declared in the object or its MetaObject can have its value set. As an Object must extend one

of the MetaObjects in the file, it has access to any of the actions and properties listed in the MetaObject. The Object also contains its own list of actions that are added to the actions provided by the MetaObject.

Actions allow the user to configure the behaviour of the object in the simulation by giving access to the defined properties. With actions, the user can configure objects to allow or restrict the ability of external programs to view or change the properties. Actions help define the communication protocol between the object in the simulation and the users application.

An action is made up of a 'payload', a 'return payload' and a list of assignments. An action is activated by sending a message over an appropriate channel to the simulator specifying that action's name and its object. The payload defines the data that is sent to the simulator, and the return payload defines the data that is received back from the simulator after sending that message. As each defined property represents a particular value that affects some property in the representation of the object in the simulator, setting these values within the action allows for control of every aspect or function of the simulated object.

3.2.3 Environment. The environment defines the simulation as a whole; it lists the objects which are to be present in the simulation, and the communication channel on which they are to listen for commands. Objects can belong to more than one channel, and channels can contain more then one Object. Each channel can be connected to at most one external program. The communication channel is fixed to a particular port on the simulator and enables the use of all actions that belong to the Object(s) assigned to the channel.

3.3 Code Generator

A code generator translates the description of the simulation within the DSL into C# scripts and a documentation file. The code generator uses a series of templates along with the model of the simulation created within the DSL to create these scripts. The generated C# scripts contain all the code necessary to load and configure the objects in the 3D environment, receive and translate incoming messages, perform the actions requested by the message and reply with the specified information. The description of the object's actions in the DSL defines the messages that will be sent and received as well as the change in value that is made.

Translation of MetaObject property names to their Unity prefab equivalent properties, is made possible by using the 'prototype config' file. This file contains a list that describes these mappings. During code generation, the MetaObject and property name is searched in the file and the resulting Unity variable and its location within the prefab is used within the scripts. This allows for custom MetaObjects to be made and integrated with SimGen, without the need to modify the code generator.

4 MARS ROVER EXAMPLE

To demonstrate how a user would use the tool to test a model of a Mars Rover controller, a step by step approach of the process for generating a simulation will be shown. The Mars Rover that will be used in this test uses the 3D model of the NASA Curiosity rover found at [19], and implements similar behaviour. The rover has 6

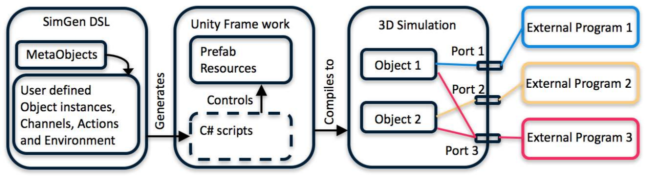

Figure 2: Creating a simulation and connecting various test programs to it

```
MetaObject Plane : others {
    property sizeX : real
    property sizeY : real
    property sizeZ : real
    property posX : real
    property posY : real
    property posZ : real
}
MetaObject NasaRover : others {
    property LFmotor : real
    property LFbrake : real
    property RFmotor : real
```

Figure 3: Defining the MetaObject

individually controlled wheels, and a system for finding its position and orientation in the simulation. This is used to build a controller which will navigate the rover on the simulated Martian surface. Figure 2 shows the steps of the entire process from configuring the simulation to connecting it to the control software that is to be tested.

4.1 Configuration

The user begins by creating a new '.prototype' file which will contain the simulation description expressed in our DSL. For the purpose of this example we will assume that the user already has the necessary prefabs loaded in his version of the Unity framework. The MetaObjects for those prefabs can then be included in the document, this would be done as shown in Figure 3.

Next the user must create an object that they would like in the simulator from the available MetaObjects that are defined, just like is shown in Figure 4. In this case the object is named 'MarsRover', and is an extension of the 'NasaRover' MetaObject. To be able to interact with the simulated rover actions must be defined. For a simple example, two basic actions are required, one to move the rover forward and one to get its position. The first action includes the payload amount, which indicates that when sending a message to the rover with the command 'setForwardPower' we must include an int value. Next we declare which properties we want to change. As shown in Figure 4 we set the three left side motors of the rover, and the three right side motors to the value 'amount' which we receive in the action message. As the properties LFmotor, RFmotor etc. are defined as properties in the MetaObject, this means that setting them will change something in the simulated model.

The next action that is defined is the GPS action which returns the X and Z position of the rover in the simulated space. The value of both posX and posZ is returned to the user upon the issuance of the GPS command to the NasaRover Object. Lastly a 'config' field is used to set the initial size and position of the rover.

```
Object MarsRover : NasaRover {
    Action setForwardPower (amount : int) return ()
    {
        LFmotor = amount
        LMmotor = amount
        LBmotor = amount
        RFmotor = amount
        RMmotor = amount
        RBmotor = amount
        Purpose : "Move forwards at this speed"
    }
    Action GPS () return (ret : real, ret2:real)
    {
        ret = posX
        ret2 = posZ
        Purpose : "get the X and Z coordinate"
    }
    config {
        sizeX = 1.0
        sizeY = 1.0
        sizeZ = 1.0
        posX = 1.0
        posY = 1.3
        posZ = 3.30
    }
}
```

Figure 4: Defining the MarsRover MetaObject

```
Env simulation {
    Instance surface : land
    Instance Rover: MarsRover
    Channel control1 direction inout type TCP (port : 8888) assign Rover
    Focus : Rover
    Author: "Michal Pasternak"
    Purpose: "Mars Rover Example"
}
```

Figure 5: Defining the simulation environment

Now that the object we want to include in our simulation is defined we can configure our simulation environment. As described in Section 3.2.3 the Environment is a collection of instances which are assigned to channels. Figure 5 shows the definition of the environment for this example. The 'MarsRover' is instantiated along with a surface for it to drive on. The rover is assigned to a channel on port 8888, and the camera is set to focus on the Rover.

4.2 Generation

Next, the code generator will create the necessary scripts to build and execute the simulation that we defined in the DSL by the code in Figures 3, 4, 5. These are in the form of C# scripts which must be placed into the resource folder of the Unity Framework. The next step is to build and save the scene in the Unity framework. We must specify which platform to build to, and the location to save the simulator to.

4.3 Execution

Now that we have a file containing the simulator we can run it to begin the simulation. On start, our simulation is initialized, and we can see a 3D environment containing our Mars rover. A server is also opened by the simulator and is listening on port 8888 as specified by our communication channel 'channel1' shown in Figure 5.

We are now ready to start up our control application and connect to the simulator through the TCP socket on port 8888. The application can send two commands to the 'Rover'. An example of a command would be "Rover,setForwardPower(25)" with no return, or "Rover,GPS()" which would return "Rover,5.0,3.0;". The 'Rover' part of the command originates from the instance name, and the 'GPS' and 'setForwardPower' components originate from the Action names defined in figure 4. From our definition we also know that we must pass a int value when calling the 'setForwardPower' command, and we know to expect a return message from the simulator after sending the 'GPS' command that will contain two real values, the X and Z position of the rover.

The user's application must be able to send these string messages to interface with the rover. Once testing is concluded, the user would only need to change their program to send messages to the hardware instead of TCP messages to the simulator.

4.4 Other Use Cases

The code generator is designed to be able to create scripts for any MetaObject in the library. It is also possible for a user, or a third party to create new MetaObjects and add them to this library. To do this the user must either create or import a 3D model or resource into the 'Resource' folder. This transforms the resource into a Unity Prefab. This Unity Prefab usually has some properties or functions that the user may want to call or query about from their application. These properties need to be mapped to a MetaObject in a text file called the 'prototype config' file as described in Section 3.3.

An example of a mapping in the 'prototype config' file would be "Rover.LFmotor=rm.ColliderL1.motorTorque". This would signal the code generator that the property 'LFmotor' of the 'Rover' MetaObject, refers to the motorTorque being applied by the component(wheel collider) named 'ColliderL1' which is accessible through another component (C# script) named 'rm' attached to the Rover prefab. These mappings are created during the creation of the prefab, by a domain expert. Any variable or property of the prefab can be accessed or changed by an action specified in the DSL if an appropriate mapping is present in the 'prototype config' file.

The design of the Unity framework component of SimGen allows it to be portable, it can be added into existing Unity projects or games to add control by external programs. Previously made game resources or environments can also be imported into the framework to allow the DSL to reference them and include them in generated simulations.

5 EXPERIMENTATION AND VALIDATION

The SimGen tool uses TCP sockets for communication. This form of communication can introduce some unnecessary overhead when both the simulator and control program are executed on the same machine. Each communication channel created in SimGen is assigned to a separate thread, which may begin to use significant resources if many simulations are running. As the number of channels increases so does the processor load, and the speed at which the simulator can check the channel for new messages. It was found that the maximum speed at which a channel can process messages is every 20 ms. Sending messages more frequently then this may lead to messages being skipped, as the channel thread records messages faster then the simulator can process them.

Metric	Given Example	Challenge Problem
Generation time	< 1 s	< 2 s
Build Time	< 5s	< 5s
Simulator Size	69.7 MB	115.9 MB
Memory Use	100.4 MB	91.1 MB

Table 1: Performance comparison of two simulations on a laptop (3.1 GHz Intel Core i5 with 16 GB 2133 MHz RAM)

To test the performance of the SimGen tool, some metrics were taken of its performance. Table 1 shows some performance metrics of the one rover simulation described in Section 4, as well as the simulation which uses two rovers and was used for the challenge problem described in Section 5.1.

5.1 Challenge Problem

One application of the SimGen tool included a challenge problem found at [20] for the MDEtools'18 workshop at MODELS 2018. The challenge problem involved two rovers similar to the one described in Section 4. Participants were asked to control one of the rovers and follow at a particular distance behind the other. Participants were free to use any MDE tool or language to create their controller.

The challenge problem made use of the SimGen tool to create the simulation component of the problem. The participants of the problem were given a set of commands to use to interact with one of the two rovers in the simulation. SimGen greatly simplified the creation of the 3D simulation by removing the necessity of knowing how to use Unity to create the simulation. Deployment of the simulation environment was possible to Mac, Windows, and Linux platforms with out any changes between the three simulations. Participant applications were compatible with all 3 versions of the simulator. Since SimGen creates simulations which are connected via TCP sockets, participants were not constrained to one language, or tool. This will allow for the comparison of the tools used by participants.

6 CONCLUSIONS

SimGen allows for the quick configuration and generation of a user defined simulation or visualization. SimGen is able to connect with any external programming language or tool that can send messages over a TCP connection. The Unity game Engine is used to render the 3D environment and solve physics calculations between the simulated objects. The DSL that SimGen uses allows for easy configuration of the simulator and specification of which objects to include. Simulations can be built to any platform, and used over a network to connect to external programs. The abilities and function of the simulator as well as the communication protocol to the various objects in the simulation can be changed according to the preferences of the user.

The Simulations created by SimGen can also be used to add external program functionality to previous games or environments built in Unity. It can also be used purely as a visualization tool, with the test code and hardware code providing the logic from different external applications. Generated simulations take place in a true 3D environment powered by one of the leading game engines on the market. Developers do not have to be familiar with the Unity software to create these simulations.

SimGen helps those who do not want to invest a lot of time into learning complicated and often expensive simulation software easily create their own custom simulators. It also provides a quick and accessible testing platform to those who wish to test their embedded systems work developed on MDE platforms with no integrated simulator. The benefits of being able to generate simulators for multiple platforms, and use any tool or language to interact with the simulator was shown in the challenge problem described in Section 5.1.

As future work, the SimGen tool would benefit from increasing the performance of the simulators which are generated. One limitation of the tool is the inability to easily generate objects at a specific time point in the simulation. This is currently overcome by creating the object at the start of simulation, and simply enabling it with an action when the user wants to use it.

REFERENCES

[1] N. Das, S. Ganesan, L. Jweda, M. Bagherzadeh, N. Hili, and J. Dingel, "Supporting the model-driven development of real-time embedded systems with run-time monitoring and animation via highly customizable code generation," in *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, 2016, pp. 36–43.

[2] M. Bagherzadeh, N. Hili, and J. Dingel, "Model-level, platform-independent debugging in the context of the model-driven development of real-time systems," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017, 2017, pp. 419–430.

[3] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A full system simulation platform," *Computer*, vol. 35, pp. 50 – 58, August 2002.

[4] "Modelica," https://www.modelica.org, accessed: 2017-01-20.

[5] "Features and description of Unity3D," https://unity3d.com/unity, accessed: 2018-07-08.

[6] J. Haas, "A history of the Unity game engine," Worchester Polythechnic Institute, Tech. Rep., 2014.

[7] I. Buyuksalih, S. Bayburta, G. Buyuksaliha, A. Baskaracaa, H. Karimb, and A. A. Rahmanb, "3d modelling and visualization based on the Unity game engine advantages and challenges," *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, vol. "IV-4/W4, pp. 161–166, 2017.

[8] M. M. A. Al, B. Christian, and H. Jorgen, "MDE-based sensor management and verification for a self-driving miniature vehicle," *Proceedings of the 2013 ACM Workshop on Domain-Specific Modeling*, 2013.

[9] E. Bonnafous, E.Saves, E. Gilbert, and J.Honore, "Experience of an efficient and actual MDE process: design and verification of ATC onboard systems," *ERTS 2008*, 2008.

[10] "Papyrus: Moka overview," http://wiki.eclipse.org/Papyrus/UserGuide/ModelExecution, accessed 2018-07-24.

[11] B. Combemale, X. Crégut, J.-P. Giacometti, P. Michel, and M. Pantel, "Introducing simulation and model animation in the MDE Topcased toolkit," in *4th European Congress on Embedded Real Time Software (ERTS'08)*, 2008.

[12] K. G. Larsen, M. Mikucionis, B. Nielsen, and A. Skou, "Testing real-time embedded software using UPPAAL-TRON: an industrial case study," pp. 299 – 306, September 2005.

[13] J. Craighead, R. Murphy, J. Burke, and B. Goldiez, "A survey of commercial and open source unmanned vehicle simulators," *Proceedings 2007 IEEE International Conference on Robotics and Automation*, May 2007.

[14] "Prescan: Simulation of ADAS and active safety," https://tass.plm.automation.siemens.com/prescan, accessed 2018-07-24.

[15] R. Ahmadi, N. Hili, L. Jweda, N. Das, S. Ganesan, and J. Dingel, "Run-time monitoring of a rover: MDE research with open source software and low-cost hardware," *Joint Proceedings of EduSymp 2016 and OSS4MDE 2016*, 2016.

[16] J. M. Thompson, M. P. E. Heimdahl, and S. P. Miller, "Specification-based prototyping for embedded systems'," vol. 1687, pp. 163–179, August 1999.

[17] M. W. Rohrer, "Seeing is believing: the importance of visualization in manufacturing simulation," pp. 1211–1216, December 2000.

[18] "Unity user manual (2018.1)," https://docs.unity3d.com/Manual/index.html, accessed: 2018-07-08.

[19] "Curiosity rover," https://nasa3d.arc.nasa.gov/detail/mars-rover-curiosity, accessed: 2018-07-23.

[20] "MDEtools18 challenge problem," https://github.com/mdetools/mdetools18/blob/master/challengeproblem.md, accessed 2018-07-23.