Crimson Publishers
Wings to the Research

Research Article

Encoding the UML Models into Script Language (U-Script)

Waseem Akhtar Mufti*

FEST, Iqra University, Pakistan

ISSN: 2640-9739

***Corresponding author:** Waseem Akhtar Mufti, FEST, Iqra University, Karachi Sindh, Pakistan

Submission: February 05, 2025
Published: February 25, 2025

Volume 3 - Issue 2

How to cite this article: Waseem Akhtar Mufti*. Encoding the UML Models into Script Language (U-Script). COJ Elec Communicat. 3(2).COJEC.000557.2025. DOI: 10.31031/ COJEC.2025.03.000557

Abstract

Protecting software from the earliest stages of development is crucial to safeguarding intellectual property. Copyright protection can be integrated at various levels of the Software Development Life Cycle (SDLC) to prevent unauthorized access and duplication. In this paper, we introduce UML Script Language (U-SCRIPT), a text-based scripting language designed to represent UML class diagrams through structured statements. U-SCRIPT enables software developers to encode UML diagrams in a secure, script-based format, concealing critical design structures from unauthorized viewers. Furthermore, it supports seamless transformation back into visual UML class diagrams. By providing both security and formal verification capabilities, U-SCRIPT helps protect algorithmic concepts and software design from unauthorized replication while maintaining usability for developers.

Keywords: Model; UML; U-script; Script; Module; Transformation; Verification

Introduction

UML is a graphical modelling language that represents programs in the form of graphical notations [1-3]. It is a generic language for graphical representation of object-oriented programs and is widely used by software practitioners for the purpose of expressiveness. Software development has become more expressive and productive by mapping each programming construct into UML's graphical notations. It abstracts away details of a particular programming language and the specifics of defining functions and classes. There are many tools to automatically generate code from a UML diagram. The typical way of developing software using graphical modelling techniques is called Model-Driven Software Engineering, which combines source code generation, graphical models, and automatic testing methods in an integrated development environment. This paper is an effort towards securing program design by encrypting into specialized language and creating an automatic integrated software engineering environment. The proposed script language, written in specialized code, would replace the graphical representation of object-oriented programs in the process of integrated development. The objective of specifying object-oriented programs in formal and precise notations is to keep our model secure from the outside world. Another advantage of U-SCRIPT is that it is formal enough to be translated into Java or any other object-oriented programming language. The transformation of one programming language into another is often required for the purpose of expressiveness and simplicity i.e. required by the model checking tools [4,5].

The specifications of systems in U-SCRIPT should be easier to support the process of formal analysis because it is easy to be an input to a model checking algorithm and more expressive than UML as well [6,7]. This language can be used as a specification language for object-oriented programs. The specified programs can then be formally verified to check for possible errors before writing the actual program; this activity should be part of an SDLC [8]. The technique is called formal analysis of programs by writing the specifications as input to the model checking tools [9-11]. Most model-checking tools receive an abstract model specified in some specification language as an input and then generate output that indicates

possible errors; otherwise, the model-checking tool returns true if the specification contains no errors. Another benefit of using a text-based script for specifying models is that it occupies less space, making it easier to communicate over a network. Compared to UML, U-SCRIPT is lightweight enough and hides the logic of the program from others, making it harder to copy easily. Transformation from one language to another is also easy, depending on the different specification languages used by different model-checking tools.

U-SCRIPT Language

We present a script-based language that describes the model depicted in UML. Since the UML model is a graphical representation of object-oriented programs, we assume that these diagrams have two-fold limitations:

a. They make programs easily readable and understandable to the outside world quickly.

b. Graphical models are difficult to process for formal analysis compared to models specified in a text-based language.

In this section, we present the representation of object-oriented Class definition, Inheritance, Composition, Aggregation, Interface and relationship labels [12]. These are the basic constructs of an object-oriented program which are graphically represented in UML model. Furthermore, we present how to specify each of these constructs in our script language using an example of a plant that has a control node and some working nodes. Each node can be represented as an independent class bound to some relationship with other classes. Each of the relationships is well defined, and how the variables and their types are specified is also given in the following section.

Plant example

A manufacturing plant or a digital twin has a control node responsible for containing the existing information about the product being manufactured or monitored, including the product's basic information received from sensors or direct input. There are two other nodes, referred to as Plant1 and Plant2, in the main plant connected to the Control Node as shown in Figure 1.

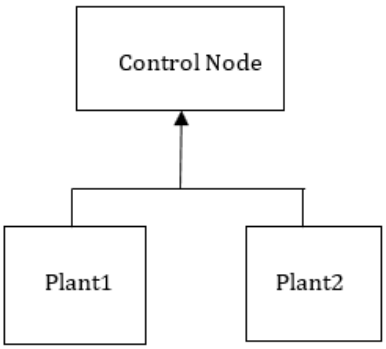

Figure 1: UML generalization of a manufacturing plant.

Generalization

Given the above diagram the Control Node class is inherited by two classes Plant1 and Plant2. Assuming some variables and methods in each class the given diagram can be represented by U-SCRIPT as follows:

```
Control_Node[] <-- Plant1, Plant2;
```

Each of these classes is defined as the collection of its variables and member functions. The scope of the members is defined as (+), (-), (#) meaning public, private and protected respectively. The scope of classes is defined using these symbols. By default, each element is considered as public if no scope specifier is given.

```
Control_Node[]:  +int x, y, String descrip-
tion;
                 -int test_array;
Plant1[]: +double dim1, dim2, dim3, volume;
Plant1[]: +void dye_make();
Plant2[]: +double density, temperature;
Plant2[]: +void material_casting();
```

Composition

A composition relationship can be represented with a diamond as shown below. This relationship means an object is a part of another object and cannot exist as an independent software component, as shown in Figure 2. The U-SCRIPT equivalent model is also provided below. It should be noted that Coolant is a normal class that may have its own member variables and methods. Therefore, at this level, this class has not been defined for the purpose of simplicity, but it can be defined similarly to other classes.

Figure 2: UML composition of a manufacturing plant.

```
Plant1[ ]<.>     Coolant[ ];
```

Where "[ ]" means UML box (Class) and "<.>" is composition relationship. Similarly, if there is an Aggregation relationship it can also be defined similarly with a minor change. In UML it is denoted by empty diamond. The equivalent U-SCRIPT is given below.

```
Plant1[]<>       Coolant[];
```

Interface

The interface is defined as Interface_Name[I], almost similar to a normal class where methods signatures and the possible variables are also defined similarly. Classes that use this interface can be connected by (<--) with interface.

UML to U-SCRIPT

The basic UML symbols used for the class diagram have been equivalently defined using U-SCRIPT. In this section, we provide a detailed example of a software module specified in a UML class model to be translated later into U-SCRIPT. This module, shown in Figure 3, contains several classes and relationships. The software module is responsible for acquiring data from sensors, performing functions such as data ingestion, and executing a method for data analytics. The UML diagram does not include variables and other methods in the classes for simplicity. The purpose is to show the transformation of the UML model into U-SCRIPT, so the details of ingestion, sensor data acquisition, and data analytics algorithms are omitted in this article. These algorithms can be added later without increasing the complexity of relationships. Adding new member functions and data will also not increase the complexity in U-SCRIPT; complexity only grows when the number of relationships increases. Our script language is capable enough to specify even complex UML models easily and quickly. Figure 4 elaborates U-SCRIPT more clearly in detail by giving its all-basic building blocks and mapping from UML to U-SCRIPT. The (*) operator indicates the classes that are further related to other member classes (lines 2,3 and 8). The classes with no further relationship are terminal classes without the (*) operator. The Interface represented as [I] does not contain (*) because it always contains some implementation relationship with its descending classes as given as in lines 1 and 2. The U-SCRIPT given in Figure 4, which is the complete transformation of UML class diagram given in Figure 3, is created according to the mapping rules defined in earlier sections.

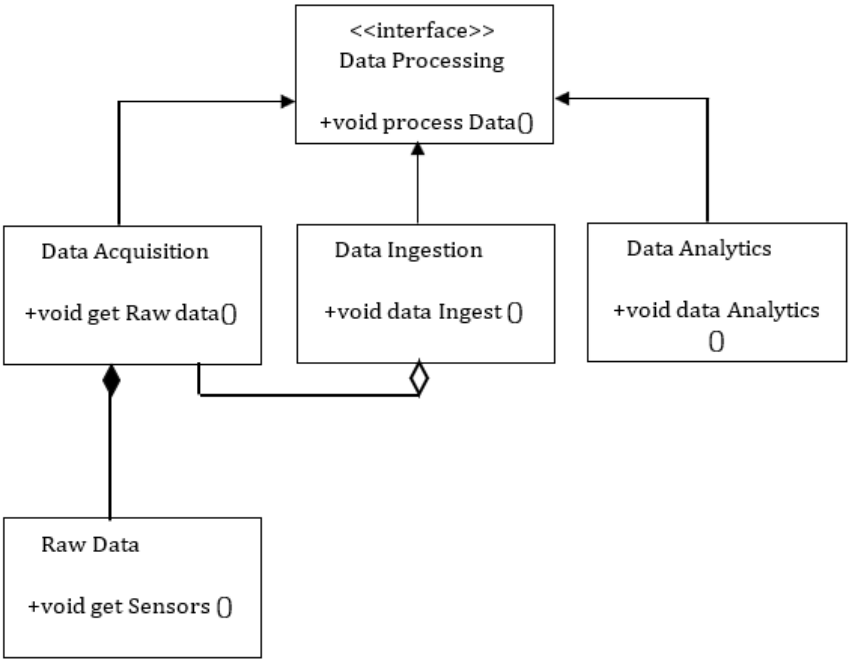

Figure 3: UML data processing module.

```
1. Data_Processing[I]: +void processData();

2. Data_Processing[I]<-- {    *Data_Acquisition[],
3.                            *Data_Ingestion[],
4.                             Data_Analytics[]; }

5. *Data_Acquisition[]: +void getRawdata();
6.          Data_Acquisition[]<.>   Raw_Data[];
7. Raw_Data[]: +void getSensors();

8. *Data_Ingestion[]: +void dataIngest();
9.          Data_Ingestion[]<>      Data_Acquisition[];

10. Data_Analytics[]: +void dataAnalytics();
```

Figure 4: UML data processing module.

Applying U-SCRIPT

The given UML module is represented in U-SCRIPT in Figure 4. It must be noted that the transformation from UML to U-SCRIPT starts from the first available box in UML (class or interface) and proceeds from top to bottom according to the flow of UML diagram. U-SCRIPT allows specifying each member of UML without adhering to the flow order of UML. However, it is advisable to write the U-SCRIPT specification in line with the UML flow order. Converting back from U-SCRIPT to UML can be a useful approach to recreate the UML diagram from U-SCRIPT. The textual format that is used by U-SCRIPT allows for precision in specification and therefore it can be used for formal verification.

U-script in automatic verification

U-SCRIPT-enabled automatic verification should be a potentially helpful tool for error identification at the early stages. It will ensure that class relationships, method behaviour and data flow are logically correct before implementing, as shown in Figure 5. It can discover design flaws that may only reveal themselves during the whole testing stages or even during the manufacturing value chain. Incorporating early evidence into correctness early in the development process is received and U-SCRIPT refines software reliability and most importantly in areas of crucial application such as medical, finance and aerospace. For example, the specification that a data acquisition and processing workflow complies with safety standards will avoid a real-life failure in industrial applications. The most interesting application of U-SCRIPT is that it can potentially contribute in the growth of automatic verification of software with the aid of the approach under formal methods. The automated analysis of UML diagrams is conducted by translating them into the U-SCRIPT language and follows a methodical approach to model checking necessary for the early identification of defects within the software development process. On the other side, the formal methods use a mathematical approach to prove that a model or given specification conforms to the requirement and hence produces software that works as designed. In light of this, the Syntax of the U-SCRIPT is less complex than that of the UML diagrams and therefore it is easier to verify through the verification tools. One of the aspects possible due to the script-based language is the ability to check if the relationships (inheritance, composition, aggregation) and constraints between classes for a model are consistent with the requirements.

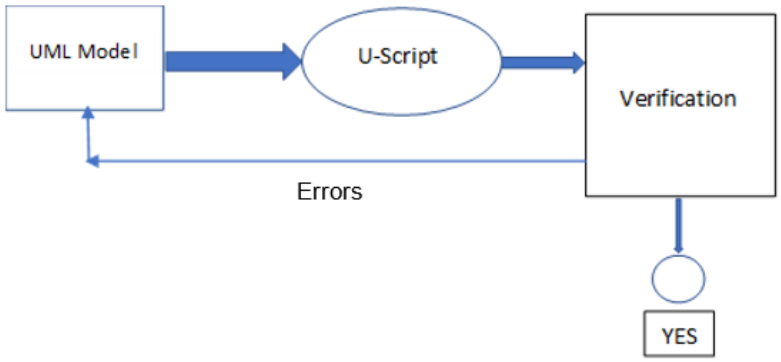

Figure 5: UML data processing module.

U-SCRIPT models can be used as input for model-checking tools to model check hence can systematically exhaust the states of the model to check properties such as correctness, security, and reliability. These tools can also analyse the system logic through specifications from the U-SCRIPT and define inconsistency, such as a deadlock or unreachable state, in large and intricate software systems. The relation of UML models to U-SCRIPT assists the developers because it is possible to perform the automated verification at the design level, usage level and code generation level without having to manually go through these phases. This helps minimize chances of human error and any design errors will also be detected right from the design process.

U-SCRIPT bears the characteristics of what could be considered a Domain Specific Language (DSL), perhaps for embedded systems and other more specific areas of application. Apparently, future versions of the U-SCRIPT may embed timing issues and concurrency control issues important for environments that need real time checking. With timing, state transition, and response time constraints being very important to these applications. U-SCRIPT could be enhanced to specify the realm of these properties reside in further enhancing its capability to validate software within highly restrictive specifications.

Related Work

Formalisms may at times require a model transformation for accomplishment of the goal as one of the methods. The formal models are also essential for the aim of automatic verification of models or programs to locate bugs which would not potentially be in manual or automated testing. Plant UML is a script-based language which uses HTML like tags to encode a set of UML diagrams with colour fonts [13,14]. Unlike our specific language, Plant UML has a style of programming language. Whereas, UML-Script (U-SCRIPT) has been described as another attempt in trying to translate UML diagrams into a more sophisticated format, not those following the programming approach of embedding diagrams into the text,

rather through one-to-one mapping of each UML construct to one U-SCRIPT symbol. It does encode more or less the graphical models in the text format through the use of a more declarative language.

Future Work

U-SCRIPT's future improvement strategies will also include some of the timing and concurrency requirements which are critical for embedded systems and some other applications that require timing constraints and other non-functional requirements [15]. Embedded systems, which are found in certain safety-critical environments include aviation, construction equipment, power generation facilities, healthcare and require clear definitions of both functional and non-functional properties [16,17]. Why we aim at enhancing U-SCRIPT by including timing requirements for embedded system specifications for example, is because embedded systems are used for controlling several life-threatening machines like airplanes, construction cranes, electrical and nuclear power plants, machinery in chemical laboratories, etc. It is necessary for such systems to have a formal specification language for their functional and non-functional systems properties [18]. Modelling in U-SCRIPT to meet such requirements will enable the specifications of systems to be compliant with strict operating conditions and enhance reliability in critical situations. In addition, we focus on enabling approaches to automatically create UML diagrams from U-SCRIPT specifications, providing model transformation from textual to graphical representation and vice versa. This will add additional value to existing workflows and make the transition easier for developers focused on the usage of UML. Future iterations of U-SCRIPT may also include support for additional UML diagram types, such as sequence and activity diagrams, further broadening its applicability. Moreover, by incorporating domain-specific constructs tailored to real-time systems, such as timing constraints and response deadlines. U-SCRIPT can evolve into a comprehensive specification language for various domains. This advancement will enable more efficient formal verification processes, allowing automated analysis of complex interdependencies, improved scalability, and enhanced accuracy in modelling and verification tasks.

Conclusion

In this paper, we present U-SCRIPT, an innovative text-based scripting language specifically created to securely encode UML class diagrams while preserving the necessary expressiveness and structure for formal software modelling. U-SCRIPT serves as a script-based alternative to the traditional graphical representation of UML, offering numerous benefits such as enhanced security, efficient model transformation, and better support for formal verification. This method ensures that software designs are safeguarded against unauthorized access while remaining analysable and convertible back into UML class diagrams when required. One of the primary benefits of U-SCRIPT is its capacity to integrate with formal verification techniques, which allows for early detection of errors in software design. By converting UML models into a structured text format, U-SCRIPT promotes automated verification, thereby minimizing human error and increasing the reliability of software systems. This feature is especially advantageous in sectors where accuracy and security are critical, including finance, aerospace, and healthcare. Furthermore, the lightweight design of U-SCRIPT makes it an efficient choice for software modelling, as it reduces storage and transmission needs while maintaining vital design information. In addition to its security and verification capabilities, U-SCRIPT enhances the field of Model-Driven Software Engineering (MDSE) by connecting graphical and textual modelling methodologies. Its ability to articulate object-oriented relationships such as inheritance, composition, aggregation and interfaces in a clear script format positions it as a valuable resource for software developers and researchers. The potential for automatic transformation between UML and U-SCRIPT paves the way for improved workflow integration, decreasing the necessity for manual translation and facilitating a more efficient software development lifecycle.

Future developments in U-SCRIPT will aim to enhance its functionality by adding support for more UML diagram types, including sequence and activity diagrams, thereby increasing its utility in software design. Additionally, the integration of features for modelling real-time and concurrent systems will enhance U-SCRIPT's relevance for embedded systems and applications that require timely responses. Improving its syntax to accommodate timing constraints and response deadlines will further solidify its importance in safety-critical and high-reliability contexts. In essence, U-SCRIPT marks a significant advancement in secure software modelling, offering a viable approach for encoding, verifying, and transforming UML-based designs. By incorporating formal verification, security measures, and efficient model transformation, it presents a promising avenue for both academic research and industry applications, driving innovation in software design, security, and automated analysis. As ongoing efforts continue to enhance its capabilities, U-SCRIPT is poised to become a vital resource in the dynamic field of software engineering.

References

1. Rumpe Bernhard (2016) Modelling with UML. Springer 98.
2. Pilone Dan, Neil Pitman (2005) UML 2.0 in a Nutshell. O'Reilly Media, Inc, Massachusetts, USA.
3. Miles Russ, Kim Hamilton (2006) Learning UML 2.0. O'Reilly Media, Inc, Massachusetts, USA.
4. El-Ramly, Mohammad R Eltayeb, Hisham A Alla (2006) An experiment in automatic conversion of legacy Java programs to C. IEEE International Conference on Computer Systems and Applications, Dubai, UAE.
5. Fussell Mark L (1997) Smalltalk to java using language transformation to show language differences.
6. Felleisen Matthias (1990) On the expressive power of programming languages. European symposium on programming. Springer Verlag GmbH, Berlin, Germany.
7. Jamil Fahad Rami (2024) Evaluating the expressiveness of specification languages: For stochastic safety-critical systems.
8. Nayak Anmol, T Hari Prasad, Vidhya Murali, P Karthikeyan, VG Venkoparao, et al. (2022) Req2Spec: Transforming software requirements into formal specifications using natural language

processing. Requirements Engineering: Foundation for Software Quality. Springer International Publishing, Switzerland, pp. 87-95.

9. Campbell Sherrie, Ann E Kelley Sobel (2008) Supporting the formal analysis of software systems. 2008 International Conference on Computer Science and Software Engineering, Wuhan, China.
10. Bonfanti Silvia, Angelo Gargantini, Atif Mashkoor (2018) A systematic literature review of the use of formal methods in medical software systems. Journal of Software: Evolution and Process 30(5): e1943.
11. Marinescu Raluca, et al. (2015) Analysing industrial architectural models by simulation and model-checking. Formal Techniques for Safety-Critical Systems: Third International Workshop, Luxembourg.
12. Rentsch Tim (1982) Object oriented programming. ACM Sigplan Notices 17(9): 51-57.
13. Axt Monique (2023) Transformation of sketchy UML class diagrams into formal plant UML models.
14. Correia Filipe F, et al. (2024) Towards living software architecture diagrams. ArXiv preprint arXiv: 2407.17990.
15. Czarnecki Krzysztof, Simon Helsen (2003) Classification of model transformation approaches. Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture 45(3):
16. Dömer Rainer, Daniel D Gajski, Jianwen Zhu (1998) Specification and design of embedded systems. It-Information Technology 40(3): 7-12.
17. Fruth Matthias (2005) Formal verification of embedded real-time systems.
18. Visser Eelco (2008) WebDSL: A case study in domain-specific language engineering. Generative and Transformational Techniques in Software Engineering II: International Summer School, Braga, Portugal, pp. 291-373.