An Automated Verification Framework for DEVS Coupled Models to Enhance Efficient Modeling and Simulation

Gyuhong Lee and Suman Nam *

Posted Date: 19 March 2025

doi: 10.20944/preprints202503.1404.v1

Keywords: Discrete Event System Specification; Model Verification; Coupled Models; Simulation Accuracy

Article

An Automated Verification Framework for DEVS Coupled Models to Enhance Efficient Modeling and Simulation

Gyuhong Lee [1] and Suman Nam [2,*]

[1] TIM Solution

[2] Cheongju University

* Correspondence: smnam@cju.ac.kr; Tel.: +82-43-229-8493

Abstract: Discrete Event System Specification (DEVS) is a formalism widely used for modeling and simulating complex systems. The main features of DEVS are defining models in a strict mathematical form and representing systems through hierarchical structures. However, when DEVS models have incorrect connection structures and inappropriate behaviors contrary to design intentions, simulation results can be distorted. This can cause serious problems that may lead to inaccurate decision-making. In this paper, we propose an automated verification framework to improve the accuracy and efficiency of coupled models in the DEVS-Python environment. This framework defines test scripts for coupled models, performs automatic verification before simulation execution, and provides the results to users. Experimental results showed that the proposed framework improved execution time by approximately 30-100 times compared to traditional unit testing methods, although memory and CPU usage increased slightly. Despite this increase in resource usage, the proposed framework provides high efficiency and consistent performance in verifying complex DEVS coupled models.

Keywords: discrete event system specification; model verification; coupled models; simulation accuracy

1. Introduction

Discrete Event System Specification (DEVS) plays a crucial role in modeling and simulating complex systems [1-5]. Developed by Zeigler, DEVS can effectively represent complex systems through discrete event abstraction with its systematic and modular approach [4, 6, 7]. The strength of DEVS lies in its model definition and hierarchical model structure based on a rigorous formalism [8-10]. This formalism consists of coupled models that represent the system's structure and atomic models that express its behavior [5, 11].

DEVS-based simulation system research has been steadily advancing through systems such as DEVS-Scheme [12-14], DEVSJava [15, 16], DEVS-Python [17, 18], ADEVS [19], and DEVSim++ [6]. Some of these systems, like DEVS-Scheme and DEVS-Python, provide test environments for verifying atomic models. However, despite coupled models being composed of complex structural elements such as sets of child models, input and output port information, internal and external coupling relationships, and execution priorities, there is currently no systematic test environment to verify these elements [20, 21]. As a result, verification of coupled model structural accuracy primarily occurs through abnormal simulation execution, requiring modelers to manually identify errors. This process demands considerable time and costs in the modeling process, thus emerging as an urgent task in DEVS research to develop an effective testing method for coupled models.

In this paper, we propose an automated verification framework for DEVS coupled models to improve the computational resource efficiency of DEVS-Python. Our proposed framework defines test scripts to systematically verify all structural elements of coupled models, including in/out ports, sub model composition, coupling relationships, and execution priorities, and performs automatic

 doi:10.20944/preprints202503.1404.v1

verification before simulation execution in DEVS-Python, providing the results to the user. Unlike traditional test-driven development [21-23], this framework can reduce coupled model verification time and maintain similar CPU usage, thereby enhancing execution time efficiency.

The contributions of this paper are as follows:

- Proposing an automated verification framework that systematically verifies structural elements of DEVS coupled models through comprehensive test scripts;
- Developing an innovative verification approach to enhance computational efficiency and reduce verification time in DEVS-Python simulation modeling.

This paper introduces DEVS formalism and discrete event systems in Section 2, explains the existing DEVS testing methods in Section 3. In Section 4, we introduce the proposed framework, and in Section 5, we analyze the experimental results. Section 6 provides conclusions and discusses future research directions.

2. Related Work

This section explains DEVS formalism and the existing testing methods. Figure 1 shows the main components of the modeling and simulation process and their relationships. In [24], the (1) Real System is the subject we aim to study, and the (2) Model Specification is its mathematical representation. The (3) Simulator is an implementation of the model specification in an executable form, consisting of source code and an execution engine.

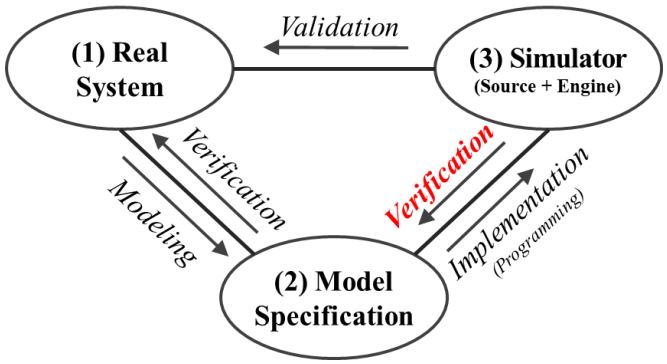

Figure 1. Modeling and Simulation Process: Components and Interactions.

These components interact through four relationships. *Modeling* is the process of observing and analyzing the real system to abstract it into a model specification, and *Implementation* is the process of implementing the model specification as a simulator through programming. *Verification* is the process of verifying whether the implemented simulator matches the model specification, confirming that the model has been accurately implemented as intended. *Validation* is the process of verifying whether the simulation results align with the behavior of the real system. This research focuses on the *Verification* process, proposing a framework to automatically verify the structural accuracy of DEVS coupled models.

Section 2.1 explains the basic concepts of DEVS formalism and the structures of atomic and coupled models, while Section 2.2 delves into existing testing techniques and the architecture of the DEVS engine system.

2.1. DEVS Formalism

The DEVS formalism [1-3, 5, 8, 11] is a mathematical framework for modeling and simulating complex systems. It provides a theoretical and well-defined means of representing hierarchical and modular individual event models to analyze these complex systems. A key characteristic of DEVS is that it decomposes systems into modularized components, not only allowing each component to operate independently but also executing them in a manner where system states change according to

 doi:10.20944/preprints202503.1404.v1

event occurrences. Therefore, DEVS is distinguished by its atomic models that describe system behavior and coupled models that represent system structure. In DEVS, an atomic model has inputs, outputs, states, time, and functions for a system. The system's functions determine the next state and output based on the current state and input. An atomic model consists of three set (X, S, Y) and four functions $(\delta_{int}, \delta_{int}, \lambda, t_a)$.

A coupled model is a complex model created by connecting multiple atomic models or other coupled models. The formal representation of this coupled model is as follows:

$$CM = \{X, Y, \{M_i\}, EIC, EOC, IC, select\}.$$

In the coupled model equation, the information from $\{M_i\}$, EIC, EOC, IC enables the expression of system structure and connectivity, allowing the creation of complex models by combining various models [25]. Here, X and Y represent the set of input and output events for coupled models.

2.2. Testing Methodology of Traditional DEVS Engines

Since its initial proposal by Bernard P. Zeigler in 1976, the DEVS formalism has evolved through various implementation systems [20]. Beginning with DEVS-Scheme (1987), the first implementation of DEVS tools, subsequent systems such as DEVSim++ (1991), DEVS-Suite (1997, formerly DEVSJava), ADEVS (2001), and DEVS-Python (2022) were sequentially developed, with each system providing modeling, simulation, and testing capabilities based on its unique characteristics.

DEVS-Scheme, as the first implementation of DEVS formalism, supports modeling and simulation of discrete event systems by leveraging the functional programming characteristics of the Scheme language. In particular, its object-oriented design enables hierarchical composition of atomic and coupled models, allowing mathematically based formal verification. However, while DEVS-Scheme's modularized structure enables component-based testing, it did not achieve complete test automation due to the complexity of coupled models.

DEVS-Suite, developed at Arizona State University, has evolved from the initial DEVSJava and now reaches version 6.0. The latest version introduces a black-box testing framework to support model test automation. In particular, it provides systematic test case definition and automated regression testing through JUnit integration, *TestFrame* class, and *@TestScript* annotations [22]. However, this testing framework is limited to atomic model verification and lacks comprehensive capabilities for validating the complex interactions and temporal dependencies of coupled models.

DEVSim++ and ADEVS, both implemented in C++, offer unique advantages. Developed in 1991, DEVSim++ focuses on efficient implementation and simulation of DEVS models, while ADEVS, developed in 2001, provides a lightweight library for high-performance simulation. However, both systems lack a dedicated testing framework, resulting in model verification that relies on manual confirmation through simulation execution.

Recently, DEVS-Python was developed by Nam (2022). This system leverages Python's flexibility and intuitive syntax to facilitate learning and implementation of DEVS concepts. In terms of testing, it supports atomic model verification using Python's native testing framework [18, 21], but like other tools, it does not provide automated testing for coupled models.

Analysis of these DEVS tools reveals a common limitation in coupled model testing. In particular, manual verification methods have high potential for human error and difficulty in ensuring test coverage. Therefore, the development of a systematic test automation methodology that considers the structural characteristics and dynamic behaviors of coupled models is urgently needed.

3. Problem Statement

In this section, the verification stage (Simulator → Model Specification as shown in Figure 1) reveals two major limitations in the existing DEVS testing methods:

 doi:10.20944/preprints202503.1404.v1

- The existing methods primarily focus on testing for atomic models without coupled model verification [18, 21]. As a result, verification of coupled model structural elements (child models, coupling, priorities) is mainly performed manually, which increases the risk of human errors;
- While the existing DEVS engines can apply unit testing provided by JAVA, C++, etc., this requires changing the entire development process and takes considerable time to understand and implement. Particularly, the lack of automated tools for integrated testing of coupled models makes it difficult to verify model accuracy.

As explained in Section 2.1, DEVS coupled models consist of seven key elements. These coupled model elements must be defined during model design. For example, Table 1 shows the formalism of the Experimental Frame (EF) coupled model presented in [7, 10].

Table 1. Formalism of EF Coupled Model.

$X = (null)$
$Y = (null)$
$\{M_1, M_2\} = \{genr, transd\}$
$EIC = (ef.in, transd.solved)$
$EOC = (genr.out, ef.out), (transd.out, ef.result)$
$IC = (transd.out, genr.stop), (transd.out, genr.stop), (genr.out, transd.arrived)$
$select = (genr, transd)$

In this coupled model formalism, the model's input and output values (X and Y) are not defined when the coupled model is initially created and are only determined during simulation execution. The coupled model includes two child models named $genr$ and $transd$, consisting of one EIC, two EOCs, and three ICs. Additionally, the model execution order, determined by the select function, proceeds from $genr$ to $transd$.

These coupled model elements can be implemented as actual source code using DEVS-Python, with an example provided in Table 2. Through this implementation process, the complexity of coupled models and the importance of testing become more prominent, emphasizing the need for an effective coupled model verification method.

Table 2. An Example Source Code for EF Coupled Model using DEVS-Python.

```
1: class EF(COUPLED_MODELS):
2:     def __init__(self):
3:         COUPLED_MODELS.__init__(self)
4:         self.setName(self.__class__.__name__)
5:
6:         self.addInPorts("in")
7:         self.addOutPorts("out", "result")
8:
9:         genr = GENR()
10:        transd = TRANSD()
11:
12:        self.addModel(genr)
13:        self.addModel(transd)
14:
15:        self.addCoupling(self, "in", transd, "solved")
16:        self.addCoupling(genr, "out", self, "out")
17:        self.addCoupling(transd, "out", self, "result")
18:        self.addCoupling(transd, "out", genr, "stop")
19:        self.addCoupling(genr, "out", transd, "arrived")
```

 doi:10.20944/preprints202503.1404.v1


```
20:
21:         self.priority_list([genr, transd])
```

As shown in the coupled model source code, the modeler-written code from lines 6 to 21 needs to be verified after implementation. Specifically, these elements include the set of models ($\{M_i\}$), external input coupling (EIC), external output coupling (EOC), and internal coupling (IC). Since these elements are directly input by the modeler in the source code, they have a relatively high potential for human error. For example, there is a valid coupling $(genr.out, transd.arrived)$ that transfers from a source model's port (*source.port*) to a destination model's port (*destination.port*). Common typing errors involve parameter errors such as character sequence order and deletion. When string errors occur, it takes considerable time to locate them. [26] indicates that when coding in Python, parameter errors and incompatible return types account for 64.8% of all errors in the dataset. Therefore, the main issues that occur in DEVS coupled models are as follows.

- Limitations of manual verification for structural elements (child models, coupling, priorities) of coupled models
- Lack of integrated test automation tools for coupled models

To address these issues, we propose an automated verification framework that can verify all structural elements of DEVS coupled models.

4. Proposed Verification Framework

This section proposes an automated verification framework for DEVS coupled models to address challenges in testing and verifying model structures. Section 5.1 provides an overview of the proposed system, and Section 5.2 presents the detailed procedures of the verification framework.

4.1. Overview

In this paper, we propose a system to improve accuracy and efficiency through automated testing of coupled models in the DEVS-Python environment. The proposed system generates test code that includes child models, coupling information, and execution priorities. Subsequently, it automatically parses the test code and the coupled model's source code to efficiently compare and analyze the two sources of information. When asymmetric information is discovered, the method visually indicates the error areas to the modeler, enabling immediate corrections. This automated system significantly reduces verification time compared to manual inspection methods while simultaneously enhancing test accuracy. In particular, by ensuring the structural accuracy of coupled models, it contributes to improving the reliability of simulation models.

The overall process of the proposed system is illustrated in Figure 2. The upper stage represents the typical modeling and simulation development workflow, while the yellow-colored process below depicts the coupled model verification system proposed in this paper.

The existing modeling and simulation development process begins with defining a Conceptual Specification based on the Real System, followed by Model Design. The designed model is then implemented through Model Code Generation and ultimately verified through Performance Analysis.

The proposed system consists of three key stages, running parallel to the existing process:

1. *Coupled Model Test Design*: Design test cases for the coupled model defined in the model design stage.
2. *Coupled Model Test Code Generation*: Automatically generate test code based on the designed test cases.
3. *Test Result Verification*: Execute the generated test code and verify its results.

Through this automated verification stage, the structural accuracy of coupled models can be ensured, and potential errors in the development process can be detected and corrected early.

 doi:10.20944/preprints202503.1404.v1

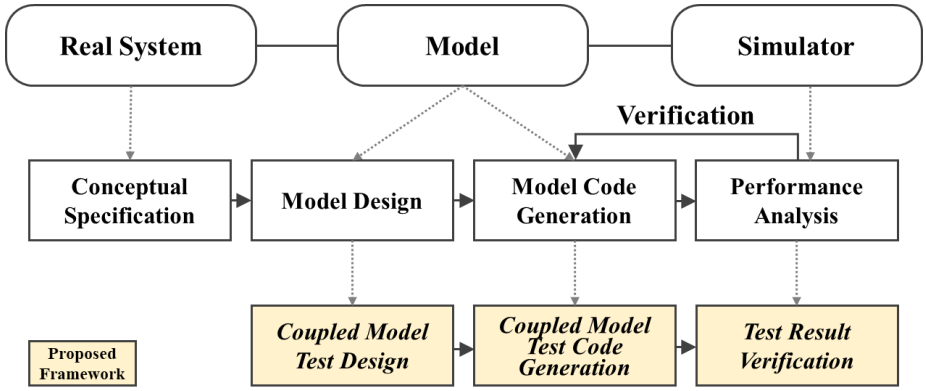

Figure 2. Overview of Proposed Framework.

4.2. Exeuction Procedure

In this section, we introduce the execution procedures of the proposed verification framework. Section 4.2.1 presents the coupled model test design, Section 4.2.2 describes the coupled model test code generation, and Section 4.2.3 explains the test result verification.

4.1.1. Coupled Model Test Design

The coupled model test design stage is the initial stage for verifying the structural correctness of DEVS-based coupled models [21]. This stage defines test items such as the set of child models, input/output port information, internal and external coupling relationships, and execution priorities, which are the main components of coupled models. During test design, the modeler clearly specifies the intended structure of the coupled model and documents it in a form that can be used in the subsequent automated verification stage. For example, when a coupled model CM consists of two atomic models AM1 and AM2, EIC verifies that the input port p1 of the coupled model is connected to the input port in1 of AM1, IC confirms that the output port out1 of AM1 is correctly connected to the input port in2 of AM2, and EOC tests that the output port out2 of AM2 is accurately connected to the output port p3 of the coupled model. This establishes a foundation for effectively identifying and verifying structural errors in coupled models.

4.1.2. Coupled Model Test Code Generation

The coupled model test code generation is a stage that converts the test items for verifying structural correctness of the designed coupled model into actual verifiable code. The test code is written in JSON format and systematically represents the metadata and structural information of the coupled model. This is automatically processed through a Parser in the verification stage to identify structural errors. The generated test code ensures the accuracy and consistency of verification and enables repetitive test execution.

The JSON test code shown in Table 3 is an example for verifying the structure of the EF model. This code specifies that the EF model consists of two child models, GENR and TRANSD, and indicates that TRANSD has execution priority over GENR when simultaneous events occur. Additionally, the code includes five coupling relationships: an EIC {"EF.in": "TRANSD.solved"}, ICs {"TRANSD.out": "GENR.stop"} and {"GENR.out": "TRANSD.arrived"}, and EOCs {"GENR.out": "EF.out"} and {"TRANSD.out": "EF.result"}.

Table 3. An Example Source Code for EF Coupled Model using DEVS-Python.

```
1: {
2:     "log": {
3:             "author": "Su Man Nam",
```

 doi:10.20944/preprints202503.1404.v1


```
 4:            "data": "Date",
 5:            "ver": "Coupling Information of EF Model"
 6:    },
 7: "EF": [
 9:            {"model": "GENR, TRANSD"},
10:            {"select": "TRANSD, GENR"},
11:            {"EF:in": "TRANSD:solved"},
12:            {"GENR:out": "EF:out"},
13:            {"TRANSD:out": "EF:result"},
14:            {"TRANSD:out": "GENR:stop"},
15:            {"GENR:out": "TRANSD:arrived"}
16:    ]
17: }
```

4.1.3. Coupled Model Test Design

Verification of the coupled model test results involves parsing the generated JSON-formatted test code to automatically verify the structural accuracy of the coupled model. In this stage, the method verifies the set of child models, input and output ports, coupling relationships, and execution priorities are correctly defined.

Figure 3 illustrates the entire process of verifying test results for coupled models. First, the coupled model to be verified is selected from the model base [4, 9]. Next, a test script for the model is selected from the test base to verify the test results. If the verification result is unsatisfactory (No), the process returns to the test script selection step to perform verification with a different test case. When verification is completed (Yes), the entire process is terminated.

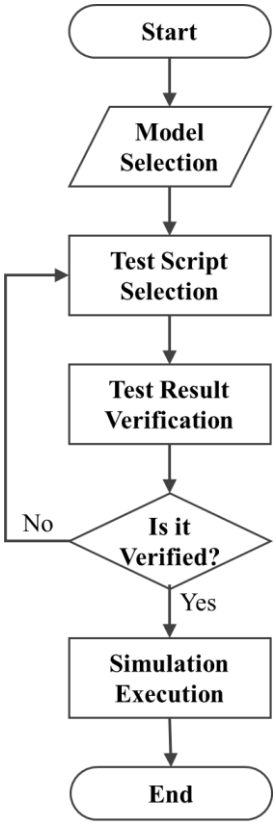

Figure 3. Flowchart of Coupled Model Test.

The proposed framework verifies the accuracy of the coupled model code directly written by the modeler, aiming to prevent human errors that may occur during manual processes. Particularly,

 doi:10.20944/preprints202503.1404.v1

mistakes that can happen while manually typing coupling information may lead to simulation malfunctions or unexpected results. Through this iterative verification process, the structural accuracy of the coupled model can be ensured.

The core of the proposed system is to compare the test code written in Section 4.2.2 with the coupled model's source code and provide verification results to the modeler. This system minimizes potential mistakes that can occur when the modeler separately writes test code and source code based on design specifications. In particular, unlike traditional unit testing, our system uniquely features the verification of coupled models always performed before simulation begins.

In the verification stage, the SequenceMatcher from Python's difflib [27] module is utilized. SequenceMatcher is a tool for comparing and analyzing the similarity between two sequences, returning a list of synchronization operations through the *get_opcodes()* method. The returned operation list consists of replace, delete, insert, and equal, with each operation indicated by a different color. For example, as shown in Table 4, when str1 is 'qabxcd' and str2 is 'abycdf', the replace operation indicates that 'x' in str1 should be replaced with 'y' in str2. In practical application, when comparing coupling information such as 'TRANSD.arrived' and 'TRANSD.arived', the 'r' would be highlighted in green, enabling the modeler to visually identify and correct coupling errors.

Figure 4 provides a visual example of the verification results for a coupled model. Specifically, it reveals a typographical error in the coupling between GENR.out and TRANSD.arrived, where 'arrived' is mistakenly typed as 'arived'. When such a typo is detected through SequenceMatcher, the corresponding part is highlighted in green, enabling the modeler to easily identify and correct the error.

```
json_file : GENR.out -> TRANSD.arived
model_coulping : GENR.out -> TRANSD.arrived
GENR.out -> TRANSD.arrived
```

Figure 4. Verification Result of The Coupled Model.

The coupled model testing framework proposed in this study consists of three stages: test design, test code generation, and test result verification. In particular, the result verification stage using SequenceMatcher visually indicates errors in the coupling information written by the modeler, enabling effective debugging. Through this approach, the reliability of DEVS-based simulation can be enhanced, and the structural accuracy of complex coupled models can be ensured.

5. Experimental Results

In this section, we selected traditional unit testing methods (unitest [28], Pytest [29]) as comparative groups to evaluate the efficiency of the proposed verification framework. The experimental environment consists of Windows 11 with an Intel(R) Core i5-14500 2.6 GHz processor with 14 cores and 32 GB of RAM. Visual Studio Code (VSCode) was used for conducting these experiments. The experimental models include the basic coupled models EF and EF-P, as well as the complex coupled models SENSORS and ACLUSTERS presented in [7, 10]. The selected coupled models were executed using both the proposed automated verification framework and the traditional methods, comparing their respective execution times and hardware resource utilization.

Figure 5 compares the execution time of the proposed framework with the existing testing methods. In the execution results of the 4 coupled models in Figure 5-(a), the proposed framework showed consistent and very short execution times with an average of 0.001 seconds. In contrast, Unittest showed execution times of 0.003-0.007 seconds on average, while Pytest showed 0.04-0.13 seconds. This demonstrates that the proposed system performs about 30-100 times faster than the existing methods when validating multiple models.

 doi:10.20944/preprints202503.1404.v1

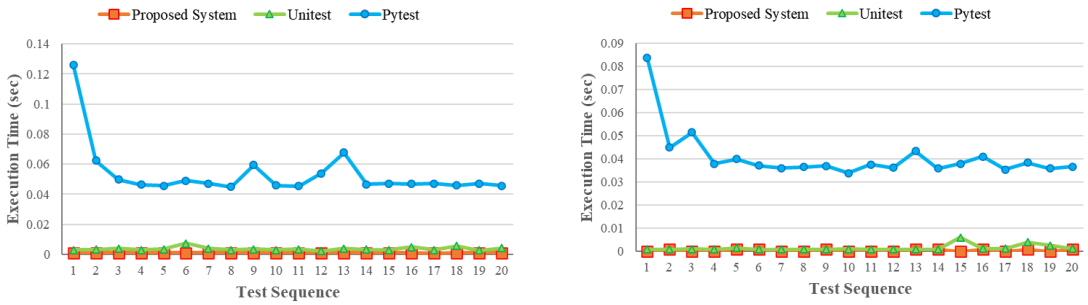

Figure 5. Execution Time Comparison of the proposed verification framework. (a) Execution time results for four coupled models, showing the performance of the proposed framework, Unittest, and Pytest; (b) Execution time results for the ACLUSTERS model, demonstrating the framework's efficiency in complex model verification.

Similar performance patterns were observed in the single execution results of the ACLUSTERS model in Figure 5-(b). Our proposed framework showed execution times of 0-0.001 seconds, Unittest showed 0.001-0.006 seconds, and Pytest showed 0.033-0.083 seconds. The superior performance of the proposed framework was proven even with the complex ACLUSTERS model. Consequently, our framework significantly outperforms the existing methods in terms of computational resource efficiency and speed in the automated validation process of DEVS coupled models.

Figure 6 shows the comparison results of CPU usage for four coupled models and the ACLUSTERS model. In the analysis of Figure 6-(a), the proposed framework exhibited an average processor load of 4.55%, with values ranging from a minimum of 0.7% to a maximum of 13.3%. Unittest showed an average of 3.95% (0.4-9.1%), while Pytest demonstrated an average of 2.67% (0.5-5.7%). Although the computational demands of the proposed framework were relatively high, it maintained a consistent pattern.

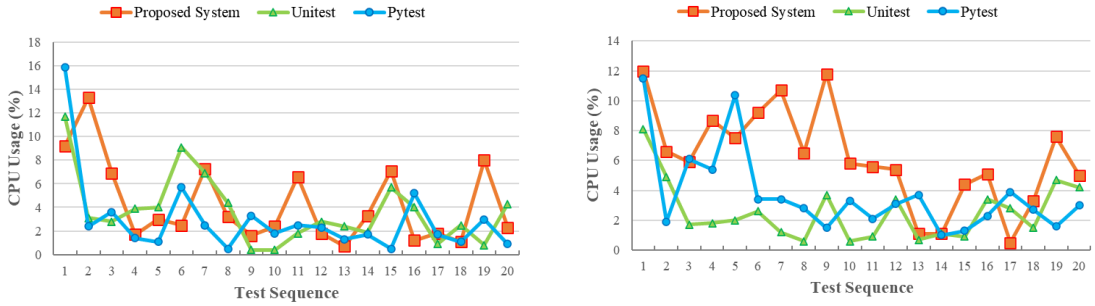

Figure 6. CPU usage comparison of the proposed verification framework. (a) Processor load analysis for four coupled models, showing variations in system performance across different testing methods; (b) CPU consumption results for the ACLUSTERS model, illustrating the framework's resource utilization in complex model verification.

In Figure 6-(b), our framework registered an average processor consumption of 6.26%, ranging from a minimum of 0.5% to a maximum of 12%. Unittest showed an average of 2.55% (0.6-8.1%), while Pytest showed an average of 3.64% (1-11.5%). The proposed system also showed higher processing requirements in the ACLUSTERS model but demonstrated stability in verifying complex models. Therefore, the proposed framework exhibited somewhat higher resource consumption in terms of CPU performance.

Figure 7 compares the memory usage of the proposed validation framework with the existing testing methods. In the test results of four coupled models in Figure 7-(a), the proposed framework used approximately 57.6-59.8MB of memory, while Unittest used about 22.4-23.4MB, and Pytest used

 doi:10.20944/preprints202503.1404.v1

about 33.0-35.5MB. It is noteworthy that although the proposed system showed the highest memory usage, it maintained a consistent memory usage pattern after initial stabilization during the 20 test sequences.

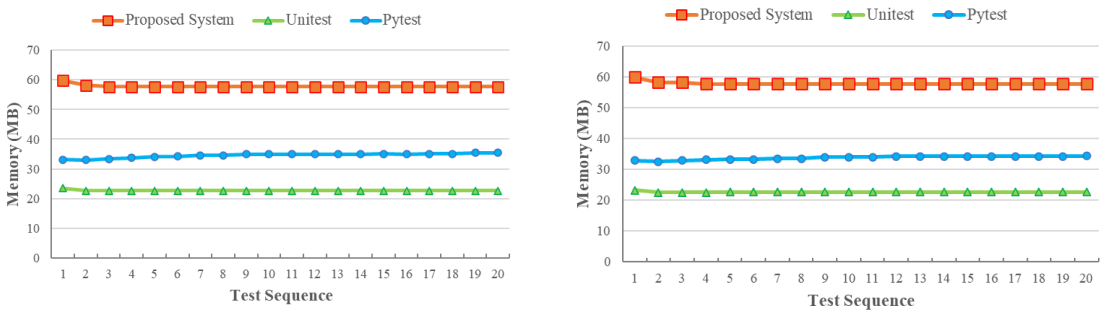

Figure 7. Memory usage comparison of the proposed verification framework. (a) Memory consumption analysis for four coupled models, demonstrating consistent memory allocation patterns across different testing methods; (b) Memory usage results for the ACLUSTERS model, highlighting the framework's resource characteristics in complex model verification.

Similar patterns were observed in the single execution results of the ACLUSTERS model in Figure 7-(b). The proposed framework used approximately 57.7-59.9MB, Unittest used about 22.5-23.2MB, and Pytest used about 32.5-34.2MB of memory. All testing methods showed a tendency for memory usage to stabilize after the initial 1-3 tests. Consequently, while our framework shows higher memory usage than the existing methods, this can be viewed as a trade-off for the significantly faster execution time shown in Figure 5. Additionally, the characteristic of maintaining consistent memory usage regardless of the number of DEVS coupled models provides the advantage of enabling predictable resource allocation in large-scale model verification scenarios.

Therefore, our proposed verification framework executes 30-100 times faster than the existing methods, despite its higher memory and CPU requirements. These increased resource demands are counterbalanced by the framework's significantly improved speed and consistent performance when verifying multiple models simultaneously. In large-scale DEVS coupled model verification environments, our solution offers efficiency through both predictable resource allocation and rapid verification processes.

6. Conclusion and Future Work

In this study, we proposed an automated verification framework for DEVS coupled models in DEVS-Python for computational resource efficiency. Experimental results showed that the proposed framework demonstrated performance improved by approximately 30-100 times in execution time compared to traditional unit testing methods (Unittest, Pytest), recording a very short average execution time of 0.001 seconds. However, memory usage of the proposed framework was higher at approximately 57.6-59.8MB, compared to Unittest (about 22.4-23.4MB) and Pytest (about 32.5-35.5MB). In terms of CPU utilization, the proposed framework showed an average of 4.55-6.26%, which was higher than Unittest (2.55-3.95%) and Pytest (2.67-3.64%) in some scenarios. This increase in resource usage can be offset by the advantages of providing significant improvement in execution speed and consistent performance in multi-model verification. Especially in large-scale DEVS coupled model verification environments, the proposed framework can be utilized as an efficient model verification solution through predictable resource allocation and fast verification speed. In future research, we plan to improve the performance of the proposed framework through additional experiments on more complex coupled models and improvements in resource usage efficiency.

 doi:10.20944/preprints202503.1404.v1

Author Contributions: Conceptualization, G.L. and S.M.N.; supervision S.M.N.; software G.L.; investigation, formal analysis, G.L. and S.M.N.; writing—original draft preparation, G.L. and S.M.N.; writing—review and editing, G.L. and S.M.N. All authors have read and agreed to the published version of the manuscript.

Funding: This research was supported by Development of Proactive Crowd Density Management Platform Based on Spatiotemporal Multimodal Data Analysis and High-Precision Digital Twin Simulation Program through the Korea Institute of Police Technology (KIPoT) funded by the Korean National Police Agency & Ministry of the Interior and Safety* (RS-2024-00405100).

Data Availability Statement: Data of this research is available upon request via corresponding author.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Zeigler, B.P. Hierarchical, modular discrete-event modelling in an object-oriented environment. Simulation 1987, 49, 219-230.
2. Zeigler, Bernard P., Tae H. Cho, and Jerzy W. Rozenblit., A knowledge-based simulation environment for hierarchical flexible manufacturing. IEEE Trans. Syst. Man Cybern. Part A 1996, 26.
3. Kim, T.G.; Zeigler, B.P. The DEVS formalism: hierarchical, modular systems specification in an object oriented framework. In Proceedings of the 19th Conference on Winter Simulation, Atlanta, GA, USA, 1 December 1987; pp. 559-566.
4. Kim, T.; Kim, H.-J. DEVS-based experimental framework for blockchain services. Simul. Model. Pract. Theory 2021, 108, 1-20.
5. Cho, T. ST-DEVS: A Methodology Using Time-Dependent-Variable-Based Spatiotemporal Computation. Symmetry 2022, 14, 912.
6. Kim, T.G.; et al. DEVSim++ Toolset for Defense Modeling and Simulation and Interoperation. J. Def. Model. Simul. 2010, 8, 129-142.
7. Nam, S.M.; Cho, T.H. Context-Aware Architecture for Probabilistic Voting-based Filtering Scheme in Sensor Networks. IEEE Trans. Mob. Comput. 2017, 16, 2751-2763.
8. Wainer, G.; Glinsky, E.; Gutierrez-Alcaraz, M. Studying performance of DEVS modeling and simulation environments using the DEVStone benchmark. Simulation 2011, 87, 555-580.
9. Nam, S.M.; Kim, H.-J. WSN-SES/MB: System Entity Structure and Model Base Framework for Large-Scale Wireless Sensor Networks. Sensors 2021, 21, 430.
10. Nam, S.M.; Cho, T.H. Discrete event simulation–based energy efficient path determination scheme for probabilistic voting–based filtering scheme in sensor networks. Int. J. Distrib. Sens. Netw. 2020, 16.
11. Concepcion, A.I.; Zeigler, B.P. DEVS formalism: A framework for hierarchical model development. IEEE Trans. Softw. Eng. 1988, 14, 228-241.
12. Zeigler, B.P. DEVS-SCHEME: A LISP-based environment for hierarchical, modular discrete event models. Technical Report; Department of Electrical and Computer Engineering, University of Arizona: Tucson, AZ, USA, 1986.
13. Aiken, M.W.; Hayes, G.J. A DEVS-Scheme simulation of an electronic meeting system. Appl. Syst. Simul. Dev. 1989, 20, 31-39.
14. Kim, T.G. Hierarchical development of model classes in the DEVS-scheme simulation environment. Expert Syst. Appl. 1991, 3, 343-351.
15. Sarjoughian, H.S.; Zeigler, B. DEVSJAVA: Basis for a DEVS-based collaborative M&S environment. Simul. Ser. 1998, 30, 29-36.
16. Arizona Center for Integrative Modeling and Simulation. DEVSJAVA. 2001. Available online: https://acims.asu.edu/ (accessed on 8 March 2025).
17. Nam, S.M. DEVS-Python. Available online: https://github.com/sumannam/DEVS-Python (accessed on March 8, 2025).
18. Nam, S.-M. Script-based Test System for Rapid Verification of Atomic Models in Discrete Event System Specification Simulation. J. Korea Soc. Comput. Inf. 2022, 27, 101-107.

 doi:10.20944/preprints202503.1404.v1

19. Nutaro, J. ADEVS. Available online: https://web.ornl.gov/~nutarojj/adevs/ (accessed on 7 February 2022).
20. Van Tendeloo, Y.; Vangheluwe, H. An evaluation of DEVS simulation tools. Simulation 2017, 93, 103-121.
21. McLaughlin, M.; Sarjoughian, H. Developing Test Frames for DEVS Models: Black-Box Testing with White-Box Debugging. Technical Report, 2020.
22. McLaughlin, M.B.; Sarjoughian, H.S. DEVS-scripting: a black-box test frame for DEVS models. In Proceedings of the 2020 Winter Simulation Conference (WSC), Online, 14-18 December 2020; pp. 2196-2207.
23. Thimothé, V.; Capocchi, L.; Santucci, J.F. DEVS models design and test using AGILE-based methods with DEVSimPy. In Proceedings of the 26th European Modeling and Simulation Symposium (Simulation in Industry)(EMSS), Bordeaux, France, 10-12 September 2014; pp. 563-569.
24. Alshareef, A.; Sarjoughian, H.S. DEVS specification for modeling and simulation of the UML activities. In Proceedings of the Symposium on Model-driven Approaches for Simulation Engineering, Virginia Beach, VA, USA, 23-26 April 2017; pp. 1-12.
25. Zeigler, B.P. Object-oriented simulation with hierarchical, modular models: intelligent agents and endomorphic systems; Academic Press: San Diego, CA, USA, 2014.
26. Chow, Y.W.; Di Grazia, L.; Pradel, M. PyTy: Repairing Static Type Errors in Python. In Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, Lisbon, Portugal, April 2024; pp. 1-13.
27. Python Software Foundation. "difflib"—Helpers for computing deltas. 2018. Available online: https://docs.python.org/3/library/difflib.html#module-difflib (accessed on 8 March 2025).
28. Python Software Foundation. unittest. Available online: https://docs.python.org/3/library/unittest.html (accessed on March 8, 2025).
29. Krekel, H. pytest. Available online: https://docs.pytest.org/en/stable/index.html (accessed on March 8, 2025).