Proceedings of the 2024 Winter Simulation Conference
H. Lam, E. Azar, D. Batur, S. Gao, W. Xie, S. R. Hunter, and M. D. Rossetti, eds.

TESTING METHODOLOGY FOR DEVS MODELS IN CADMIUM

Curtis Winstanley[1] and Gabriel Wainer[1]

[1] Dept. of Systems and Computer Eng., Carleton University, Ottawa, ON, CANADA

ABSTRACT

The practice of testing in modeling and simulation software development can be a very lengthy and tedious process but is arguably one of the most important phases in the software development lifecycle. As the complexity of a simulation model increases, so does the amount of testing to thoroughly verify and validate and to achieve adequate quality assurance of the software. This paper introduces a testing framework that is used to assist in proving the validity of DEVS atomic models in the open-source simulation tool Cadmium. Furthermore, this framework utilizes the ChatGPT Application Programming Interface (API) to help lighten the workload involved in testing those DEVS atomic models. We show how to use the framework using the Cadmium simulator to show the effectiveness of the framework.

2024 Winter Simulation Conference (WSC) | 979-8-3315-3420-2/24/$31.00 ©2024 IEEE | DOI: 10.1109/WSC63780.2024.10838817

1 INTRODUCTION

Modelling and simulation are powerful concepts used across a broad range of professions. The ability to take a complex system or real-world process and model it for computer assisted simulations is cost effective solution that allows researchers to gain insight into outcomes, design more efficient systems, and saving resources by removing the need for expensive and time-consuming real-world experiments. Specialized software packages and tools are crucial in this domain as they give users access to intuitive environments for creating detailed simulations.

In order to determine the quality of software, one needs to use rigorous and well-designed software testing strategies. There are many ways to go about applying different testing techniques, but a common factor between all of them is that for better product quality assurance, more tests are required. More tests mean longer periods of time for testing which then increases the costs to achieve the desired quality assurance. For this reason, when developing models for simulation there should be a reasonable amount of effort to automate testing and testing procedures as much as possible.

Automated testing generally includes two important steps for efficient and reliable tests. One step is designing the testing environment which prepares a controlled setting specific to the objectives of the tests (Berner et al. 2005). The test environment will allow the tester to expose the software under test to certain conditions to be able to observe the behavior of the software or even interactions with other systems. Furthermore, observing the behavior in a controlled environment allows for easier fault detection. Reproducing conditions in a controlled setting takes away from the complexity of trying to recreate and diagnose faults in a program's global scope.

Another step is providing the test environment with test cases to use for testing. A significant amount of time spent during software testing is the process is designing/maintaining test cases. Designing even a single test case can take a lot of effort as generally an effective test case should exercise code that has not yet been covered. Covering all execution paths of a complex system is in most cases infeasible so designing a set of effective test cases that satisfy the required testing criterion is a timely process. Furthermore, as the complexity of a system increases, more tests must be designed or modified to address these changes (Battina 2019).

Particularly, software that implements Discrete Event System Specification formalism (DEVS) takes on an added layer of complexity when testing. DEVS systems work using continuous time and are provided inputs through discrete events (Zeigler et al. 2000). To approach testing such systems, one must account for timing and sequencing of events to verify and validate a system's behavior. The requirements to effectively test a DEVS system can then become large depending on the system's complexity which shows the need for automating the testing process.

When testing a system component, the structure of the test can be broken down into 3 main components: *stub*, *oracle*, and *driver* (Mouchawrab et al. 2005). A common type of testing in software development that uses this test structure is integration testing. Applying integration testing comes with various advantages such as making the testing environment modular and flexible. The modularity allows for each component to be isolated and tested separately from the overall program (Lemos et al. 2009). As DEVS is by nature a modular formalism, applying integrating-based testing can be beneficial. A stub mimics the behavior of the system in a restrained fashion to provide the test with inputs in an isolated manner. Isolation of a component will ensure the functionality of the component itself is tested as there is no interference from the other components (Spadini et al. 2019). The oracle observes the behavior and outputs of the component under test and decides on whether the test passed or failed based off some provided criteria. To launch the tests, a driver is used to prepare the test environment and provides it with test cases.

The test cases that can be provided are traditionally designed by someone who understands the system being tested. Due to an explosion of advancements in artificial intelligence in recent years, Artificial Intelligence (AI) could be provided some description of the component being tested to generate test cases for us. This description can be in the form of code, a qualitative summary, or both (Alagarsamy et al. 2024). Utilizing AI for test case generation would free up a large portion of time that would be spent making test cases.

The intention of this research is to introduce a testing method and a prototype implementation that can be used to test DEVS models and systems. We use integration testing strategies (Mouchawrab et al. 2005) to design a simple yet robust testing environment using DEVS models. To populate our testing environment with data, the ChatGPT API (OpenAI 2024b) is given a description of the model, and its response is parsed and organized into test cases. The test case generation using ChatGPT is exclusively for DEVS atomic models as at present we have not researched a method for testing for coupled models. This is due to coupled models being of higher complexity as compared to atomic models since they are made up of one or more atomic models. The DEVS models used in this framework are implemented in the C++ library Cadmium (Cárdenas and Trabes 2024), but the ideas can be extended using other libraries that implement a DEVS framework.

The following sections are organized as such: Section 2 explores other work like the work done in this paper that may provide more insight into the methodology presented. Section 3 introduces the testing environment designed for DEVS and how ChatGPT was used to provide the test environment with test cases. Section 4 provides a case study to show how the framework is used and the results of using the framework. Section 5 concludes the paper discussing future work and other ideas that can be expanded upon in the framework.

2 RELATED WORKS

DEVS is a framework designed to provide users with a hierarchical and modular way to design systems and run simulations (Zeigler et al. 2000). It allows for the complexity of systems to be broken down into atomic models. The reverse is also possible where atomic models or even coupled models can be combined into coupled more complex coupled components. This makes it a very powerful tool when used in the field of modelling and simulation.

Labiche and Wainer (2009) take verification and validation strategies and apply them to DEVS. This led them to introduce a testing environment for testing DEVS models by taking advantage of the DEVS formalism. By doing this they were able to create a testing infrastructure by creating a set of DEVS models to generate test data and observe test outputs.

A paper that introduces a verification and validation framework for DEVS is from McLaughlin et al. (2020). The authors provide a black box testing framework that uses similar strategies as Labiche and Wainer (2009) by using the DEVS formalism to create their testing environment. They utilize the concept of *Experimental Frame* (EF) introduced by Zeigler et al. (2000) to define the structure of their test environment. From the EF they create three models they use to construct their environment: *generator*, *acceptor*, and *transducer*. The generator provides inputs to the model under test, acceptor differentiates between transitional and steady states of the model, and the transducer assesses the outcome of the test based on the outputs and inputs.

The concept of EF is explored and further developed by Traoré et al. (2006). The authors take the idea of the EF and translate it into an experimentation framework. They connect the input and output ports of the model under test to the frame which internally contains a generator, acceptor, and transducer model. The generator provides the experiment with inputs, the transducer takes the outputs from the model under experiment and reports the acceptor of execution times and errors which the accepter will then decide on the success of the experiment. Work done by Van Acker et al. (2024) extends on some of the work done by Traoré et al. (2006) as the authors introduce Validity Frames (VFs). These VFs are meant to capture any aspect that can impact the validity of a model throughout the engineering process of the model.

Jaffari et al. (2020) put forward a model-based approach to automating test case generation by search-based algorithms. They would take UML model diagrams and convert them into an XML format which they would parse into an *ElementTree*. The *ElementTree* was then passed into their search-based algorithm to generate test cases based off data flow criteria. This research intends to use a similar approach to automating test case generation where we provide artificial intelligence a data serialization format of DEVS atomic models.

Work done by Ansari et al. (2017) shows how Natural Language Processing (NLP) can be used to generate test cases. NLP is a subfield of computer science that deals with bridging the gap between human and computer communication (Chowdhary 2020). They show that you can organize the functional requirements of your applications as the input to the computer or AI. The functional requirements will be analyzed by the NLP and based on the requirements test cases will be created. For creating the most effective set of tests, the functional requirements should be made up of key words and very unambiguous statements. Large language models (LLMs) that utilize NLP that receive inputs with ambiguities could make inaccurate assumptions when analyzing what it received which could make following responses invalid (Kuhn et al. 2020). Alagarsamy et al. (2024) show how text to test case generation can be done using LLMs using a prompt template for querying ChatGPT for a consistent response structure. This allows them to have an easier time processing responses into usable test cases.

The purpose of this research is to be able to test the atomic components of a complex DEVS system. Such a system shown by Horner et al. (2022) is a supervisory controller designed by the authors for a Bell 412 helicopter. The supervisory controller is used to assist in orchestrating the autonomy of the aircraft and contains many atomic components that must be verified and validated. Automating this process would become very useful as the supervisory controller evolves in complexity.

To do this, we use a similar testing environment proposed by Traoré et al. (2006) but instead of a single generator and acceptor model, we use a generator for each input port and an acceptor (we call our acceptors Comparators since the purpose of a Comparator is to “compare” data) for each output port. This allows us to have a flexible, scalable, and simple testing environment. We also combine ideas shown by Jaffari et al. (2020); Ansari et al. (2017); Alagarsamy et al. (2024) to take a description of a model and turn it into a ChatGPT query using a query template. By doing this we can use AI to generate test data for our test environment which will assist in automating the testing process.

3 A SCALABLE TESTING FRAMEWORK FOR ATOMIC DEVS MODELS

As mentioned before, we will apply the concept of integration testing hence we apply the testing structure of a stub, oracle, and driver to develop a test environment. The behaviors of the stub and the oracle will be translated into DEVS atomic models to be able to easily control the testing environment. The stub of a test is used to mimic the behavior of the system that interacts with the component under test. In this framework the “stub” will be known as a series of Generator DEVS atomic models where the number of generator models is equal to the number of input ports that the model under test has. The oracle must determine whether the test was successful or not based on the provided inputs. Two kinds of DEVS atomic models were developed in this framework to handle the task of the oracle: Comparator and Decider. The oracle in this framework is a series of Comparator models, one for each output port the model under test has, and a single Decider model.

The Generator model is a template model that is used to give the desired input data to a single port on the model under test. The Generator model is comprised of a single output port which it uses to output the data to feed to the port it is connected to. The data it outputs is provided by the driver when the tests are run. The Generator uses data in the form of a list of *time-value* pairs. The time part of the pair is the time when an internal transition will occur, and the value is what the output function will send through the output port of the Generator when the internal transition occurs. If the Generator is provided with an empty set of data or it runs out of data to send, it will set its time advance value to infinity thereby passivating the model.

To determine the success of a test, the oracle of this framework is implemented using Comparator(s) and a Decider. The role of the Comparator is like that of the Generator but instead of generating data, it waits for data and produces reports for the Decider. There is one input and one output port on the Comparator. For every output port that a model under test has, there is a Comparator connected to it. The input port will receive outputs from the model under test and in the Comparator's external transition, it will compare the output to some expected data. The expected data that the Comparator uses is provided by the driver when the tests begin. This data is in the form of *sequence number-value* pairs where the sequence number is the order in which the data is output, and the value is the expected value to be output. When a comparison is complete, the model will immediately enter an internal transition where the Comparator will 'report' its conclusion of the output it received to the Decider. The structure of the report is a *sequence number-bool* pair where the sequence number is the order in which the data is output (same as before), and the bool is a flag that indicates whether the comparison was successful or not i.e. the output value is equal to the expected value.

The final part of the oracle is in the Decider where it must make the decision of the success of a test based on some parameters. These parameters are the reports mentioned above and one or more state transition paths depending on whether the model under test is an atomic or coupled model. The Decider has a single port which is an input port that all the Comparator output ports connect to so that the Decider can gather every report that a Comparator produces. When the Decider receives a report, it saves the order in which it arrived and its comparison flag. The Decider must decide when all possible reports for a test are received. Since DEVS models are black boxes, then we do not know when the model under test will finish and passivate for a particular test. What we do know is that for a model to passivate, its time advance must be infinity, and remain at infinity. This will tell the Cadmium simulator that the model will no longer have any active part in the simulation. Therefore, every time we receive a report, we set the time advance of the Decider to the boundary of infinity. Once the last possible report has been received the time advance will be set (to the boundary of infinity) and the internal transition will be carried out, which is when the decision of the test will be made. Since these tests are done in simulated time and not real time, we do not actually wait until the boundary of infinity to decide but only a few microseconds.

An overview of how the framework is built using a DEVS model diagram is shown in Figure 1.

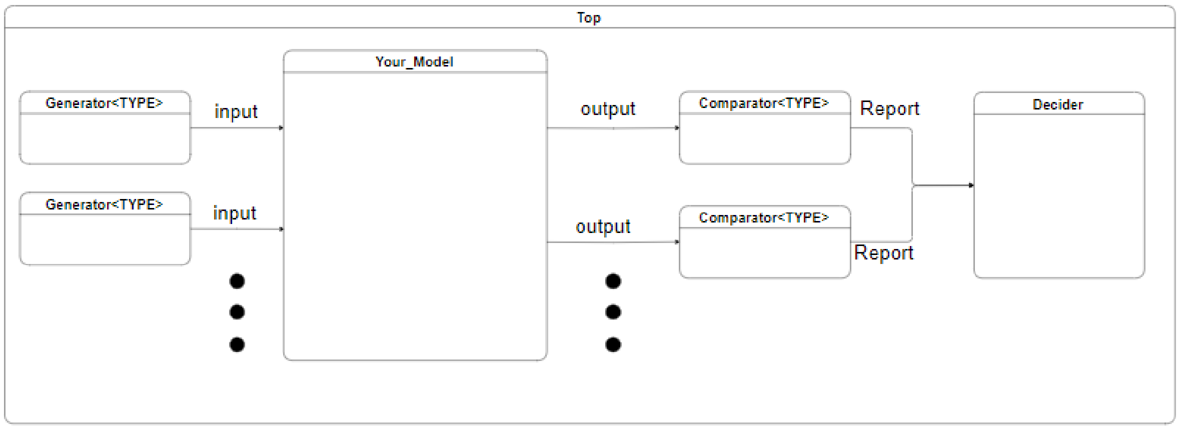


Figure 1: Proposed test environment using DEVS models.

Figure 1 shows the framework model layout for a generic DEVS model that we are testing. Suppose that the model under test named “Your Model” has n input ports and k output ports where n and k are integers. Then the framework would have n Generators and k Comparators and the data type these models are instantiated with would match to the type of the port they were instantiated for. The k Comparators will have all their output ports coupled to the single input port of the Decider. Everything is encapsulated in a “Top” coupled model so that the model environment can be instantiated in the driver for simulations to be run. Since this paper uses Cadmium, these models are C++ classes, the drivers used for the framework are C++ files that instantiate the Top model. The driver will provide the Top model with the data sets for the Generators, Comparators, and Decider through the Top model’s constructor. The Top model will then sort this data and provide each Generator its data to generate, each Comparator its data to compare, and the Decider the state transition paths to check.

3.1 ChatGPT Integration

To address the test case automation, its best to begin by understanding how much data we need for each test. As discussed earlier, each Generator needs a set of *time-value* pairs, each Comparator a set of *sequence number-bool* pairs, and the Decider a state transition path. Utilizing the ChatGPT API, we must convey the model information such as its state transitions and ports, and then ask it to provide us with test case data for the Generators, Comparators, and the Decider. A high-level overview of the steps taken to use ChatGPT to do this is shown in Figure 2.

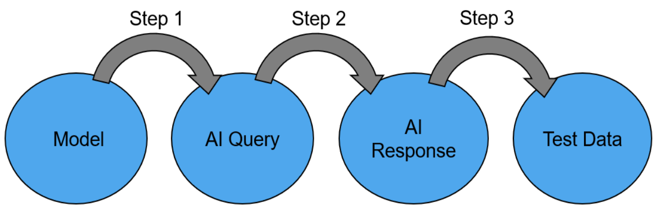


Figure 2: Steps to go from model to test data.

The first step in Figure 2 is to have a model definition which can be in the form of DEVS model diagram or DEVS specification and to take that model definition and use it to produce a message we can use to query the ChatGPT API. To try to get the most consistent response from the AI, a message template using JSON was created and can be seen in Appendix A. The key elements that a message must have, are:

- model struct types: a section specifying non-built-in data types that the model ports may use.

- functions used in model: a section specifying functions the model uses.
- state variables: a section specifying the state variables of the model.
- input ports: a section specifying the input ports of the model.
- output ports: a section specifying the output ports of the model.
- transitions: a section describing all the different transitions of the model.
- question: a section that requests for the AI to provide us with a number test cases in a specific format.

By using the JSON format for the message, it should be easy for the AI to understand due to its clear structure. ChatGPT 4 is a Large Language Processing (LLP) model which means any requests to is API will involve parsing and analyzing. Therefore, the clearer the request is, the more consistent and correct the response will be.

The format of the JSON message template was designed in such a way to show the control flow of the atomic DEVS model while minimizing the length of the overall message by excluding as much code as possible. There are some parts of the template where code or pseudo-code must be included as we need the AI to know what functions and types we use throughout the model. However, the bulk of the code which is the class (C++ class in the case of Cadmium) that defines the entire atomic model is not included. If we were to include the class that defines the atomic model we are testing, then the message would become far too large for a reasonable query to the ChatGPT API to expect a reasonable response. For large messages, ChatGPT has been known to not fully process or even ignore large messages.

Step 2 does involves sending the request and receiving the response; this is mostly done by the ChatGPT API. Before we send a message, there are some useful configuration options we can use to get the best response. The first option is choosing what ChatGPT model we would like to query. For our prototype implementation, the ChatGPT model chosen was the gpt-4-0125-preview model. This is due to this model being a newer version of GPT4 and is designed to reduce "laziness" in the responses (OpenAI 2024a). Laziness refers to cases when there are large requests (such as our JSON messages), and the model may skip over detailed information to produce responses faster that are of less quality. Another parameter we can set in a request is the 'temperature' of the model when it produces a response. The temperature impacts the randomness of the response where the value can be set from 0 to 1 and the lower the temperature, the more deterministic the model will be. Randomness in a response comes in the form of creativity and human like speech elements to make answers to the same or similar questions different. For the purposes of this prototype implementation a low temperature value of 0.15 was used as it is best to avoid the AI becoming too creative when formulating responses to make parsing easier (OpenAI 2024b).

The step 3 in Figure 2 is taking the ChatGPT response and parsing it to be used as test case data for use in the testing framework. The parsed data will be written as C++ code to a header file that can be included in the driver. When we parse the response, we also apply error checking and correcting methods. A common issue when developing and testing the effectiveness of the JSON message provided to ChatGPT is that when we receive a response, there are many times where the sequence numbers of the expected outputs may be partially incorrect. For example, there are times where instead of giving us sequence numbers, the AI would give us expected times of the outputs. On other occasions the sequence numbers may not start from 0 but from some other number. This is not a problem because when we apply error correcting methods, we check in case the sequence numbers may be times or offset values and if they are then we reorder them in ascending order starting from 0. Making these corrections will then make the expected outputs data given to us by ChatGPT valid for use in the Comparators.

The driver will provide the Top model its necessary data to populate the Generators, Comparators, and the Decider for each simulation test. Figure 3 shows a general summary of how the testing framework uses ChatGPT for the test cases.

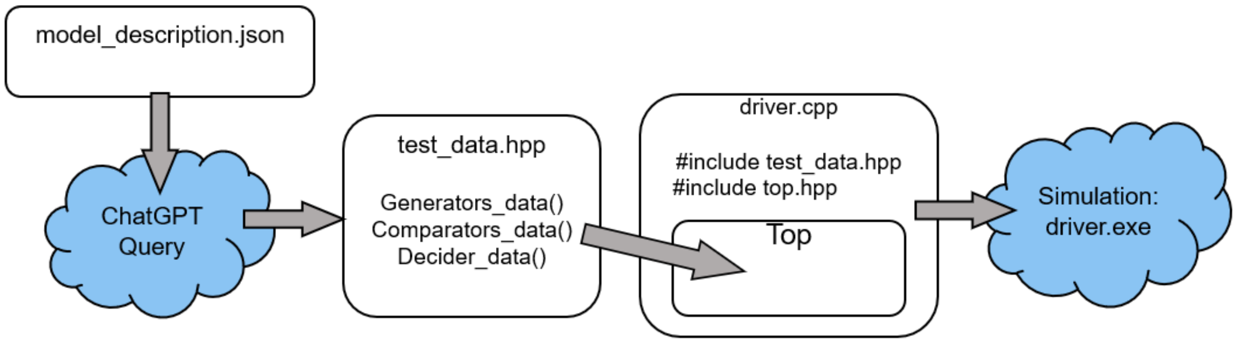


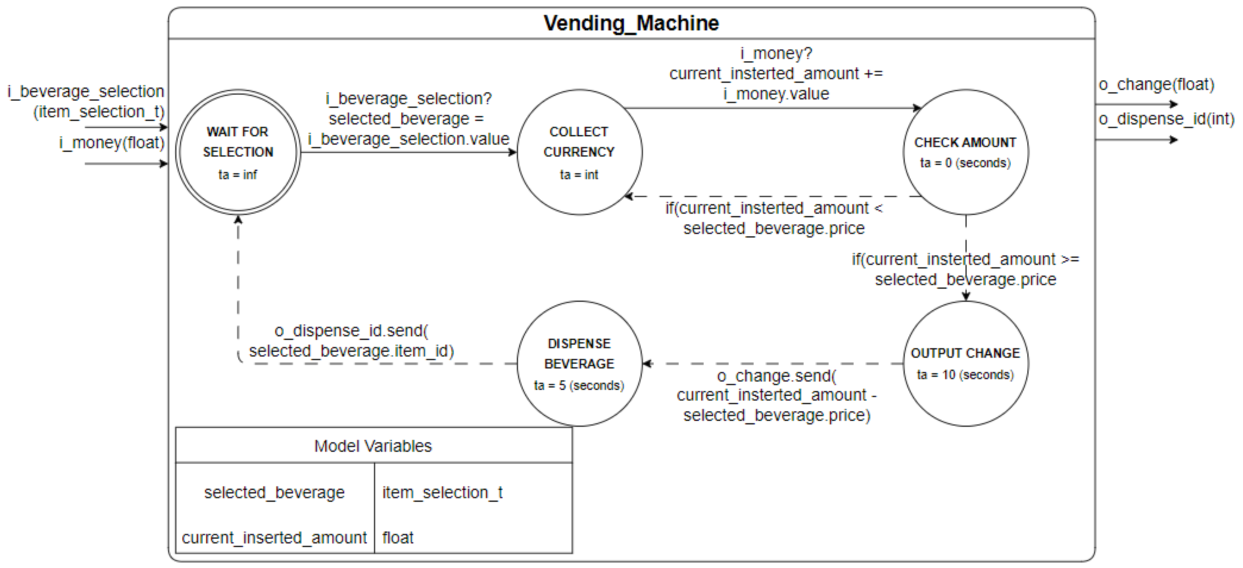
Figure 3: Diagram depicting how the test data populates the test environment for test simulations.

Figure 3 shows how the work shown in section 3 comes together for the prototype implementation. The test data created using ChatGPT, which was discussed earlier in section 3.1, is used in the Top model which is the test environment shown in Figure 1.

4 CASE STUDY

This section will introduce a DEVS atomic model that will be tested using the testing framework discussed in the previous sections. The model chosen for this case study will be a model that mimics the basic behavior of a vending machine.

A customer will select a beverage they would like to purchase and inserts their money. If the money inserted into the machine is not sufficient, then the machine will wait for more money to be added. Once enough money has been acquired by the machine it will calculate the change to be deposited and it will deposit the change. After returning the change to the customer, the beverage purchased will be dispensed. With this model description in mind, the corresponding DEVS created is shown in Figure 4.


Vending_Machine
i_beverage_selection (item_selection_t)
i_money(float)
o_change(float)
o_dispense_id(int)
WAIT FOR SELECTION ta = inf
COLLECT CURRENCY ta = int
CHECK AMOUNT ta = 0 (seconds)
OUTPUT CHANGE ta = 10 (seconds)
DISPENSE BEVERAGE ta = 5 (seconds)
i_beverage_selection? selected_beverage = i_beverage_selection.value
i_money? current_insterted_amount += i_money.value
if(current_insterted_amount < selected_beverage.price
if(current_insterted_amount >= selected_beverage.price
o_change.send(current_insterted_amount - selected_beverage.price)
o_dispense_id.send(selected_beverage.item_id)

Model Variables	
selected_beverage	item_selection_t
current_inserted_amount	float

Figure 4: DEVS atomic model diagram of a generic vending machine.

The model has five states: WAIT_FOR_SELECTION, COLLECT_CURRENCY, CHECK_AMOUNT, OUTPUT_CHANGE, DISPENSE_BEVERAGE. Each of these states has a *ta* element which represents the time advance of the state. There are two state variables the model uses to store incoming data: *selected_beverage* and *current_inserted_amount*. The *selected_beverage* variable is used to store the customers selection and is of type *item_selection_t* which is a C++ structure of two elements: *price* of type float and *item_id* of type int. The *current_insterted_amount* is the amount of money the customer has inserted into the vending machine and is of type float. *i_beverage_selection* and *i_money* are the two input ports that the model uses to receive beverage selections and the amount of inserted money. *o_change* and *o_dispense_id* are the output ports that send the amount of change to be returned to the customer after the purchase and the ID of the beverage that needs to be dispensed, respectively.

The solid black lines represent external transitions while the dotted black lines represent internal transitions. The behavior of these transitions is described by the pseudo-code next to them. In general, the model begins with the initial state of WAIT_FOR_SELECTION and when it receives a selection through the *i_beverage_selection* port it will advance to the COLLECT_CURRENCY state. In COLLECT_CURRENCY we wait for the customer to insert money into the vending machine and the amount will come through the *i_money* port. We will then transition to CHECK_AMOUNT and immediately check the amount that was provided was enough for the purchase. If the amount was not sufficient, we will return to the COLLECT_CURRENCY state. If it was sufficient then we will immediately transition to the OUTPUT_CHANGE state. After ten seconds we will then transition to DISPENSE_BEVERAGE and output the amount of change to customer should receive through *o_change*. After five more seconds the model will output the id of the item to be dispensed through *o_dispense_id* and it will transition back to the beginning idle state.

To test this model using the framework presented in section 3, we begin by configuring the test environment by applying Generators, Comparators, and a Decider models to the vending machine model. This configuration can be seen in Figure 5.

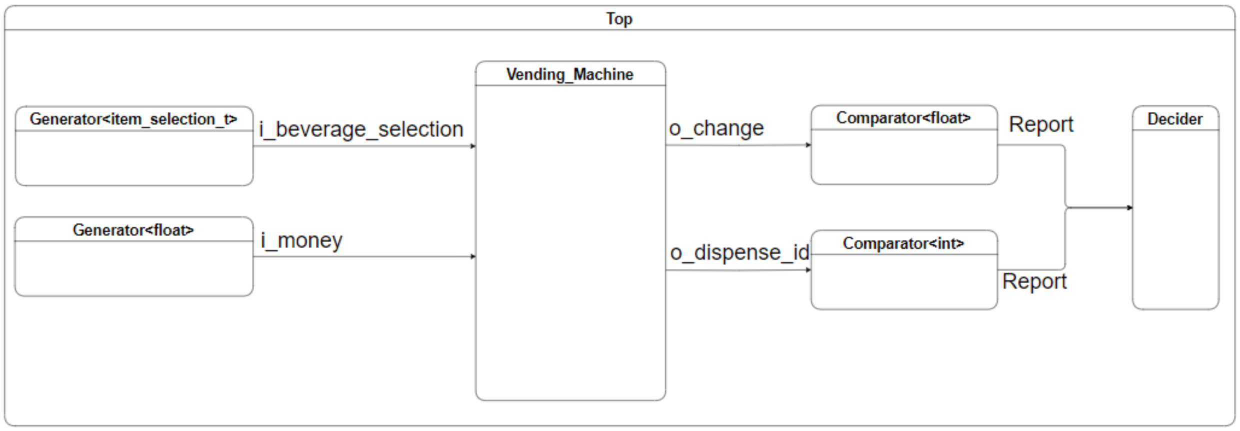


Figure 5: The Vending Machine model prepared in the test environment.

In the Top model we add two Generator models, one connected to each input port of Vending Machine. We also add two Comparator models to connect to each output port of Vending Machine and connect the output ports of each Comparator to the Decider model. With this test environment set up in the Top model, we can now instantiate this in a driver C++ file *driver.cpp*.

Although our test environment is ready, we need to populate our Generators, Comparators, and Decider with test data to run simulation tests. For this we will use ChatGPT 4 to generate three test cases or in other words, three sets of data for the Generators, Comparators and Decider. Using this data, we will be able to run three test simulations and we can observe how capable ChatGPT is at creating test cases.

We need to provide a JSON file that the AI will use to understand our model and provide us with test data. The JSON provided to ChatGPT for this case study will ask the AI for three test cases and can be seen in Appendix B. When we query ChatGPT and receive the response, we will parse it and a test data header file is created and used for the test environment in Figure 5. Table 1 shows a summarized response when asking ChatGPT for test cases.

Table 1: Summarized ChatGPT response.

TEST CASES	Input Data	Expected Outputs	State Transitions

1	i_beverage_selection
	{(0), [5.00, 1]}
	i_money
	{(1), [5.00]}
		o_change
		{(0), [0.00]}
		o_dispense_id
		{(1), [1]}
			{WAIT_FOR_SELECTION,
			COLLECT_CURRENCY,
			CHECK_AMOUNT,
			OUTPUT_CHANGE,
			DISPENSE_BEVERAGE,
			WAIT_FOR_SELECTION}

2	i_beverage_selection
	{(0), [3.50, 2]}
	i_money
	{(1), [5.00]}
		o_change
		{(0), [1.50]}
		o_dispense_id
		{(1), [2]}
			{WAIT_FOR_SELECTION,
			COLLECT_CURRENCY,
			CHECK_AMOUNT,
			OUTPUT_CHANGE,
			DISPENSE_BEVERAGE,
			WAIT_FOR_SELECTION}

3	i_beverage_selection
	{(0), [4.00, 3]}
	i_money
	{(1), [2.00]}, {(2), [2.50]}
		o_change
		{(0), [0.50]}
		o_dispense_id
		{(1), [3]}
			{WAIT_FOR_SELECTION,
			COLLECT_CURRENCY,
			CHECK_AMOUNT,
			COLLECT_CURRENCY,
			CHECK_AMOUNT,
			OUTPUT_CHANGE,
			DISPENSE_BEVERAGE,
			WAIT_FOR_SELECTION}

Test Case 1 and Test Case 2 are similar in which they express the same set of state transitions in the Vending Machine model. Looking specifically at Test Case 1, the user selects the beverage at 0 seconds. This beverage has a price of $5.00 and an item ID of 1. This will transition our model from WAIT_FOR_SELECTION to COLLECT_CURRENCY. The user then inserts $5.00 at 1 second which s the exact amount of money needed to purchase the beverage. This will transition the state of the model from COLLECT_CURRENCY to CHECK_AMOUNT. Since the user inserted the exact amount needed the state will then transition to OUTPUT_CHANGE. OUTPUT_CHANGE will produce the first output (sequence number 0) which is the calculated amount of change. The output will be sent through *o_change* which is $0.00 and the model transitions to DISPENSE_BEVERAGE. DISPENSE_BEVERAGE will produce the next output of the model (sequence number 1) by sending the ID of 1 through *o_dispense_id* and the model will transition back to is original state of WAIT_FOR_SELECTION.

Test Case 3 is slightly more intricate as instead of inserting a sufficient amount of money the first time, the user makes two payments. When the model transitions to COLLECT_CURRENCY after a selection was made, the user at 1 second inserts $2.00 which will transition the model to CHECK_AMOUNT. The model will notice that necessary amount to buy the beverage was not given hence it will immediately transition back to COLLECT_CURRENCY. At 2 seconds the user inserts their second payment of $2.50 and the model will transition back to CHECK_AMOUNT where it will proceed with outputting the change and the id of the item to dispense.

Based of the explanation of the test cases above, the tests ChatGPT generated above are all valid in the sense that the inputs reflect the expected outputs and corresponding state transition path. Furthermore, we ran nine more test case generations to see if the AI generated expected outputs and state transitions were consistently valid based off the generated inputs. These queries will use the exact same JSON that was used to generate the response for Table 1. The summarized results of the validity of the responses are shown in Table 2 but due to the length constraints, the full responses can be seen in Appendix C.

Table 2: Validity results of ChatGPT responses.

Generated Tests Response Number	Valid Number of Expected Outputs	Valid Number of State Transition Paths	Notes
1	3/3	3/3	Same response as used for the discussion above in Table 1.
2	3/3	3/3	
3	3/3	3/3	
4	1/3	3/3	Response was missing expected outputs for the *o_change* port.
5	3/3	3/3	
6	3/3	3/3	
7	3/3	3/3	
8	1/3	3/3	Response was missing expected outputs for the *o_change* port.
9	1/3	3/3	Response was missing expected outputs for the *o_change* port.
10	2/3	2/3	Calculation error made when calculating total money inserted.

Table 2 suggests that the AI has a solid understanding of the control flow of the Vending Machine model. Out of 30 test cases generated in total, only one test case had a problem with the state transition paths due to a calculation error made when computing the amount of money the user inserted. With regards to the expected outputs, out of the 30 sets of expected outputs that were generated, 7 were incorrect. One of them was due to the issue mentioned above where there was a calculation mishap. The other 6 were due to the response not outputting the calculated change of $0.00. The AI clearly assumed when producing some of the responses that if the calculated change is $0.00, then an output was not necessary.

It is important to also understand how frequently these issues occurred. 6/10 of the responses had no issue when generating the test cases. 1 test had the calculation error while the other 3 contained the incorrect change output assumptions made by ChatGPT. This shows that the mistakes made by the AI are not distributed across all the test cases but are limited to a select few. The AI assumes that the model itself is a vending machine and not modeled vending machine logic, hence it thought that if the change to return is $0.00 then there should not be an output event. It shows that even though we made our JSON request to the AI as unambiguous as we could, there is still a chance that the AI could make a silly assumption that even a human would make.

5 CONCLUSIONS

In this paper we introduced a method that can be used to assist in the verification and validation of DEVS models. We created three DEVS atomic models; Generator, Comparator, and Decider that are used for making up the test environment for the framework. To automate test cases for DEVS atomic models, we created a JSON template structure that is used to show the control flow of the model and request test cases from ChatGPT. The response received from the AI is parsed, passed through some error correction methods, and then the data is used to populate our test environment with test cases.

We then showed the use of the framework using the behavior of a generic vending machine as a case study. The vending machine atomic model was connected to the test environment and the model's control flow behavior was translated into a JSON file using the JSON template in Appendix B. ChatGPT was then queried and the response received was parsed to populate the test environment with test cases. This was repeated 10 times to see the consistency of ChatGPT's understanding of the vending machine models' control flow. The result showed that when generating test cases, the AI sometimes made incorrect assumptions when producing expected outputs but had a particularly good understanding of the state paths of the vending machine model.

In the future, we plan to develop a JSON template that can be used to describe the behavior of a DEVS coupled model in a concise but accurate manner for test case generation of coupled models.

APENDICES

The appendices are available here.

REFERENCES

Alagarsamy, S., C. Tantithamthavorn, C. Arora, and A. Aleti. 2024. "Enhancing Large Language Models for Text-to-Testcase Generation". *arXiv preprint arXiv:2402.11910.*

Ansari, A., M. B. Shagufta, A. S. Fatima, and S. Tehreem. 2017."Constructing Test Cases Using Natural Language Processing." In *Third International Conference on Advances in Electrical, Electronics, Information, Communication and Bio-Informatics (AEEICB)*, Feb 27th – 28th, Chennai, India, 95-99.

Battina, D. S. 2019. "Artificial Intelligence in Software Test Automation: A Systematic Literature Review." *International Journal of Emerging Technologies and Innovative Research, ISSN* 6(12): 1329-1332.

Berner, S., R. Weber, and R. K. Keller. 2005. "Observations and Lessons Learned from Automated Testing." In *Proceedings of the 27th International Conference on Software Engineering*, May 15th – 21st, St. Louis, USA, 571-579.

Cárdenas, R. and Trabes. G. 2024. "Cadmium 2: An Object-Oriented C++ M&S Platform for the PDEVS Formalism". https://github.com/SimulationEverywhere/cadmium_v2, accessed 9th April.

Chowdhary, K.R. 2020. *Fundamentals of Artificial Intelligence*. New Delhi: Springer.

Horner, J., T. Trautrim, C.R Martin, G. Wainer, and I. Borshchova, 2022."December. Discrete-Event Supervisory Control for the Landing Phase of a Helicopter Flight". In *2022 Winter Simulation Conference (WSC)*, 441-452, https://doi.org/10.1109/WSC57314.2022.10015393.

Jaffari, A., C. J. Yoo, and J. Lee. 2020."Automatic Test Data Generation Using the Activity Diagram and Search-Based Technique." *Applied Sciences*, 10(10): 3397.

Kuhn, L., Y. Gal, and S. Farquhar. 2022. "Clam: Selective Clarification for Ambiguous Questions with Generative Language Models". *arXiv preprint arXiv:2212.07769*.

Labiche, Y., and G. Wainer. 2005. "Towards the Verification and Validation of DEVS Models". In *Proceedings of 1st Open International Conference on Modeling & Simulation*, June 13th – 15th, Clermont-Ferrand, France, 295-305.

Lemos, O.A.L., Franchin, I.G., and Masiero, P.C. 2009. "Integration Testing of Object-Oriented and Aspect-Oriented Programs: A Structural Pairwise Approach for Java". *Science of Computer Programming* 74(10): 861-878.

McLaughlin, M. B., and H. S. Sarjoughian. 2020."DEVS-Scripting: A Black-Box Test Frame for DEVS Models". In *2020 Winter Simulation Conference (WSC)*, 2196-2207, https://doi.org/10.1109/WSC48552.2020.9384024.

Mouchawrab, S., L. C. Briand, and Y. Labiche. 2005. "A Measurement Framework for Object-Oriented Software Testability". *Information and Software Technology*, 47(15): 979-997.

OpenAI. 2024a. "GPT 4 and GPT 4 Turbo". https://platform.openai.com/docs/models/gpt-4-and-gpt-4-turbo, accessed 9th April.

OpenAI. 2024b. "API Reference". https://platform.openai.com/docs/api-reference, accessed 9th April.

Spadini, D., M. Aniche, M. Bruntink, and A. Bacchelli. 2019. "Mock Objects for Testing Java Systems: Why and How Developers Use Them, and How They Evolve". *Empirical Software Engineering* 24: 1461-1498.

Traoré, M.K. and A. Muzy. 2006. "Capturing the Dual Relationship Between Simulation Models and Their Context". *Simulation Modelling Practice and Theory*, 14(2): 126-142.

Van Acker, B., P. De Meulenaere, H. Vangheluwe, and J. Denil. 2024. "Validity Frame–Enabled Model-Based Engineering Processes". *SIMULATION*, *100*(2):185-226.

Zeigler, B.P., H. Praehofer, and T.G. Kim. 2000. *Theory of Modeling and Simulation*. New York: Academic Press.

AUTHOR BIOGRAPHIES

CURTIS WINSTANLEY is an M.A.Sc student at Carleton University. He has a bachelor's degree in Communications Engineering from Carleton University, Canada. His research interests are in automated software verification and validation, and digital twins. His email address is curtiswinstanley@cmail.carleton.ca

GABRIEL WAINER is a professor at the department of Systems and Computer Engineering at Carleton University. He received is M.Sc. (1993) from the University of Buenos Aires, Argentina, and his Ph.D (1998, highest honors) from UBA/ Université Aix-Marseille-III, France. He is a fellow of SCS. His email address is gwainer@sce.carleton.ca.