Proceedings of the 2020 Winter Simulation Conference
K.-H. Bae, B. Feng, S. Kim, S. Lazarova-Molnar, Z. Zheng, T. Roeder, and R. Thiesing, eds.

DEVS-SCRIPTING: A BLACK-BOX TEST FRAME FOR DEVS MODELS

Matthew B. McLaughlin
Hessam S. Sarjoughian

Arizona Center for Integrative Modeling and Simulation
School of Computing, Informatics, and Decision Systems Engineering
Arizona State University
Tempe, AZ 85281, USA

ABSTRACT

Experimental frames have been used in DEVS-based simulations to drive scenarios through injecting inputs and interpreting outputs. This design has traditionally called for separate models with distinct roles: generator, acceptor, and transducer. In certain controlled experiments such as model testing, sequential programming offers a simpler design with many benefits, specifically: code reduction, test case development throughput, and diagnostics for failed tests. This research offers a test framework that is derived from atomic DEVS and facilitates testing through scripting. The challenge for this research is to prove DEVS semantics are maintained when the experimental frame is tightly controlled by a script. Our solution uses a separate thread for this script and synchronizes program execution switching with a nest lock. Synchronization is key in showing that this design maintains DEVS semantics by nesting script code within the state transition functions of DEVS modeling components.

1 INTRODUCTION

Testing is a critical on-going process of software development to break the system and expose as many failures as possible toward satisfying all user requirements. To make effective use of a test developer's time, each test should be small, focused, and cover unique execution paths (Srikanth and Banerjee 2012). In black-box testing, tests are developed based on a requirements model and the inherent complexity within the requirements. Testers must also consider operational profiles, customer priority, fault proneness, and requirements volatility to improve testing efficiency and coverage.

Rich concepts, methods, processes, and frameworks help develop simulation models that can be useful for their intended purposes, e.g., Ören (1981), Balci (1994), Zeigler et al. (2000). The results of these efforts have led to model validation, verification, and testing. Considering simulation models, test cases are defined and used to test their proper behaviors. Software testing concepts and frameworks are commonly used to develop tests. The results are used for validating model structure and behavior. Tests can also be defined and used for verifying the accuracy of model transformation from one form into another.

Black-box software testing with test oracles (input, expected vs. actual output) is not original research (Peters and Parnas 1994), but the test oracle problem is increasingly more present than ever before because the complexity of some models prevent us from knowing what our expected output should be (Garousi and Felderer 2016). In simulation, we add another dimension that may also not be known, timing. Metamorphic testing is just one technique to address the test oracle problem and is usually scripted due to the complex nature of representing these types of tests.

The research in this paper proposes a robust black-box testing framework based on and for the Discrete Event System Specification (DEVS) formalism (Zeigler et al. 2000). This framework is developed for and integrated into the DEVS-Suite simulator (ACIMS 2020). The resulting simulator supports integrated *(i)*

JUnit testing with code debugging, *(ii)* black and white box component animation (I/O messages with states), and *(iii)* configurable input, output, and state trajectories for run-time visual debugging. This proposed testing framework has been designed to be scalable from unit testing for atomic models to integrated testing for modular, hierarchical coupled models specified in the DEVS formalism. It supports multiple individual test scripts using the same collection of models to manage complexity. Though this framework has a DEVS approach, its principles can be a basis for developing testing frameworks for other modular, hierarchical modeling and simulation approaches (Alur 2015).

1.1 Parallel Atomic and Coupled DEVS I/O System Formalism

The parallel DEVS formalism is a universal discrete event, modular, hierarchical set-theoretic modeling language (Zeigler et al. 2000). It supports developing models for systems-of-systems using basic atomic and coupled models. The basic atomic model is specified as $DEVS = \left(X^b, Y^b, S, \delta_{ext}, \delta_{int}, \delta_{conf}, \lambda, ta\right)$. The state S changes as a response to a received bag (or multiset) of input messages $x^b \in X^b$ via $\delta_{ext} : Q \times X^b \mapsto S$ where $Q = \{(s,e) \mid s \in S, 0 \leq e \leq ta(s)\}$. State transitions due to internal events are specified as $\delta_{int} : S \mapsto S$. The order of simultaneous δ_{ext} and δ_{int} executions is specified as $\delta_{conf} : S \times X^b \mapsto S$. For any state change, the amount of time to advance before the next internal state transition is specified as $ta : S \mapsto \mathbb{R}^+_{0,\infty}$. Output messages are specified as $y^b \in Y^b$ via $\lambda : S \mapsto Y^b$. Output messages may be produced prior to any internal state change. Every input and output is specified as a bag of messages that pair a port name and a value. The behavior of every atomic model follows an abstract execution protocol responsible for advancing a clock and updating the elapsed time e, the time period since the occurrence of the last state change.

DEVS requires that state transition functions (operations) can only generate new states based on its current internal state, elapsed time, and input. This forbids the use of shared or global resources such as the simulator clock. Any behavior on these resources would create an ambiguous state change for the parameters given to state transition functions. Any stochasticity must be seeded and maintained internal to the DEVS model to guarantee I/O modularity and repeatable behavior.

The basic coupled model is specified as $N = \left(X^b, Y^b, D, \{M_d\}, \{I_d\}, \{Z_{i,d}\}\right)$. The input X^b and output Y^b have the same specifications as those given for the basic atomic DEVS model. D specifies a list of unique names for referencing in the components of a coupled model N. M_d is either an atomic or a coupled DEVS model component. $I_d \subseteq D \cup \{N\}$ contains the names of the influencers for d. For $d \in D$, the DEVS model by this name is M_d. The coupled model has inputs and outputs which can be coupled to its immediate children (components). $\{Z_{i,j}\}$ describes the external I/O couplings (EIC/EOC) and internal couplings (IC). N satisfies closure under coupling which is essential for model reuse and allowing any number of levels in a hierarchical decomposition of systems-of-systems. The behavior of the highest level coupled model N and all of its atomic and coupled model components is according to the abstract coupled simulator and the abstract coordinator protocols.

1.2 Experimental Frame and Test Frame

The concept of *Experimental Frame* (EF) is aimed at specifying some conditions under which a model of a given system can be evaluated toward the verification and validation goals (Zeigler et al. 2000). Well-formulated experimental frames can show the degree to which a model's dynamics satisfy the expected or actual system's behavior. It can also be strictly used for "testing" the behavior of a model's observed output matches exactly the expected or actual system outputs. The EF can have three types of components: generator, acceptor, and transducer. A *generator* stimulates a model of interest with a set of input trajectories. These inputs can be used to subject the model through a finite number of intermediary state changes until the acceptor is triggered. An *acceptor* discriminates the intermediary states from steady states. It is useful for extracting a select collection of possible experiment trajectories. Finally, a *transducer* receives and evaluates the inputs and outputs of the model under some experiments. The components of the EF have

set-theoretic specifications akin to the DEVS formalism. The main experimental frame role is to explore possible dynamics of a model (or more generally software systems).

Unlike validation (showing the right model is built) and verification (showing the model is built right), testing checks for inaccuracies or errors in the model (Balci 1994). Thus, a *test frame* is a special type of experimental frame where the objective is to show for a given input to a model, the resulting output matches the expected output. The primary result for this type of experiment is either pass or fail. A test frame can also use assertions for debugging code at the exact moment any test fails. A test frame's objective is to stimulate and observe a model under test according to a prescribed test plan. Software unit testing frameworks support defining tests and assertions (Garousi and Felderer 2016). In unit testing, the test frame targets a single atomic or coupled model. For integration testing, a test frame is applied to a multi-component model as in a hierarchical coupled DEVS model. For complex models, it is beneficial to write many independent tests using the same test frame. Section 2 will address scalability and complexity by making the test frame extensible for different levels of testing and using Java annotations for running a different test script in each iteration.

1.3 Automated Test Scripts

Consider the elements of a software test plan and the role each model will have in an automated test script (Garousi and Felderer 2016) and simulation.

- *Test documentation.* For scripted test plans, test documentation may be the same as code documentation. It should clearly communicate to the tester input specifications, expected behaviors and results, environment for executing tests, and any inter test case dependency.
- *Setup.* The goal of this phase is to create the test, construct inputs, construct models, their coordinators and simulators, and guide the system to the correct initial state for testing.
- *Exercise.* This phase injects inputs, propagates time, and collect outputs from the system.
- *Verify Results.* Compare results with expected results. For automated script test plans, this is usually done with logical assertions. If an exception is raised, a debugger will capture it and halt the simulation at the exact moment a test fails. This offers developers the diagnostics they might need to solve any failures.
- *Teardown.* This phase resets the system and frees up resources.

Experiment frames that separate the generator, acceptor, and transducer behaviors also separate the elements of a test plan. This abstract, high-level design is limiting in terms of test development, software complexity, and diagnosing failed tests. In a scripting design, all elements—documentation, setup, exercise, verify results, and teardown—reside in one location. Scripting combines the generator, acceptor, and transducer into a single model enabled through a script. Similar to the `@Test` annotation in JUnit, scripts are annotated with `@TestScript` to run independent tests using the same test frame.

Many M&S test frameworks focus on using a platform independent model (PIM) that structure tests into a test oracle. This is beneficial in that non-programmers can write tests in formats like XML and JSON. However, not all tests have well-defined test oracles. Scripting offers flexibility that these rigid formats cannot easily emulate.

1.4 Multithreading and DEVS

DEVS formalism by design makes no assumption about the number of threads or processes used for processing models concurrently in simulation (Zeigler et al. 2000). The code for a DEVS model only executes as a response to an internal state transition or when an input is received. Another design for modeling may generate a thread or process for each model and allow each model to control when it executes code or when it sleeps. There are benefits and consequences to both designs. The following are disadvantages for each model to control its execution via its own thread:

- *Resource limitations.* Operating systems and available memory will limit the number of processes and threads than can be created. A simple DEVS model that takes a few bytes of memory can expect to allocate a much larger stack space for execution.
- *Thread switching.* The number of threads is not usually limited by the number of cores for the processor. Therefore threads must share processor time with other threads with performance cost.
- *Synchronization.* With models running in parallel, synchronization is required to address issues with parallel execution such as a race condition to a shared resource like time and message coordination.

We can immediately see that a thread-per-model design will not be scalable. In this research, we explore the usefulness of scripting the behavior of a model in a thread—more specifically, a test frame because it will typically have one instance in any scenario and thus avoids issues with scalability. Now consider the advantages of scripting its behavior:

- *Reduced lines-of-code and complexity.* Models that move sequentially through code can do so without leaving the routine. Without using a separate thread and stack frame, a DEVS model would require the same routine to save intermediate variables and use logic statements to rebuild each phase of the overall sequence.
- *Exceptions versus Logging.* Logging a pass/fail to an output file is one way to capture results, but raising exceptions can be caught by a test framework or debugger. This practice may be more useful when programmers need to diagnose failed tests.
- *Persistent Scope.* Generator and transducer behaviors can be contained in a single persistent scope. If a test case fails or program execution hits a breakpoint, variables within the scope will still be initialized and used for diagnostics within a debugging environment. If the generator was separate, the injected problematic message has likely lost its reference from generator's scope.

Parallel processing is not an advantage for this double threaded design because nest lock synchronization only permits one thread to have the lock (Waddell and Leathrum 2019). The concepts of threading and synchronization do not exist in DEVS formalisms, but a goal for this research is to prove automated test scripts are a useful and viable substitute testing tool for DEVS models. However, if we challenge the conceptual design of a test frame, we must show the DEVS semantics are not broken for DEVS-Scripted models. A proof for this is offered in Section 2. Recall that state variables can only change in state transition functions. Therefore, program control must wait inside the thread running the script while the model is outside its state transition function. This is accomplished with synchronization.

1.5 Nest-Lock Synchronization

Program execution is considered blocked when a caller must wait on the results of the called function. In a single thread, either an atomic simulator, a coupled coordinator, or one of the models has control of program execution. Synchronization is not required because the simulator hands off program control when it calls a model's state transition method and therefore cannot advance simulation time, coordinate message I/O, or carryout any other roles until the model hands program control back to the last caller.

A single-thread DEVS enforces an encapsulated design. Given that only one state transition may be executed at any time, there is little need for a locking mechanism to protect the model's private data. In multi-threaded DEVS simulators, multiple DEVS models are permitted to execute simultaneously. Synchronization within multi-threaded DEVS simulators is key to promoting repeatability. The proposed script-based testing used in this research will run a script in a separate thread and use a nest lock to synchronize events. A typical nest lock is used to synchronize two threads: outer thread and inner thread—also named source thread and sink thread, respectively. Since only one of these threads can be active at any time, it will behave as if the code executed in the inner thread was executed in sequence within the outer code. We can observe the synchronization of a typical nest lock within the left graphic of Figure 1.

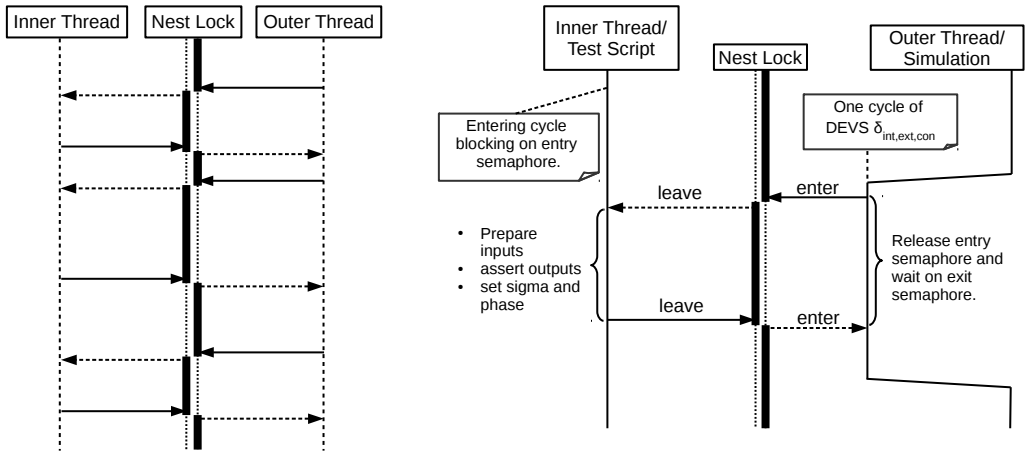


Figure 1: Sequence of coordination in nest lock.

In this research, the DEVS-Script will take the role of the inner thread. The script thread has control until it attempts to read inputs, generate outputs, or enters waiting. These interactions release control back to the outer thread. The outer thread is the DEVS model and simulation thread. The script depicts how the model should behave over time and is enabled by atomic DEVS state transition and output functions. A nest lock uses two semaphore locks, one for entry and one for exiting nested code. Note that the test thread does not execute until the simulation thread reaches $\delta_{int,ext,conf}$. The response from $\delta_{int,ext,conf}$ is passed to the test script and yields control of the nest lock. When the test script performs its next command, it yields control back to $\delta_{int,ext,conf}$. A closeup of this behavior is observed on the right side of Figure 1.

1.6 Black-box Testing of DEVS models

Black-box testing is a type of test driven development. The code may not be available or not written yet. Black-box testing can be designed and developed based on requirements. The greater the inherent complexity of requirements is, the more number of tests that should be developed to test these models.

When modeling Figure 2 in DEVS, the upper self transition arc on **busy** represents δ_{ext}, which assumes *in*?. The lower self transition arc on **busy** represents δ_{conf}. The arc from **busy** to **idle** represents δ_{int}. The only state transitions from the **idle** phase is δ_{ext}, which is shown by the arc from **idle** to **busy**.

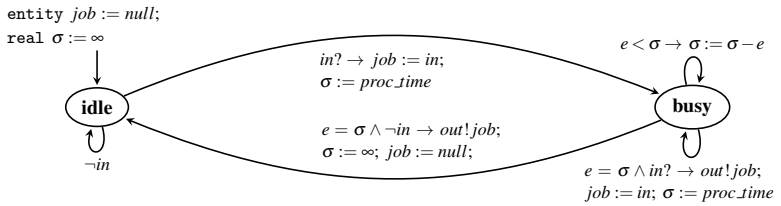
The existence of state variables is implied by the required behaviors, but the number of state variables, their data structures, and names, are unknown. Even the type of model, atomic or coupled, cannot be assumed. With this encapsulation, the only way to interact and test the proper behaviors of this model is via its ports. In software testing, a test oracle defines the inputs and expected outputs. Since time can be another metric to test, tests for the processor model shown in Figure 2 should also verify timing. The following tests visit all states and arcs from the extended state machine described in Figure 2:

1. Allow the system to initialize and assert no messages leave the processor.
2. Send a job to the processor and assert the job leaves the processor at $e = proc\_time$ and matches the job sent in.
3. Send a job to the processor. While the processor is busy, send another job to it. Assert the first job is released on schedule and the second job is not released.
4. Send a job to the processor. Wait and send another job at the time the first job would be released. Assert the first job is available and assert the second job leaves the processor at $e = proc\_time$.

Ideally, we would like to test as few states and arcs as possible in each test, but also offer coverage across all states and arcs when combined. However, the model we are testing is a black box and we can

Atomic Processor Modeling Requirements

1. The model **shall** implement the following extended state machine design:



2. The model **shall** have an input port—"in"—and output port—"out".
3. The model **shall** have a constructor with the signature `proc(name:String, proc_time:double)`.
4. Messages received and sent from this model **shall** be of a generic entity type `GenCol.entity`.

The Process model accepts a job and delays its release. Jobs received while the model is busy are balked. Internal state transitions occur when $e = \sigma$. External state transitions occur when *in*?. Confluent state transitions occur when both conditions are met. Arcs are annotated with $Guard \rightarrow Update$.

Figure 2: Requirements for a simple processor model. The concept of round for this extended state machine is triggered by each call to $\delta_{fun}(S,e,X^b) : \delta_{fun} \in \{\delta_{int}, \delta_{ext}, \delta_{conf}\}$.

only achieve confirmation of results when the system under testing produces outputs. In other words, we cannot prove a processor was busy or working on the correct job until it finishes and passes the job back to the test frame. Many practices in testing software translate over to testing models including boundary value analysis and equivalence partitioning (Reid 1997). Visiting states and arcs achieves the same goals as statement and branch coverage. A simple coverage standard for testing coupled models is to ensure every internal and external coupling has transmitted at least one message. Black-box testing also has applications in education, service oriented architectures, and crowd sourcing. Suppose Figure 2 was used as a homework assignment. We receive numerous designs and implementations. Automated test scripts can be written to interact with each model to ensure the proper I/O behaviors are observed—assigning a pass/fail for each student. This concept is demonstrated in the reference for Section 3.

2 DEVS-SCRIPTING

A test frame atomic model contains generator, acceptor, and transducer behaviors. The model(s) under testing are coupled with the test frame inside a test fixture that serves as the root model in the simulation. However, the test frame—an atomic model—contains all the elements of a test. Therefore, it is responsible for generating the fixture model, inserting itself and all relevant internal models into the fixture, and establishing couplings. Once the fixture is generated, a coordinator is generated for it to run the simulation and ultimately the underlying test thread. During initialization of the test frame, the thread shown in Figure 4 is created.

```
test_thread = new Thread(){
   public void run(){
      nest_lock.innerSectionAcquire();
      script.run(); // may make multiple calls to nest_lock.leave()

      setPhase("done");
      setSigma(Double.POSITIVE_INFINITY);
      nest_lock.innerSectionRelease();
   }
}
```

The initializing phase is held for $\sigma = 0$. The first action by this thread is to synchronize with the simulation and block until the model enters the δ_{int} state change at $e = 0$ where it enters the inner section. The test script is then called. Once the test script exits, the model passivates ($\sigma = \infty$). The base `TestFrame` model implements its δ_{int}, δ_{ext}, and δ_{conf} methods and child classes extend by implementing test scripts. Since test scripts are run in a different thread from the main simulation, the script and the `TestFrame` model must coordinate using a nest lock to safely exchange the method's parameters and state variables.

It is important to note that program control within the test script only occurs while inside either δ_{int}, δ_{ext}, or δ_{conf} as shown in Figure 1. Even though the nature of the test scripting design does not appear to follow DEVS formalisms, it proves that an implementation can be made to conform to a DEVS model—albeit, far more complicated than a single test script. The blocks of code executed when the test thread has control can be written in $\delta_{int,ext,conf}$ and guarded by a large switch statement to identify the stage of testing the frame is currently in. This practice would not be very useful because it increases the complexity of the test case—potentially introducing bugs in testing—and past diagnostics are lost to the debugger when local variables are freed at the end of each call to a state transition method for the test frame. The class diagram depicted in Figure 3 highlights the versatility of the DEVS Scripting for several testing applications. The Methods and attributes for the `TestFrame` are shown in Figure 4.

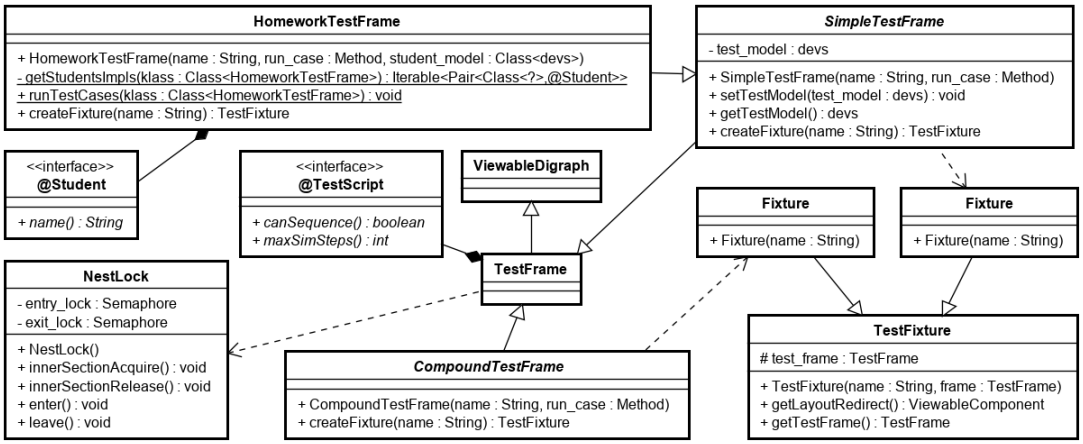


Figure 3: UML class diagram for DEVS-Scripting in DEVS-Suite.

2.1 Scripting Methods

The `TestFrame` methods shown in Figure 4 are intended to mimic instructions similar to a software test procedure (STP). The user usually follows instructions like 'do this action/input', 'wait until a specific event', and 'check the output.' Common methods include `Inject`, `Wait`, and `WaitForOutput`. If the modeling requirements are lenient on timing, testers may want to inspect on a range of time. These methods include `WaitForOutputAround` and `WaitUntilOutputBetween`. On the other extreme, to effectively test superdense trajectories (Sarjoughian and Sundaramoorthi 2015), we need the ability to read the next imminent message. This is done by `ReadOutput` using a sigma greater than zero, but asserting that zero time has elapsed when an output is received.

Some methods are required to effectively test the confluent state transitions of parallel-DEVS models. Such behaviors like wait and inject must be merged into a single cycle. Without the method, `WaitAndInject`, a message received by the end of `Wait` would be cleared during the call to `Inject`.

The implementations of each method above controls one or more cycles through the nest lock and the phases the test script will enter (McLaughlin and Sarjoughian 2020). At the core of each scripting method is the following low-level method, `# ReturnToDeltFunc()`. This function performs basic checks on the

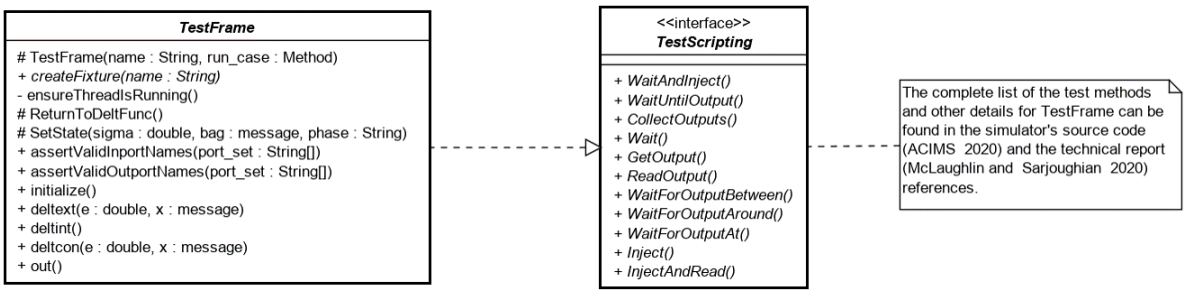


Figure 4: UML `TestFrame` methods and attributes.

thread, nest lock, and output ports of the message bag. It sets the phase, sigma, and output for the test frame before surrendering control back to the simulation thread.

2.2 Writing Tests

Similar to the `@Test` annotation used in JUnit, each test that can be run using the same test fixture—i.e. same components and couplings—can be written as separate methods and identified with annotations. The `@TestScript` annotation has been created under the `TestFrame` class to identify individual tests in child classes. The four test case scripts generated for the processor described in Section 1.6 are:

```
@TestScript
public void TestPassivated(){
   entity e = WaitUntilOutput(100.0, "out", true);
   assertNull(e);
}

@TestScript
public void TestNormal(){
   entity job_in = new entity("job");
   Inject("in", job_in);

   entity job_out = WaitForOutputAt(10.0, "out", false, false, false);
   assertEquals(job_in, job_out);
}

@TestScript
public void TestBalked(){
   entity job_in1 = new entity("job1");
   entity job_in2 = new entity("job2");

   Inject("in", job_in1);
   WaitAndInject(5.0, "in", job_in2, false);

   entity job_out = WaitForOutputAt(5.0, "out", false, false, false);

   assertEquals(job_in1, job_out);

   job_out = WaitUntilOutput(100.0, "out", true);
   assertNull(job_out);
}

@TestScript
public void TestConfluence(){
   entity job_in1 = new entity("job1");
   entity job_in2 = new entity("job2");

   Inject("in", job_in1);
   WaitAndInject(10.0, "in", job_in2, false);
```

```
    entity job_out = GetOutput("out", false, false);
    assertEquals(job_in1, job_out);

    job_out = WaitForOutputAt(10.0, "out", false, false, false);
    assertEquals(job_in2, job_out);
}
```

Similar to writing tests in JUnit, these test cases can be written in a single test script, however if a failure occurs in the first case, any following test cases will not be executed. It is generally better to keep test cases separate for this reason, but other limitation should be considered. Each test case created in JUnit creates a simulation and constructs all models. If the model under testing loads large databases or the simulation has heavy overhead, the setup and teardown consume significant resources including processing time. If there are a large number of test cases, a comprehensive regression testing may no longer be feasible. For managing this issue, an attribute called `canSequence` under `@TestScript` should be set to true so that all sequencible tests—ones that return the model to a common initial state—are run in one instance during regression testing. Only failures within the sequence would warrant independently testing each script.

2.3 Running DEVS-Scripts in JUnit Jupiter

The DEVS-Suite simulator allows users to run simulations without the overhead for run-time visualization and animation by creating a facade-less coupled model coordinator around the test fixture. The simulation runs indefinitely allowing the test frame to inject messages into and receive messages from the system under testing. If the test frame throws an `AssertionError`, JUnit will capture it and fail the test.

Typically Java software tests are written using the `@Test` annotation in a separate class. However, each test is already recorded as a `@TestScript` within a derived class of `TestFrame`. To simplify testing, we will take advantage of the `@TestFactory` annotation in JUnit Jupiter. Any new tests can be created without modifying the following code:

```
class procTest{
    @TestFactory
    @DisplayName("Test Case Generator")
    Stream<DynamicTest> getTestCases(){
        return TestFrame.getTestCases(procFrame.class);
    }
}
```

The method, `+ getTestCases(Class<? extends TestFrame>):Stream<DynamicTest>`, shown in Figure 4 runs through the specified class for public methods using the `@TestScript` annotation and generates dynamic tests for JUnit to execute. Each dynamic test first constructs the test frame, builds a test fixture, assigns the fixture to a coordinator, and runs the simulation. Exceptions caught by JUnit are handled by showing a back-trace, but breakpoints can be inserted and the conditions that caused the failure be observed. Running test frames inside a JUnit test case are especially useful for running automated regression tests.

3 DEMONSTRATION AND USE CASES

DEVS-Scripting is introduced to the DEVS-Suite simulator version 6.0 (ACIMS 2020). Details covering exemplars, creating black-box test scripts, performing white-box debugging, and the DEVS-Scripting API can be found online (McLaughlin and Sarjoughian 2020). The models in Figure 5 demonstrate the extensiblility of test frame shown in Figure 3.

The components presented in the first row of Figure 5 represent a few elements of manufacturing: FIFO buffer, CONstant Work-In-Process (CONWIP) control system, Work Station (WS), and the simple processor model developed from Figure 2. Each of the models are tested by the components in the second row using scripting in DEVS. The third row interfaces JUnit in support of automated regression testing.

The `HomeworkTestFrame` is an extension of `SimpleTestFrame`, but imports multiple implementations annotated by the `@Student` annotation. This specialized test frame tests each implementation against

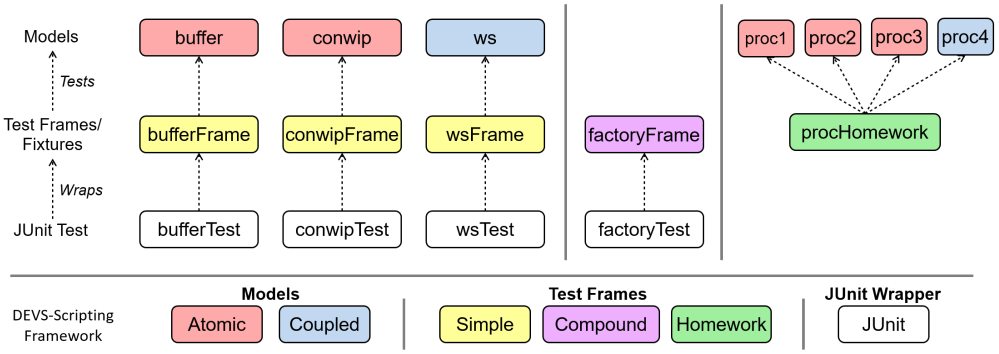


Figure 5: Demonstrative models included in DEVS-Suite.

each test script. These implementations could have been acquired by crowd sourcing, evaluation of similar web-service models, or homework solutions as highlighted in Figure 5. This is one of many ways that DEVS-Scripting demonstrates extensibility.

3.1 Loading the Test Frame into DEVS-Suite

The model loader in DEVS-Suite identifies `TestFrame` objects (see the `Test` drop down menu in "model configuration" UI snippet in Figure 6. DEVS-Suite will construct the frame, but use its fixture as the root model for the simulation. This redirection is shown by the **[Test Fixture]** in Figure 6. DEVS-Suite also recognizes the test scripts and allows testers to observe any annotated test script.

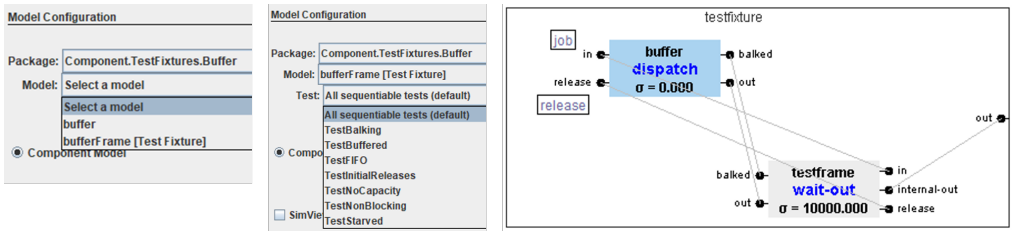


Figure 6: Buffer test frame, test case, and a coupled testframe and simulation component view.

DEVS-Suite allows operators to step through the simulation, watch animated message traffic, interact and inspect with models, and plot superdense input, output, and state trajectories. In terms of the testing, it serves as another diagnostics tool for debugging failed tests.

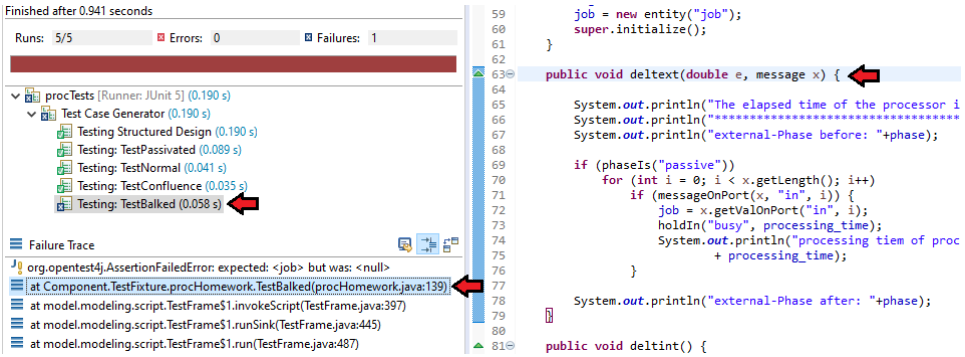


Figure 7: Debugging a failing test using JUnit in Eclipse.

Figure 7 depicts a failure exposed by the `TestBalked` test script written in Section 2.2. Unlike other tests, balking interrupts the processor model while it is actively busy. From the diagnostics on the left, we see that this interrupt caused the first job to miss its deadline. Since the assertion is thrown in the test script thread, all other threads continue to run. We can take advantage of this feature by hovering over models in DEVS-Suite to examine the phase and sigma of the processor model among other debugger diagnostics to find that external state changes do not advance time—effectively resetting the internal state transition to be scheduled on a full delay. A line should be written on the right side of Figure 7 to subtract the elapsed time from the current sigma.

4 RELATED WORK

This research gathers a plethora of software and modeling topics. DEVS-Scripting was enabled using a separate thread for scripting the behavior of a model from initialization until it finishes. Scripting was then used to drive automated black-box testing of models. This research focuses on creating a test frame—a type of experimental frame—around one or more Parallel DEVS models. Scripts written for a test frame support all elements of testing: documentation, setup, exercise, verification of results, and teardown (Garousi and Felderer 2016). Proper synchronization (Zeigler et al. 2000) and Black-box testing techniques (Reid 1997) are employed to ensure the semantics of the DEVS models are preserved.

Considering DEVS models, the concept of a black-box test unit is applied for debugging dynamic structure DEVS models as a collection of static structure DEVS models (Barros 1998). Unit tests are defined as atomic DEVS models and coupled with DEVS models of a system to form coupled DEVS models. These tests are defined according to the DEVS formalism and must use the same simulation protocol as the one for the DEVS models being tested. This is in contrast to the Test Frame approach proposed in this paper. The testing and simulation are strictly separated in terms of distinct specification approaches, each with its own executing protocol. A similar concept for black-box testing I/O behaviors is proposed (Hu et al. 2007), but rather than defining test scripts, the test frame is a coupled DEVS model containing `holdSend`, `processDetect`, `waitReceive`, and `waitNotReceive` atomic DEVS primitives. This approach can be difficult to apply when strongly defined inputs and expected outputs are not the focus of a test. For example, meta-morphic properties or emergent behaviors of complex systems may need complex logic to test them. Another framework for testing offering some unique advantages (Li et al. 2011), but the framework is not implemented as a DEVS model and requires a test oracle.

Another possible implementation of DEVS-Scripting may be employed by a test simulator. However, the capability would be intertwined with simulation objects and classes and thus would not be DEVS derivable. We would not be able to step, visualize, and track the testing in DEVS simulation software—all tools needed to help troubleshoot why tests failed. Ironically, this technique is employed in this research as a means to test the test frame code in DEVS-Suite. In addition to the diagnostic advantages of running test scripts in a separate thread, our test framework can scale up in complexity to support testing complex system-of-systems. Using scripting to leverage the ease of test case development and debugging in M&S has applications outside DEVS. In fact, this research was first implemented in FireSimXXI—a simulator being used in US DoD. As the concepts grew, disciplined and well-known simulation formalisms were needed to gain footing within a broader simulation community. Lastly, a test frame can be derived that loads XML or JSON files to dynamically generate test scripts—offering the flexibility of running test oracles as well as other types of testing.

5 CONCLUSION

The novel black-box test framework proposed in this research is derived from atomic DEVS and facilitates testing through scripting. It provides the advantages of scripting in a separate thread without breaking the correctness of the simulator. This approach is grounded in maintaining modularity for experimental and testing (i.e., DEVS models can be used seamlessly with the experimental frame components or the DEVS-

Scripting). Synchronization was key to proving semantics of DEVS formalisms were not undermined by nesting script code within the state transition functions of DEVS modeling components. When considering platform independence and structured tests, DEVS-Scripting was purposefully made to be extensible as demonstrated in Figures 3 and 5. We demonstrated the process of generating test scripts from a requirements model for a simple processor model. We highlighted a simple prototypical CONWIP control system as well as an example for use in educational settings. The specifics for scripting methods are excluded in favor of accurately representing the approach behind it.

REFERENCES

ACIMS 2020. "DEVS-Suite, Version 6.0". https://acims.asu.edu/software/devs-suite/, accessed 15th September.

Alur, R. 2015. *Principles of Cyber-Physical Systems*. Massachusetts Institute of Technology Press.

Balci, O. 1994. "Principles of Simulation Model Validation, Verification, and Testing". Technical report, Virginia Polytechnic Institute & State University, Blacksburg, Virginia.

Barros, F. J. 1998. "Hierarchical Testing of Dynamic Structure Models: A Practical Approach". *Transactions of the Society for Computer Simulation International* 15(4):181–189.

Garousi, V., and M. Felderer. 2016, May-June. "Developing, Verifying, and Maintaining High-Quality Automated Test Scripts". *Institute of Electrical and Electronics Engineers Software* 33(3):68–75.

Hu, X., B. P. Zeigler, M. H. Hwang, and E. Mak. 2007. "DEVS Systems-Theory Framework for Reusable Testing of I/O Behaviors in Service Oriented Architectures". In *2007 IEEE International Conference on Information Reuse and Integration*, 394–399. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Li, X., H. Vangheluwe, Y. Lei, H. Song, and W. Wang. 2011. "A Testing Framework for DEVS Formalism Implementations". In *Proceedings of the 2011 Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium*, TMS-DEVS '11, 183–188. San Diego, CA, USA: Society for Computer Simulation International.

McLaughlin, M. B., and H. S. Sarjoughian. 2020. "Developing Test Frames for DEVS Models: Black-Box Testing with White-Box Debugging". Technical report, School of Computing, Informatics, and Decision Systems Engineering, Arizona State University, Tempe, Arizona, USA.

Ören, T. I. 1981, April. "Concepts and Criteria to Assess Acceptability of Simulation Studies: A Frame of Reference". *Communications of the Association for Computing Machinery* 24(4):180–189.

Peters, D., and D. L. Parnas. 1994. "Generating a Test Oracle from Program Documentation: Work in Progress". In *Proceedings of the 1994 ACM SIGSOFT International Symposium on Software Testing and Analysis*, edited by T. Ostrand, 58–65. New York, New York, USA: Association for Computing Machinery.

Reid, S. C. 1997. "An Empirical Analysis of Equivalence Partitioning, Boundary Value Analysis and Random Testing". In *Proceedings Fourth International Software Metrics Symposium*. November 5th-7th, Albuquerque, New Mexico, 64–73.

Sarjoughian, H. S., and S. Sundaramoorthi. 2015. "Superdense Time Trajectories for DEVS Simulation Models". In *Proceedings of the Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium*, 249–256. San Diego, CA, USA: Society for Computer Simulation International.

Srikanth, H., and S. Banerjee. 2012. "Improving Test Efficiency Through System Test Prioritization". *The Journal of Systems and Software* 85(5):1176–1187.

Waddell, B., and J. F. Leathrum. 2019. "A Multithreaded Simulation Executive in Support of Discrete Event Simulations". In *2019 Winter Simulation Conference*, edited by N. Mustafee, K.-H. Bae, S. Lazarova-Molnar, M. Rabe, C. Szabo, P. Haas, and Y.-J. Son, 2677–2688. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Zeigler, B. P., H. Praehofer, and T. G. Kim. 2000. *Theory of Modeling and Simulation*. 2nd ed. Orlando, Florida, USA: Academic Press, Inc.

AUTHOR BIOGRAPHIES

MATTHEW B. MCLAUGHLIN graduated with a Masters of Engineering in Modeling and Simulations from Arizona State University in the Spring of 2020. He works as a lead computer scientist for the Fires Battle Lab. His email address is mclaughlin.matthew.2@gmail.com.

HESSAM S. SARJOUGHIAN is an Associate Professor of Computer Science & Computer Engineering at Arizona State University and Co-Director of the Arizona Center for Integrative Modeling and Simulation. His research focuses on model theory, polymorphic model composability, distributed co-design modeling, visual simulation modeling, real-time modeling, and service-oriented simulation. He can be contacted at sarjoughian@asu.edu. His website is http://sarjoughian.faculty.asu.edu/.