

A Model-Based Testing Framework For Distributed Simulations

Tom van den Berg
0009-0000-9121-3148
TNO-MST
The Hague,
The Netherlands
Email: tom.vandenberg@tno.nl

Mathijs Schuts
0009-0002-4166-288X
TNO-ESI
Eindhoven
The Netherlands
Email: mathijs.schuts@tno.nl

Jeroen Voogd
0000-0002-9253-5090
TNO-IDS
The Hague
The Netherlands
Email: jeroen.voogd@tno.nl

Abstract—This paper explores the use of Model-Based Testing (MBT) techniques and in particular the Component Modelling & Analysis (ComMA) framework to automate the verification of distributed simulation components. As defence simulation environments grow in complexity, ensuring reliable interoperability between modular components becomes increasingly challenging. ComMA's MBT support is extended with simulation-specific capabilities, such as logical time management and dynamic entities, enabling automated online testing from formally specified interface models. Use cases demonstrate the feasibility of this approach. The work also aligns MBT activities with established simulation engineering processes. Results highlight benefits, remaining challenges, and directions for future work.

Index Terms—MBT, HLA, ComMA.

I. INTRODUCTION

MODELLING and Simulation (M&S) has become a critical capability for the North Atlantic Treaty Organization (NATO) and its member nations in addressing the increasing complexity of modern defence operations. Simulation environments are widely used to support collective training, capability development, mission rehearsal, and decision support. As operational contexts evolve rapidly, these environments must be flexible, scalable, and cost-effective, enabling rapid reconfiguration to meet new and emerging requirements.

A key enabler for this flexibility is the ability to compose simulation environments from modular, reusable simulation components. Such components are often developed by different partners and organisations and must interoperate seamlessly when combined. Modularity therefore plays a central architectural role in distributed simulation systems. However, as the number and diversity of components grow, ensuring that these components interact correctly and reliably becomes increasingly challenging. Faulty interfaces or incorrect assumptions about behaviour can undermine the credibility and usability of the resulting simulation environment.

To address this challenge, robust interoperability standards, integration processes, and verification mechanisms are essential. In particular, automated verification capabilities that can systematically test whether simulation compo-

nents conform to their specified interfaces and interact correctly within a composition are crucial for establishing trust in synthetic environments. Manual integration and testing approaches are often time-consuming, error-prone, and difficult to scale, making them inadequate for large and dynamic simulation compositions.

This paper investigates how automatic verification of compositions of simulation components can be realised using Model-Based Testing (MBT). MBT offers a systematic approach in which formal or semi-formal models of component interfaces are used to generate and execute tests automatically. The focus of this work is on the use of the open-source Component Modelling and Analysis (ComMA) framework [1], which provides language and tool support for specifying component interfaces and conducting model-based tests. Although ComMA was originally developed for cyber-physical systems, this research explores its adaptation and applicability to distributed simulation systems.

A. Research Questions

The central research questions addressed in this paper are:

- how can a composition of simulation components be automatically verified, and
- what extensions or additions to existing Verification and Validation (V&V) standards for simulation are required to support such automated verification.

To answer these questions, the ComMA framework is adapted to accommodate simulation-specific concerns, such as time management and the dynamic creation of simulation entities during runtime. The feasibility of this adapted approach is demonstrated through a set of representative simulation use cases.

In addition, this work reviews existing research on the application of MBT within the simulation domain, with particular attention to online testing approaches and recent standardisation efforts. Based on this review, implications for established simulation engineering processes are analysed. The findings are mapped onto the Distributed Simulation Engineering and Execution Process (DSEEP) and its V&V overlay, see Section V.B, ensuring that the proposed approach aligns with current engineering practices

and can be understood and applied by both simulation systems engineers and decision-makers.

B. Scope

The scope of this paper is limited to the verification of interfaces between simulation components. While interface verification is a necessary step toward reliable simulation compositions, it does not constitute complete verification of a simulation component, nor does it address validation of whether a simulation is suitable for a specific operational or training purpose. These broader aspects are acknowledged but fall outside the scope of this study.

By demonstrating how automated, model-based verification can support interface testing in distributed simulation compositions, this paper contributes to improving the efficiency, reliability, and scalability of simulation engineering for defence applications, including collective training and Modelling and Simulation as a Service (MSaaS) contexts [2].

C. Document Overview

This paper is structured as follows. Chapter II (Background) provides foundational concepts and summarises the ComMA framework. Chapter III (Methodology) outlines the research questions and the steps for addressing these questions. Chapter IV (Applying ComMA: Testing of Simulation Components) identifies simulation-specific challenges and describes the test environment and key extensions added to ComMA. Chapter V (Model Based Testing in the DSEEP) maps test activities onto the DSEEP steps and the associated V&V overlay. Chapter VI (Discussion) reflects on the extensions introduced and highlights additional aspects requiring attention. And finally, Chapter VII (Conclusion and Recommendations) summarises how automatic verification of simulation component compositions is feasible using the extended ComMA approach.

II. BACKGROUND

This chapter provides foundational concepts: component-based modelling, interoperability versus composability, the High Level Architecture (HLA), and simulation engineering processes. It also summarises the ComMA framework.

A. Component Based Modelling

Component-based modelling is an approach to modelling complex systems by representing the system as a collection of interconnected components, each performing specific functions. In this paradigm, each component is a modular, reusable unit that can be combined with others to build larger, more complex models and simulations, meeting evolving simulation requirements as they arise.

While component reuse certainly occurs in practice, it is often not without significant engineering efforts. Component interoperability and composability remain key concerns to be addressed in Component Based Modelling

(CBM). These terms are close in meaning, but still significantly different:

- *Interoperability* relates to the ability of components to work together within a single simulation environment, developed for a specific purpose.
- *Composability* relates to the ability to use components across simulation environments, with each simulation environment developed for a specific purpose.

From this, it can be said that composability requires interoperability, but not vice versa. If simulation components are interoperable for a specific simulation environment, then they are composable in that specific simulation environment. However, simulation components that are interoperable within a specific simulation environment are not necessarily composable to create other simulation environments.

Research described in [3] and [4] propose a multi-level framework for analysing composability based on component metadata. The identified levels are:

- *Syntactic composability*: at this level, the structure of the components is studied to know if they can fit together, i.e., if the output of one can be read as an input by the other and that the syntactic information of the connected components, such as message type, mode of action, and number of parameters, match each other.
- *Static semantic composability*: this level is concerned with the meaningful interaction of the composed components. This involves having a concise and mutual understanding of the data exchanged by the components participating in the composition. For instance, message types must match, state machines must fit at a static level, and the same concepts are referenced in common ontologies.
- *Dynamic semantic composability*: this level handles matching of the components' behaviour over time. Dynamic-semantic implies that the components are dynamically consistent, i.e., they have suitable stateful behaviour, necessary to reach the desired goals and subsequently satisfy user requirements.
- *Pragmatic composability*: evaluates the difference of actual effect of the messages with the intended effect of messages during communication.

The dynamic semantic and the pragmatic composability levels are the hardest to verify, whereas the syntactic composability and some aspects of the static semantic composability levels are simpler to verify. The syntactic composability level is a prerequisite for the other levels. This multi-level approach is useful to differentiate the various aspects in composability and in the verification of a composition.

Extensive research has been performed in the area of simulation interoperability and simulation composability, resulting in standards like the IEEE High Level Architecture [5] (for simulation interoperability) and the SISO Base Object Model (BOM) [6] (for simulation composability). However, in general, simulation composability is still considered a challenge to be fully resolved.

B. High Level Architecture

The High Level Architecture (HLA) [5] is an IEEE standard for the development of distributed simulation environments. In the terminology of the HLA, an individual simulation component is known as a *federate*. A federate may, for instance, be a simulator, a data recorder, an analysis tool, or a passive 3D viewer. The collection of federates brought together form a synthetic environment known as a *federation*, see Fig. 1.

It is the common interpretation of a shared data model, called the *Federation Object Model* (FOM), that allows federates to interact within a single synthetic environment. A federation execution refers to the process of conducting a distributed simulation. Federates interact via a *Run Time Infrastructure* (RTI). The RTI provides a number of Application Programming Interface (API) service groups that are used by a federate to interact with the underlying communication layer.

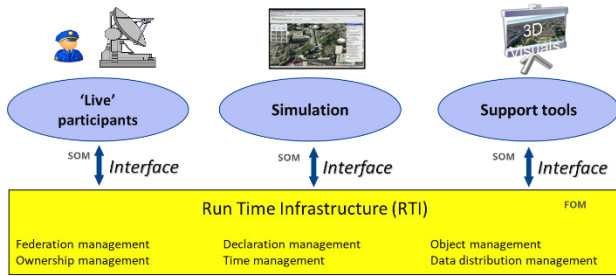


Fig. 1: A graphical view of the HLA: federates operate together as a federation through a common Run Time Infrastructure (RTI).

The HLA is focused on interoperability between several types of simulations and promotes reuse of simulations and their components. The HLA follows two general design principles:

- *modularity*: simulation components (federates) are composed into larger systems (federations) to obtain a specific functional behaviour.
- *separation of concerns*: the functional behaviour of the components (federates) is separated from the supporting communication infrastructure via a well-defined interface.

The standard [5] consists of three parts: the HLA Framework and Rules (IEEE 1516-2025), the HLA Interface Specification (IEEE 1516.1-2025), and the HLA Object Model Template (OMT) Specification (IEEE 1516.2-2025).

The Framework and Rules define responsibilities for federations and federates, including the use of FOMs and Simulation Object Models (SOMs), ownership of object attributes, and time management.

The Interface Specification defines RTI API service groups, covering federation management, data exchange, object lifecycle management, time coordination, and data distribution. These services enable scalable and efficient distributed simulation execution.

All data exchanged within an HLA federation is defined using the Object Model Template (OMT). The OMT specifies the structure of object classes, interaction classes, data types, and synchronization points. Three types of object models are defined: the SOM, which describes an individual federate's capabilities; the FOM, which defines all data exchanged within a federation; and the Management Object Model (MOM), which supports monitoring and control of federation execution. The FOM and SOMs function as technical contracts between federate developers, enabling component replacement and reuse as long as interface agreements are maintained.

C. Engineering Processes for Federated Simulation

The Distributed Simulation Engineering and Execution Process (DSEEP) [7] is a standardized, high-level systems engineering process for planning, developing, integrating, executing, and evaluating distributed simulations. It provides a structured, seven-step workflow—ranging from defining objectives to post-execution analysis—ensuring that simulation federations are built consistently and are interoperable across organizations and technical domains. DSEEP is technology-agnostic but commonly applied to HLA-based simulation environments, and its associated V&V overlay [8] specifies how V&V activities should be embedded throughout the process. Together, these standards help establish repeatable, credible, and well-governed simulation engineering practices.

D. Model Based Testing

Software testing has traditionally relied on *scripted testing*, in which predefined test cases follow fixed sequences of test steps. This approach offers strong reproducibility and supports regression testing. However, scripted testing suffers from the well-known pesticide paradox: as tests are repeatedly executed, they become less effective at revealing new defects [9]. In addition, scripted tests are unsuited for systems exhibiting non-deterministic behaviour because this will lead to complex test scripts with numerous conditional branches that are difficult to maintain.

Exploratory testing provides an alternative, adaptive testing strategy in which test steps are determined dynamically during execution. In this approach, a test application interacts with the System Under Test (SUT) by sending inputs, observing outputs, and selecting subsequent inputs based on those outputs. As a result, exploratory testing naturally accommodates non-determinism and may uncover defects that scripted approaches fail to detect. In practice, an effective testing strategy combines the strengths of both scripted and exploratory testing. Both approaches typically require substantial manual effort to design, implement, and maintain test cases.

Model-Based Testing (MBT) [10], as depicted in Fig. 2 for a schematic representation, addresses this challenge by using formal or semi-formal test models to automate test generation. In MBT, models can be employed either to generate scripted tests in advance (offline testing) or to

guide a long, random walk through the model's state space during execution (online testing). The latter constitutes an automated form of exploratory testing. MBT is a black-box testing technique: the internal implementation of the SUT is not assumed to be known, and testing is driven solely by the externally observable behaviour of the system. This behaviour is captured in a test model, often represented in terms of a state-based protocol. Test steps are then derived automatically from the test model during execution. Compared to large sets of manually written test cases, test models provide a more concise, more maintainable and inspectable representation of intended behaviour.

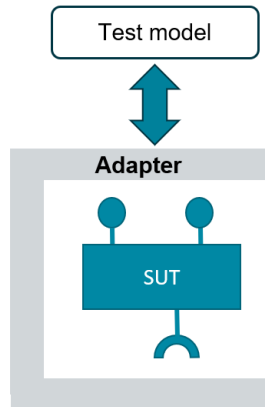


Fig. 2: Model-Based Testing concepts.

In MBT, the SUT may implement behaviour that is not fully captured by the test model, resulting in a partial model. This is acceptable as long as the model test does not prescribe behaviour that the SUT fails to implement, which would lead to failing test cases. To ensure applicability across different implementation technologies, MBT tools rely on *adapters* that translate events between the abstract test model and the concrete interfaces of the SUT. This separation allows the test model to remain independent of technological details.

With online MBT each test execution is expected to be unique, therefore test generation algorithms must incorporate random choices. Integration with automated test environments requires command-line execution to support use within Continuous Integration and Continuous Delivery (CI/CD) pipelines. Furthermore, tools must provide explicit stop conditions, such as time limits, coverage thresholds, or combinations thereof, to prevent unbounded test runs. Test verdicts can be communicated via exit codes, enabling automated interpretation of results in CI/CD scripts.

Effective debugging support is essential in online MBT. Tools should allow test executions to be stored and replayed to reproduce detected issues. To reduce reproduction time, cycles in recorded test traces may be removed. Additionally, step-by-step execution is required to support detailed investigation using conventional debugging tools. Online MBT tools should also support disabling transitions, both to focus testing on specific behavioural subsets

and to temporarily work around known defects that would otherwise block further exploration.

To avoid duplication when testing related systems, online MBT tools should support compositional modelling. This allows reusable model behaviour to be combined for specific SUTs. Finally, because not all SUT behaviour is controllable or predictable, online MBT tools must make dynamic decisions at runtime, selecting model paths that correspond to the actual output observed. After test execution, coverage data—including covered and uncovered states, transitions, and events—should be provided for analysis. A wide range of MBT tools exists, differing in modelling paradigms, and test generation strategies.

E. ComMA

Component Modelling & Analysis (ComMA) is a compositional modelling and testing framework used to formally modelling data types, signatures, interface and component behaviour. Data types include primitives and user-defined structures. Signatures define the interaction events on an interface through *commands* (test application-SUT, blocking), *signals* (test application-SUT, non-blocking), and *notifications* (SUT-test application, non-blocking). Interface behaviour is described as a finite state machine from the server perspective, where transitions are triggered by client events and server reactions are defined in transition actions. Components instantiate interfaces and may specify Functional Constraints (FCs) to relate events across interfaces. Every FC is described in terms of a finite state machine.

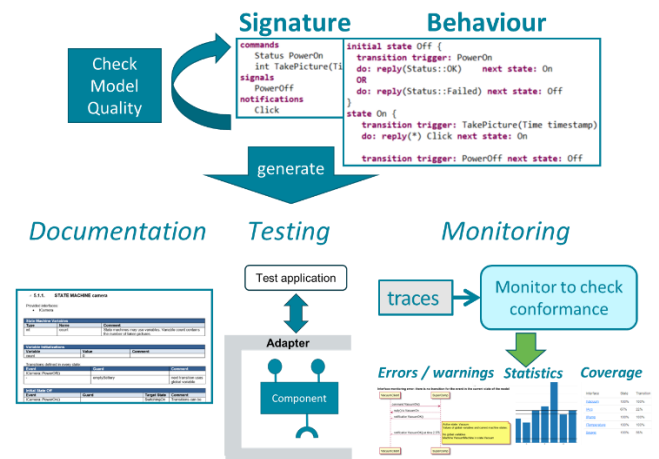


Fig. 3: ComMA overview.

As shown in Fig. 3, ComMA includes generators for documentation, and runtime monitoring. The user can choose the capability it wants to use in a project model. It also supports online MBT [11]. A test application in Python is generated from interface models, and a component model. The combination of these two models, we call the test model. The test application selects enabled actions randomly, and communicates with the SUT via an adapter. A Graphical User Interface (GUI) version visualizes

events, states and transition coverage, and provides debugging controls, while a command-line version supports CI/CD pipeline integrations with stop conditions like coverage and timeout.

The approach is compositional: test applications can be generated for any combination of interface and component models.

III. METHODOLOGY

In this paper, the research objectives are investigated by studying the use and feasibility of ComMA's MBT support [1] for:

- the specification of the interface of simulation components, and
- the verification of a composition of simulation components.

To answer the first question of the research objectives, the ComMA language and framework is adapted in a way it can work not only with cyber-physical systems (where ComMA was originally intended for), but also with simulation systems. This includes aspects such as time management and how new simulation entities (unlike in cyber-physical systems) can be generated at runtime. The applicability of this adapted version of ComMA is demonstrated on a series of simple simulation use cases that requires considering the aspects previously mentioned (e.g., time and entities management).

To answer the second question, earlier work [12] on the use of MBT in simulation is reviewed, with a focus on online MBT. This review takes into account recent work in standardising MBT as test approach. The aim of this review is to identify additions to both IEEE 1730 (DSEEP) and IEEE 1730.2 (DSEEP V&V Overlay).

IV. APPLYING COMMA: TESTING OF SIMULATION COMPONENTS

In this chapter simulation-specific challenges are identified, such as time, dynamic entities, and test model composition. It describes the test environment and presents the key extensions added to the ComMA framework.

A. Aspects to Explore

Online MBT of simulation components requires careful consideration of characteristics that are fundamentally different from those of traditional real-world software systems. In Modelling and Simulation (M&S), the system under test is itself a model of a real-world cyber-physical system rather than the implemented system. This distinction introduces unique challenges for testing, particularly with respect to representation, time, and system dynamics. This section outlines the most relevant aspects that must be addressed when applying MBT in simulation-based environments.

A primary distinguishing factor of simulation is model *validity*, which concerns whether a model is an adequate

representation of reality for its intended purpose. As defined in IEEE 1730.2, validity refers to the extent to which a model or simulation is complete and correct enough for its intended use. Importantly, validity is contextual: a model may be valid for training purposes but unsuitable for predicting rare failure modes. Validity assessment does not address whether the model has been implemented correctly – that is the domain of *verification* – but whether the underlying assumptions, abstractions, and behaviours are credible. Evaluating validity often requires expert judgement and comparison of simulation outcomes with real-world performance data.

Closely related is the concept of *fidelity*, which describes the level of detail and accuracy with which a simulation represents the real-world system. IEEE 1730.2 defines fidelity in terms of representational capabilities such as resolution, precision, error, and sensitivity. The required fidelity depends strongly on the intended application: for example, a flight simulator may require high visual and behavioural fidelity, whereas a higher-level operational simulation may only require coarse physical representations.

On the other hand, a real-world system provides *true performance data* under actual conditions. Since it is a real-world system there are no modelling assumptions and all interactions with and within the system are real.

Another core concept is *ground truth*, defined as the actual, error-free state of the system being modelled. In simulation, ground truth represents perfect information and serves as the reference against which simulated outputs are evaluated. For example, a sensor model may derive perceived positions from ground truth positions expressed in a standard coordinate system. Ground truth is essential for verification and validation, as it provides an objective benchmark for assessing accuracy and fidelity. In contrast, real-world systems typically provide only one truth, influenced by sensor limitations and human interpretation.

The concept of *time* is a particularly distinctive aspect of simulation. Unlike real-world systems, which operate exclusively in wall-clock time, simulations operate in simulation time, which may progress faster, slower, or independently of real time. Multiple notions of time coexist, including simulation time, wall-clock time, and logical time as defined in IEEE 1516 for distributed simulations. Logical time is used to ensure causal ordering of events and may be mapped to simulation time. Simulations may progress simulation time *time-stepped* or *event-driven*, affecting both execution semantics and test design. Time representation itself influences fidelity and correctness; limitations in numerical precision can introduce rounding errors or pathological behaviours such as *zeno effects*, where time advances without ever reaching a specific value. Mechanisms such as minimum time advancement (epsilon, and lookahead) are therefore critical in distributed simulation frameworks.

Simulation environments further differ from real-world systems through the presence of *dynamic entities*. These entities can be created, modified, interact, and be destroyed during execution, often in large numbers. Their interactions may give rise to complex emergent behaviours and typically dominate the computational load of a simulation. This dynamic and scalable nature has significant implications for testing strategies.

Based on these characteristics, several priority aspects for MBT in simulation have been identified. First, simulation time must be explicitly addressed. MBT tools must handle real-time and non-real-time execution, synchronize simulation time with wall-clock time when required, support both time-stepped and event-driven models, and verify causal ordering of events in distributed settings. Second, dynamic entities require MBT support for entity lifecycle management, interaction testing, and scalability to large and variable populations. Third, a compositional approach to test modelling is essential, as modern simulations are built from modular components with multiple interfaces. MBT tools must support flexible composition of test models and verify correct interoperability across component interfaces. Finally, reporting is crucial for understanding test coverage and diagnosing issues. Reporting mechanisms must account for dynamic entities, provide detailed coverage metrics, ensure traceability, and remain usable as system complexity increases.

Together, these aspects define the unique challenges and requirements for effective MBT of simulation components.

B. Test Environment

Fig. 4 illustrates that the test environment comprises three components: a Test Application, SUT Adapter and a Simulator.



Fig. 4: Main components in the test environment.

The Test Application, which is generated from a ComMA model, and SUT Adapter communicate via the ComMA protocol, i.e., formatted JSON messages. The SUT Adapter and Simulator exchange information through the HLA RTI.

C. Results per Aspect

This section presents four major extensions made to ComMA and the generated test application to support advanced distributed simulation and testing scenarios. These extensions address limitations related to time management, dynamic simulation entities, multi-model composition, and reporting in environments where the number of active entities varies over time.

I. Logical Time

Logical time is implemented using two administrative messages:

- TAG (Time Advance Grant): Sent by the RTI to the Test Application, informing it that it may advance to logical time T .
- TAR (Time Advance Request): Sent by the Test Application to the RTI, indicating that it is ready to advance its logical time to T .

A four-phase interaction pattern is used, illustrated in Fig. 5.

- The RTI sends a number of messages to the Test Application (via the SUT Adapter) just like it would do in wall-clock time.
- A TAG message is sent from the RTI to the Test Application (via the SUT Adapter) that tells the Test Application how far it may advance in time.
- Next, the Test Application ticks the state machines and sends thereby messages to the RTI (via the SUT Adapter). How far to advance here depends on how much time the transitions take versus the time in the TAG message.
- Finally, a TAR message is sent from the Test Application to the SUT Adapter with the amount of time (dt) to progress. The SUT Adapter adds the value T to the amount of time to progress, and issues a Time Advance Request Available (TARA) to indicate to the RTI that the SUT Adapter is ready to advance to $T+dt$. The TARA (instead of the TAR) enables the RTI to deliver future messages to the SUT Adapter at a time $\geq T+dt$, rather than at a time $> T+dt$. This is a subtle yet important time advancement, because the SUT may very well want to send messages at $T+dt$.

This cycle continues for the duration of the test.

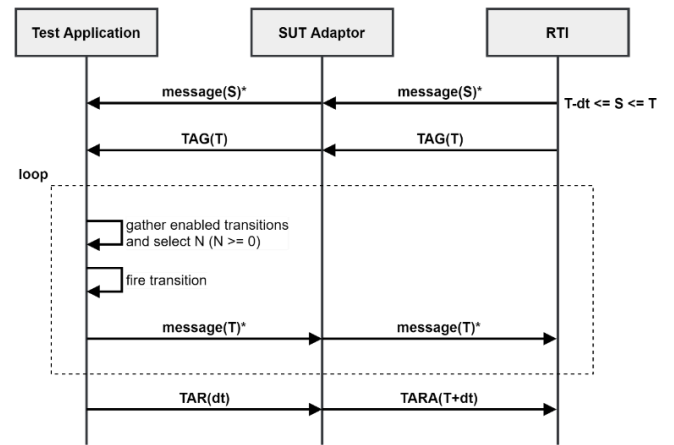


Fig. 5: Sequence diagram of the time advance implementation.

II. Dynamic Entities

Previously, ComMA supported only a fixed number of entities (state machine instances). This created issues: over-allocation wastes resources, while under-allocation risks exhausting available state machine instances. To

overcome this, ComMA now supports run-time creation and deletion of entities. These additions required create/delete functions, and an update of the ComMA protocol between Test Application and SUT Adapter, illustrated in Fig. 6 and Fig. 7.

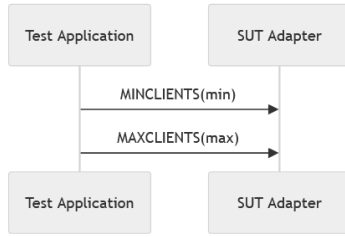


Fig. 6: Configuring the initial number of clients in the SUT Adapter.

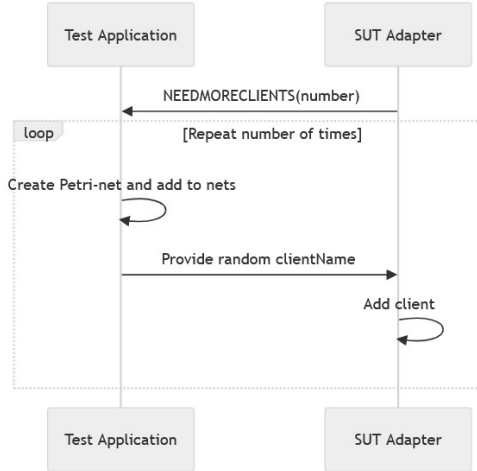


Fig. 7: Request for more clients by the SUT Adapter.

III. Composition of Multiple Test Models

A ComMA test model component may provide several interfaces, but the SUT Adapter does not include interface identifiers in its messages. To support multi-interface and multi-model compositions, a router was added inside the Test Application, illustrated in Fig. 8.

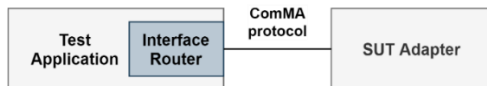


Fig. 8: Connection between Test Application and SUT Adapter.

This router takes the entity name from the ComMA test model and maps it to an interface name. From each incoming message from the SUT Adapter, the interface property is changed into the right interface; hence, a message is augmented with the interface name.

Testing with four similar interfaces confirmed correct routing: incoming messages originally labelled only as "adapter" were rewritten to the correct component interface, while outgoing messages required no modification. The Interface Router thus enabled the composition of test models in ComMA.

IV. Enhanced Reporting for Dynamic Entities

Because the total number of states and transitions changes when entities are created or removed, the original ComMA coverage metrics no longer applied. A new reporting mechanism was introduced. Reporting enhancements:

- Two-dimensional coverage graphs tracks:
 - Seen vs. total states
 - Seen vs. total transitions
- Comma-Separated Value (CSV)-based logging to file allows post-processing in tools like Microsoft Excel.
- Management-only states (e.g., initialization or deletion states) are excluded from coverage to avoid skewing results.
- Coverage resets when entities are deleted and re-created.

Coverage thresholds can also serve as stop conditions for automated testing.

V. MODEL BASED TESTING IN THE DSEEP

This chapter maps MBT activities onto the DSEEP steps and the associated V&V overlay, demonstrating compatibility with established engineering processes. It explains how MBT activities enrich engineering steps with MBT strategy design, test model creation, MBT adaptation, automated execution, and reporting.

A. DSEEP Overview

As distributed simulations increase in scale and complexity, they increasingly resemble systems-of-systems, requiring a structured and disciplined engineering approach. Although generic systems engineering processes can be applied to distributed simulation environments, they lack simulation-specific activities that are essential for credible development and use. These activities include the development of simulation conceptual models and scenarios, the specification of runtime data exchange models with associated operating agreements between member applications, and simulation-specific V&V. The only recognised industry standard that explicitly addresses these needs is the IEEE 1730 Distributed Simulation Engineering and Execution Process (DSEEP) [7]. DSEEP is independent of any particular simulation architecture (such as HLA) and provides a consistent seven-step process covering objectives definition, conceptual analysis, design and development, integration and testing, execution, and post-execution analysis.

DSEEP	Step 1: Define Simulation Environment Objectives	Step 2: Perform Conceptual Analysis	Step 3: Design Simulation Environment	Step 4: Develop Simulation Environment	Step 5: Integrate and Test Simulation Environment	Step 6: Execute Simulation Environment and Prepare Outputs	Step 7: Analyze Data and Evaluate Results
MBT Activities  products			Design MBT Strategy Create Test Model  MBT Strategy Test Model	Implement Adaptation Module  Adaptation Module	Generate and Execute Tests Analyze Test Results  Test Script Test Execution Log Requirements Traceability and Coverage Report Test Issues		
V&V Overlay	Phase 1: Define Simulation Environment Objectives	Phase 2: V&V distributed simulation conceptual model	Phase 3: Verify distributed simulation design	Phase 4: Verify distributed simulation development products	Phase 5: Validate distributed simulation	Phase 6: Accept distributed simulation and verify output processing	Phase 7: Consolidate distributed simulation V&V products

Fig. 9: The DSEEP process steps (top), its V&V Overlay phases (bottom) and the MBT activities and products (middle). Steps 3, 4 and 5 are the most relevant for the MBT overlay.

Fig. 9 presents the seven high-level steps of the DSEEP, omitting the detailed activities, tasks, and information flows between steps. Each DSEEP step is further decomposed into activities, tasks, and work products, and the guidance is applicable to both distributed and standalone simulations. For the purposes of this work, steps 3, 4, and 5 are most relevant and are summarised below; the reader is referred to the standard for a complete description of the remaining steps.

Step 3 – Design simulation environment.

This step produces the design of the simulation environment by identifying the member applications that will participate, determining whether existing applications can be reused or must be newly developed, and allocating required functionality to the selected member applications.

Step 4 – Develop simulation environment.

In this step, the information to be exchanged at runtime is defined, interface agreements between member applications are established, applications are modified where necessary, and the simulation environment is prepared for integration and testing.

Step 5 – Integrate and test simulation environment.

Integration activities are performed and formal testing is conducted to verify that interoperability and interface requirements are met.

B. DSEEP MBT Overlay

Earlier work [12] mapped MBT activities onto the DSEEP for federated simulations. Experience gained through the application of the ComMA framework confirms that this mapping largely remains valid. The DSEEP steps can be overlaid with MBT activities derived from IEEE 29119-8 (Model-Based Testing) [13], which itself extends IEEE 29119-2 (Test Processes) [14], with a particular emphasis on online testing.

MBT activities are primarily applied in DSEEP steps 3, 4, and 5. The overlay highlights where MBT introduces new activities or contributes to existing ones, and which MBT-specific work products are generated. These products include test models, adaptation modules, execution

logs, traceability information, and coverage reports. The following sections describe the MBT activities associated with each relevant DSEEP step.

C. DSEEP Step 3 – MBT Strategy and Test Modelling

In step 3, MBT introduces two additional activities: the design of the MBT strategy and the creation of the test model. The MBT strategy defines the overall approach to testing the SUT and addresses key topics such as test objectives, test scope, required coverage, metrics to be collected, selected MBT tools, and the modelling language used to describe the test model. The strategy forms an integral part of the test plan produced in this step.

Following IEEE 29119-8 terminology, an MBT tool typically consists of a modelling module for test model development, a generation module for test case generation, and an adaptation module for automated test execution and communication with the SUT. Depending on the tool, these modules may be implemented as separate applications or integrated into a single environment.

The test model itself is a formal and abstract representation of the expected externally observable behaviour of the SUT. It is derived from the simulation scenario, conceptual model, and simulation environment requirements established in earlier DSEEP steps and is expressed using the modelling language selected in the MBT strategy.

D. DSEEP Step 4 – Adaptation Module

The primary MBT activity in step 4 is the implementation of the adaptation module. This module translates abstract test actions generated from the test model into concrete messages understood by the SUT and maps responses back into abstract events. It also manages connections, message translation, and technical or syntactical issues. By encapsulating SUT-specific details within the adaptation module, the test model remains independent of implementation details. In the context of ComMA, this functionality is realised by the SUT Adapter.

E. DSEEP Step 5 – Test Execution and Analysis

In step 5, MBT supports the automated generation and execution of test cases, as well as the analysis of test results. For online MBT, test case generation and execution are tightly coupled in a closed loop: the generation module derives test actions from the test model, these actions are executed by the SUT, and observed behaviour is fed back to guide subsequent test steps.

The outcomes of this activity include executable test scripts, detailed test execution logs capturing all actions and events during a test run, and test reports summarising results and coverage. These artefacts are used in the analysis of test results, which may identify issues in the SUT, the test model, the MBT toolchain, or the simulation environment requirements.

F. DSEEP V&V MBT Overlay and Evaluation

The DSEEP V&V overlay (IEEE 1730.2) [8] defines seven V&V phases aligned with the seven DSEEP steps. MBT activities naturally align with phases 3, 4, and 5, corresponding to design, development, and integration and testing. At this level, the mapping of MBT activities to DSEEP steps and V&V phases is straightforward. Compared to earlier mappings [12], minor refinements were made to align terminology with IEEE 29119-2 and IEEE 29119-8 and to accommodate online, on-the-fly test generation. More detailed mappings to individual DSEEP tasks are possible but offer limited additional value for describing MBT of simulation component compositions.

VI. DISCUSSION

This chapter reflects on the extensions introduced (logical time, dynamic entities) and highlights additional aspects requiring attention, such as state/time reset and ownership transfer. The chapter also identifies limitations of MBT and ComMA, including test model complexity, skill requirements, and the labour-intensive development of adapters and test models.

A. Aspects

Several simulation specific aspects were investigated in this research activity. Two important aspects are the concept of simulation or logical time, and the concept of dynamic entities. For both aspects, needed changes were identified in the ComMA protocol, and these changes were successfully implemented and tested.

Extending ComMA to support logical time for the testing of a simulator may not be limited to just that purpose. The support for logical time could also be valuable when testing software that runs on a real-time operating system. In such systems, the main processing cycle is often triggered by a periodic event, and software synchronizes with this event. If the Test Application can control this event,

synchronization issues are eliminated, and testing can happen faster or slower than the event's wall-clock periodicity.

Besides the listed aspects there are more aspects to explore and determine how these effect the ComMA test model and/or the ComMA protocol, when applied to simulation systems as opposed to cyber physical systems. For example:

- State and time re-initialisation. In a simulation the state and associated simulation time may be saved at a certain point and restored back at a later point. A cyber-physical system on the other hand operates in real world time, and time cannot be reset to a point in the past. The save and restore of state and time require further extensions to the ComMA protocol and require the ability of the Test Application to save and restore its state.
- Taking ownership of entity properties. In a simulation the right to update (a property of) an entity may be transferred between applications. I.e. the Test Application can acquire the ownership of certain entities and actively start updating these for testing purposes. This also requires extensions to the ComMA protocol.

B. Limitations

Online MBT in general, and ComMA's test application specifically, are well suited for verifying state-based or event-driven SUTs. This technology is not intended for verifying algorithms (i.e., the model behaviour and effects) or other data-driven SUTs. Because ComMA supports partial specifications, it is particularly effective for testing components, subsystems, and systems. For testing the internals of individual components other means are preferred.

In general, MBT in relation to simulation interoperability testing is most useful for testing message exchange patterns between simulation components (i.e., at a semantical level). MBT is less useful for testing the message format (i.e., at the syntactical and technical level). Syntactical and technical message verifications should be performed by the SUT adapter, avoiding the verification of these lower level complexities in the test model.

Other general limitations of MBT include the need for MBT skills and the potential complexity of the test model and language. On the other hand, experience has shown that a one-day course¹ is typically sufficient to learn the basics of the ComMA language.

C. Use of ComMA

In order to make ComMA suitable for simulations (including dynamic entities and logical time), a range of activities have been performed:

- Creation of the ComMA test model. The ComMA test model has been created completely manually; hence,

¹ The training material is part of the open-source release. More information can be found at <https://eclipse.dev/comma/tutorial/intro.html>

no creation from sources like the HLA FOM has been used. Moreover, the ComMA test model has in turn not been used to create (parts of) the SUT Adapter.

- For the case, we investigated the types, signature, interface and component models could express what needed to be modelled. Only minor additions on the project model grammar were required, i.e., added parameters concerning dynamic entities and logical time.
- Minor changes to the generated test application's Python code. Especially for dynamic entities new functionality to define clients dynamically, add clients, deleting clients and triggering a newly created client some code was added. However, these functionalities can be reused in a different context. For logical time, dealing with TAR and TAG messages and computing how much to progress in time was added.
- Minor additions to the generator that generates the test application. The added settings concerning dynamic entities and logical time were forwarded and the changes to the generated test application's Python code was captured in static (partial) Python files.
- The creation of a SUT Adapter. Although the SUT Adapter forwards messages between the Test Application and the RTI and vice versa, it came with an extensive Object Store to keep track of clients, entities, tasks, and mappings between them. Additionally, the ComMA protocol used to communicate with the Test Application and the protocol used to communicate with the RTI needed to be learned and applied.

In summary, the development of a SUT Adapter is time-consuming and error prone. Automatic generation of the SUT Adapter (particularly including the object store) based on ComMA signature model and/or the HLA FOM could resolve this.

Furthermore, the development of a ComMA test model is also time consuming. Techniques such as passive model learning [15] can alleviate this. Also test model generation from simulation design models such as the SISO Base Object Model [6] will be helpful.

D. DSEEP Overlays

Compared to the DSEEP MBT overlay presented in [12], at the DSEEP step level only a few changes were made to map the MBT activities with ComMA onto the DSEEP. The DSEEP V&V overlay provides sufficient sub-steps to allocate the MBT activities in. It was, however, noted that there is no explicit activity to implement the experimental frame that is used to obtain the V&V information, while for MBT with ComMA some development work is required, namely: the SUT Adapter and the generation of the Test Application.

VII. CONCLUSION AND RECOMMENDATIONS

This chapter summarises how automatic verification of simulation component compositions is feasible using the extended ComMA framework. It recommends scaling the

approach to more complex scenarios, standardising tools and reporting, and extend automation.

A. Research questions

The two main research questions and the key findings are as follows.

How can a composition of simulation components be automatically verified?

To support the verification of simulation components, the ComMA framework has been extended in three key ways:

- *Logical time support:* ComMA can now handle simulation-specific time progression, independent of real-world time.
- *Dynamic entities:* Test applications can create and manage entities at runtime, enabling verification in scenarios where entities appear and disappear.
- *Composed test models:* Separate test models for different interfaces and components can now be combined, supporting verification of both individual components and full compositions.

From the experiments conducted with this extended version of ComMA, the following findings were observed: automatic verification is feasible and beneficial for compositions of simulation components using online MBT and the ComMA framework. The approach relies on formal specification of interfaces and test applications generated from test models. Practical examples demonstrated the effectiveness of the methodology. Also, further work is needed to standardise reporting and automatically SUT adapter generation from a test model.

What additions to the existing verification and validation (V&V) standards for simulation are needed?

As described in Chapter V, MBT activities were mapped onto the Distributed Simulation Engineering and Execution Process (IEEE 1730) and its V&V overlay (IEEE 1730.2), showing how these activities fit into established simulation engineering processes. However, future work should evaluate these methods on larger, more complex use cases.

B. Recommendations

Future work should focus on scaling up the approach to more complex scenarios, standardising tools, and reporting, integrating the new features into open-source ComMA, and exploring additional automation and runtime monitoring capabilities. These larger use cases should also contain more complex simulation models with e.g. non-deterministic behaviour.

ACKNOWLEDGMENT

The authors acknowledge the financial and institutional support provided by the Dutch Ministry of Defence, which made this research possible. Special thanks are extended to Dr. F. van den Berg for his invaluable contributions throughout the execution of the project.

During the preparation of this work, the author(s) used Copilot to improve the language. The author(s) take(s) full responsibility for the content of this publication.

REFERENCES

- [1] I. Kurtev, J. Hooman, M. Schuts and D. van der Munnik, "Model based component development and analysis with ComMA," *Science of Computer Programming*, vol. 233, 2024, doi: 10.1016/j.scico.2023.103067.
- [2] J. Hannay, T. v. d. Berg, S. Gallant and K. Gupton, "Modeling and Simulation as a Service infrastructure capabilities for discovery, composition and execution of simulation services," *JDMS*, no. Special Issue, 2020, doi: 10.1177/1548512919896855.
- [3] F. Moradi, A framework for component based modelling and simulation using BOMs and semantic web technology, Stockholm: KTH, 2008, ISBN 978-91-7178-957-0.
- [4] I. Mahmood, A Verification Framework for Component Based Modeling and Simulation, Sweden: KTH, 2013, ISBN 978-91-7501-628-3.
- [5] IEEE 1516, "IEEE Standard for Modeling and Simulation (M&S) - High Level Architecture (HLA) (IEEE 1516)," IEEE, 2025.
- [6] SISO, "Base Object Model (BOM) Template Specification (SISO-STD-003-2006)," SISO, 2006.
- [7] IEEE, "IEEE Recommended Practice for Distributed Simulation Engineering and Execution Process (DSEEP)," IEEE 1730-2022.
- [8] IEEE, "IEEE Recommended Practice for Verification, Validation and Acceptance/Accreditation of a Distributed Simulation: An Overlay to the Distributed Simulation Engineering and Execution Process," IEEE 1730.2-2022.
- [9] S. AbdelGaber, A. Abdo, L. Abdelhamid and R. Mohamed, "PSC-CTEM Ontology to overcome pesticide paradox in Model Based Testing," *International Journal of Computer Science and Information Security (IJCSIS)*, 2021, doi: 10.5281/zenodo.5504287.
- [10] M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker and A. Pretschner, "Model-based testing of reactive systems," in *Volume 3472 of Springer LNCS*, 2005, doi: 10.1007/b137241.
- [11] M. Schuts, J. Hooman, I. Kurtev, I. Tlili and E. Oerlemans, "Online Model-Based Testing Reusing Multiple Design Models in an Industrial Setting," *The Journal of Object Technology*, vol. 24, 2025, doi: 10.5381/jot.2025.24.2.a6.
- [12] T. van den Berg, "Interoperability testing tools and techniques," in *SISO Simulation Interoperability Workshop (18F-SIW-007)*, Orlando, 2018.
- [13] IEEE, "DRAFT Software and systems engineering - Software testing - Model Based Testing (IEEE 29119-8)," IEEE, 2025.
- [14] IEEE, "Software and systems engineering - Software testing - Test Processes (IEEE 29119-2)," IEEE, 2013.
- [15] M. Schuts, J. Hooman, I. Kurtev and D. Swagerman, "Reverse Engineering of Legacy Software Interfaces to a Model-Based Approach," in *Federated Conference on Computer Science and Information Systems*, 2018, doi: 10.15439/2018f64.