Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

ARTICLE TYPE

Supplementary Material: An Infrastructure for Automated Formal Verification of System-of-Systems Architectural Models using Statistical Model-Checking

Wallace Manzano*[1] | Valdemar Vicente Graciano Neto[2] | Elisa Yumi Nakagawa[1]

[1]University of São Paulo, São Carlos-SP, Brazil
[2]Federal University of Goiás, Goiânia-GO, Brazil

Correspondence
*Wallace Manzano, This is sample corresponding address. Email: wallace.manzano@usp.br

Present Address
This is sample for present address text this is sample for present address text

Summary

Contemporary society increasingly relies on software-intensive systems, a dependency driven by the growing complexity of demands. Isolated systems, unable to meet these multifaceted requirements, have given rise to Systems-of-Systems (SoS). SoS emerge as a solution, capitalizing on the interoperability of independent systems to deliver complex solutions that individual systems cannot achieve in isolation. The inherent dynamism of SoS architecture, with constituents joining and departing as needed, results in an ever-evolving structure. SoS often underpin critical domains, where failures can have severe human, environmental, social, or financial consequences. Thus, SoS engineers bear the responsibility of ensuring that specific properties within the software architecture persist throughout operations, ensuring reliability and faultless performance. However, due to the non-deterministic nature stemming from dynamic architecture of SoS, formal verification of such properties necessitates the combined use of simulation and a statistical verifier. Recognizing the substantial costs and risks associated with implementing an SoS without rigorous architectural verification, this article's principal contribution lies in presenting the outcomes of developing and evaluating an infrastructure tailored to enable statistical and formal verification of properties within SoS software architectures. To achieve this, an interprocess communication framework was devised, facilitating seamless interaction between the MS4ME simulator and the PLASMA statistical verifier. This statistical verification capability empowers the early verification of SoS properties during architectural design, proactively addressing potential challenges and vulnerabilities. The results obtained are expected to not only advance the state of the art in SoS Software Engineering but also contribute significantly to a broader research initiative dedicated to SoS architectural design. The conducted experiment conclusively demonstrates that our infrastructure excels at conducting statistical verification for SoS dynamic architectures, achieving a high confidence interval while effectively managing computational costs at an admirably sustainable level.

KEYWORDS:
Simulation, Statistical Model Checking, Formal Verification, Software Architecture

1 | PLASMA-MS4ME INTER-PROCESS COMMUNICATION

To use our approach, the user needs to perform the following activities:

Activity 1. Building the DEVSNL models, satisfing the requirements described in Section 1.
Activity 2. Implementing the dynamic characteristics into DEVSNL models.
Activity 3. Add the MS4ME Simulation Interface with the Atomic Models and add it to the target Coupled Model. User can also define the value of boolean variable *multipleExperiments*. This variable is used so that the MS4ME Controller can be aware whether the server socket will be closed after the experiment (false value) or not (true value). Preserving the server open is useful when multiple experiments are performed on a single model; however, if the user needs to run another experiment, the user must manually close the server by injecting a value in the *inClose* port in the MS4ME Simulation Interface. The default value of *multipleExperiments* is false.
Activity 4. Building a file in the Models.txt package in MS4ME named "operators.txt" with the user-defined operators to be performed in the architecture.
Activity 5. Add the jar files into project and add their into the project build path.
Activity 6 Building a plasma model containing the line "localhost:4242:", which correspond to the address used by the MS4ME Controller to the PLASMA connect with.
Activity 7. Building a property using DynBLTL to be checker.

Activity 8. Run the simulation in the MS4ME.
Activity 9. Run the experiment in the PLASMA.
Activity 10. Analyze the probability outputted by PLASMA. Also is possible check all the operators performed in the architecture in the file "executedOperators.txt" in the package Model.txt in the MS4ME and check the execution trace in the file "trace.txt" in the same package. To start a new experiment in the same models, if *multipleExperiments* value is false, needs only perform the Activities 8 and 9 again, otherwise, needs only perform the Activity 9 again, and, in both cases, analyze the results on Activity 10. The approach for connecting PLASMA to MS4ME is detailed in Section 1.

To connect the PLASMA to the MS4ME, we established a inter-process communication that allows the PLASMA to request to the simulation in MS4ME. This communication uses network socket to establish a client-server communication between PLASMA (client) and MS4ME (server).

The remainder of this section is organized as follows: Section 1.1 presents the MS4ME platform, Section 1.2 presents the PLASMA platform, Section 1.3 presents the Simulation Plugin of PLASMA, and Section 1.4 presents the approach to handle with the inter-process communication between PLASMA and MS4ME

1.1 | MS4ME

MS4 Modeling Environment (MS4ME or MS4 Me) is a modeling and simulation environment. MS4ME simulates models described in DEVS Natural Language (DEVSNL), which is based on DEVS formalism.

Among several components held by MS4ME, its operation is based on four main objects: Coupled Model, Simulator, Simulation and Coordinator. Figure 1 shown the relationship among these objects. The responsibilities of such objects are:

- **Coupled Model**, which corresponds to the model being simulated with its respective children (Atomic Model and Coupled Model) and couplings. This object also allows add/remove models and couplings in the simulation and allowing access to the Atomic Models children.
- **Simulator**, which checks the next events of the children of the Coupled Model, injects and processes inputs/outputs through the couplings, to perform the operations that are carried out together with the Coordinator.
- **Coordinator**, which is responsible for the management of the running Coupled Model, management of children and couplings, and performing operations on the Coupled Model requested by the Simulator.
- **Simulation**, which corresponds to the simulation being executed by a Simulator. It stores information such as elapsed time, the number of steps it takes, and in addition it has the function of starting and stopping the simulation, executing a defined amount of iterations (steps).

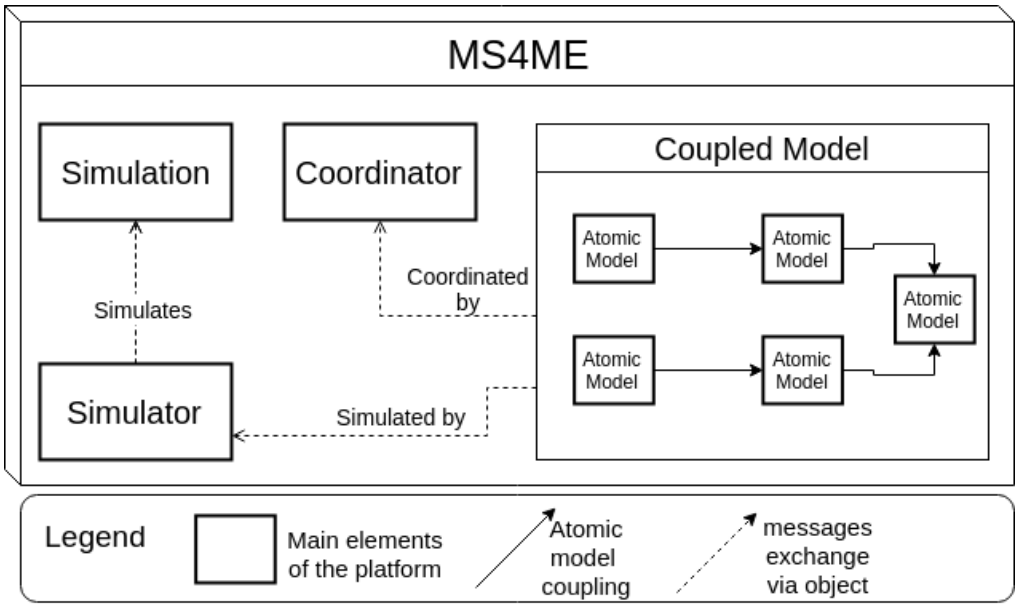


Figure 1 Main elements of MS4ME.

1.2 | PLASMA Lab

PLASMA Lab (PLASMA) is a Statistical Model Checker, featuring a customizable simulator class. PLASMA allows you to add SMC functionality to existing domain-specific modeling platforms, such as MS4ME. PLASMA allows this via plugins that can be developed by the user, using a well-defined interface. The operation of Plasma Lab is mainly based on three plugins, as shown in Figure 2:

- **Simulation Plugin**, which is responsible for providing the execution trace. This plugin establishes a connection with a domain-specific modeling platform and makes requests for the simulation so that the Model Checker can check the simulation execution trace and that the algorithm can request for new simulation to calculate the probability.
- **Model Checker Plugin**, which is responsible for checking some user-defined property in the execution trace provided by the Simulation Plugin, and requests for new states for the Simulation Plugin for check the property.
- **Algorithm**, which is responsible for calculating the probability that the model being simulated satisfies some property through the Model Check Plugin Results and performing statistical tests, and requests for a new trace for the Simulation Plugin to calculate the properties.

1.3 | Simulation Plugin Interface

PLASMA already has a specific simulation plugin; however, it should be connected via a client-server connection to a simulator. To make requests to the simulator, this plugin sends a request to the simulator, and waits for a specific response of the simulator. Figure 3 shows the class diagram of the specific request and responses illustrated.

The requests made by the PLASMA Simulation Plugin is composed by the *Request* enumerate and their respective response is represented by a object that has *Response* as its superclass. The response by each enumerate item are:

- **INIT**, in which the server-client connection is initiated by the client, and the plasma server returns an *Init* object stating that the connection was successfully made.
- **NEW_TRACE**, which informs the server that the client needs to start a new simulation, and the server returns a *TraceStart* object stating that the simulation was successfully started.
- **NEW_STATE**, which informs the server that the client needs more states for model checker plugin analyze the property and the server responds with an Events object that contains the Event that happened in the new state.

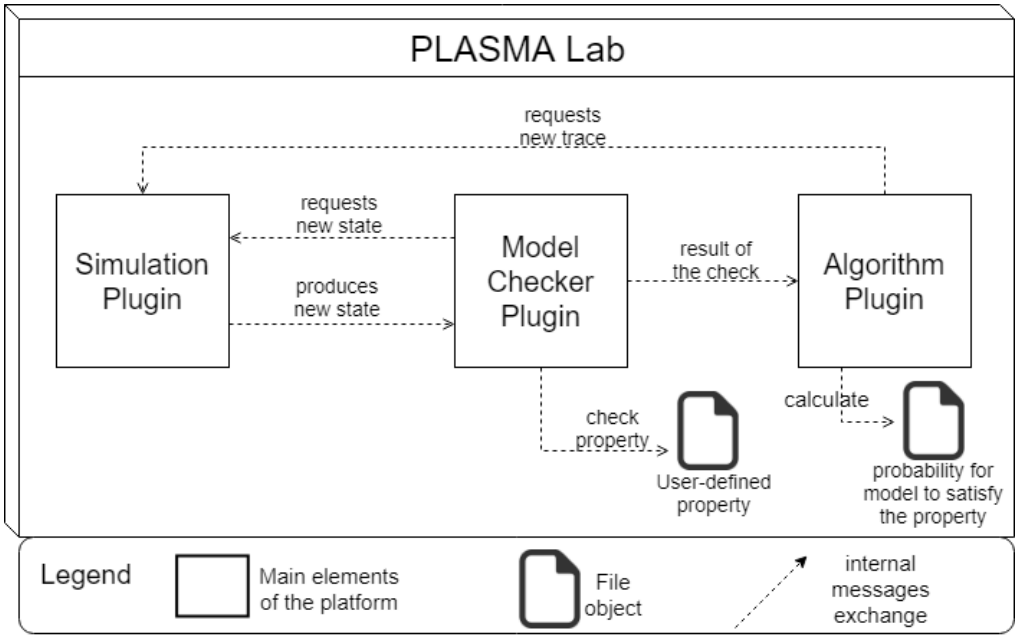


Figure 2 Main elements of PLASMA.

The possible Events that the client needs in the NEW_STATE request are:

- **NewNode**, which corresponds to the creation of a child model in the Coupled Model, being necessary to inform its name, type and their respective ports.
- **NewLink**, which corresponds to the creation of a coupling in the Coupled Model, being necessary to inform its ports and their respective owners.
- **RendezVous**, which corresponds to the data exchanges by coupling. For that, it is necessary inform the ports with their owners and the data exchanged.
- **Done**, which corresponds to the removal of a child from the Coupled Model, being necessary to inform the model name.

1.4 | MS4ME-PLASMA Communication

To receive requests of Simulation Plugin, we developed the MS4ME Controller, which is a jar file required to be added in the build path of simulation project. MS4ME Controller is responsible for receiving the requests and cooperate with the Simulation to execute that request. However, the controller is not automatically added to the simulation. For addressing that, we developed the MS4ME Simulation Interface, which is an Atomic Model inside the Coupled Model that establishes an interface between the controller and the current simulation. That Atomic Model also is responsible for creating the MS4ME Controller in a parallel thread when the simulation is started and establish a bridge between the MS4ME Controller and the M4Me objects of the current simulation. In addition, the MS4ME Simulation Interface receives events produced by the other Atomic Models and transfers them to the MS4ME Controller. To do this, Atomic Models triggers a *NewNode* event whenever they are started, and whenever they receive some data they trigger a *RendezVous* event.

Whenever the controller receives a NEW_TRACE request, the controller performs a set of user-defined dynamic reconfiguration operators to perform changes on the architecture configuration under simulation. To manage the user-defined operators, we established the Operator Manager. The Operator Manager reads the user-defined file with the operators. The file should satisfy the syntax shown in Figure 4. The architect can define each operator that will be performed in each operator request, or define an amount of random operators that the Operator Manager can generate randomly at each operator request and the operators that can be performed with its probability. In that manner, each time that the controller receives a NEW_TRACE request, the controller requests to the Operator Manager the next operators to be performed.

MS4ME Controller also needs a Dynamic Reconfiguration Controller (DRC) that can execute the dynamic reconfiguration operators in the coupled model under simulation. DRC can be generated as result of the model transformation used in Dynamic-SoS. To execute the MS4ME

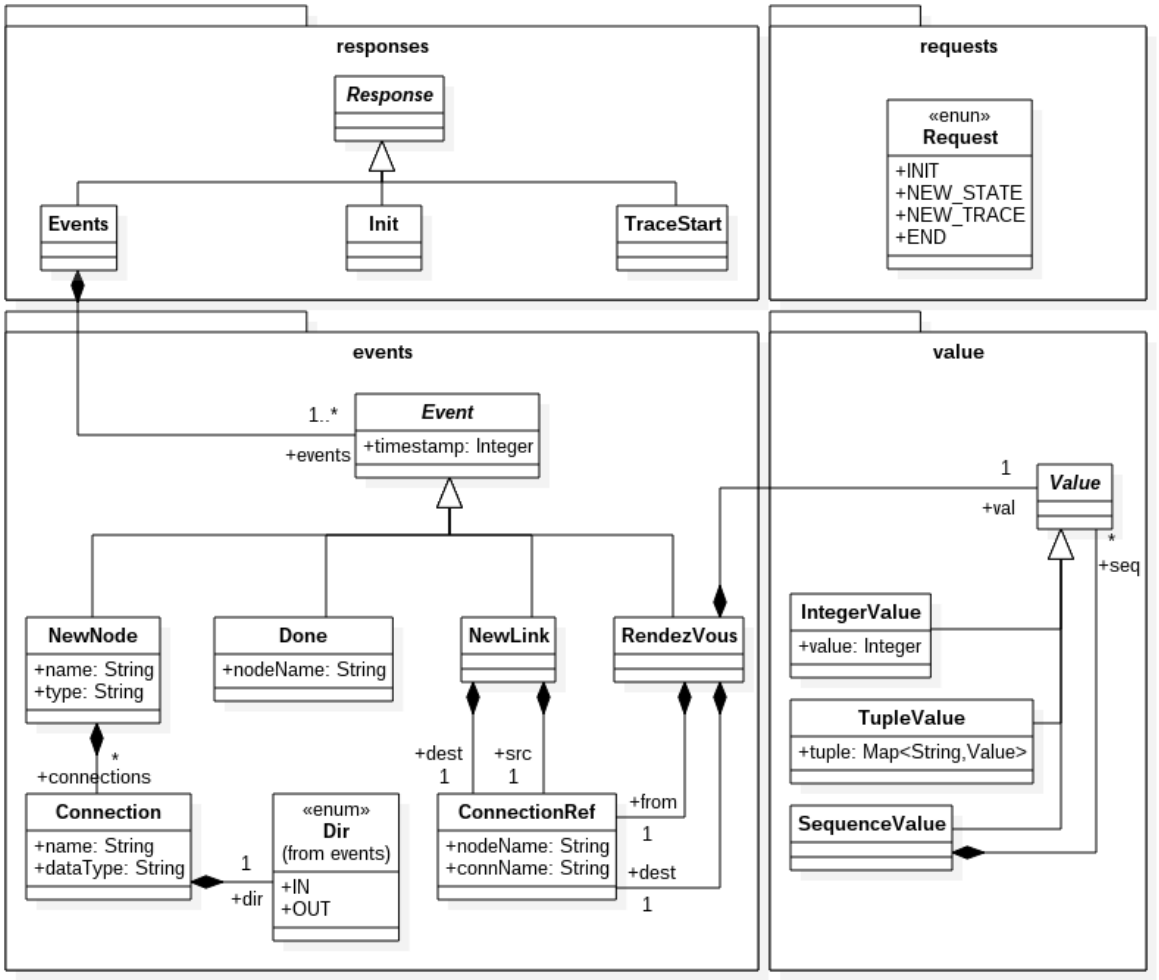


Figure 3 Plasma interface class diagram.

Controller operations, the DRC demands the name to be equals to “DRC + Coupled Model name”. Due to that, a specific DRC is required for each architecture. Hence, to provide the MS4ME Controller independence, we can only determine the DRC class at runtime. Thus, to request for operators, the MS4ME Controller communicates with DRC through Java Reflection. For those purposes, the DRC implements the following methods:

- `additionOperator(String)`, which receives a string with the constituent type name and add it to the simulation.
- `deletionOperator(AtomicModel)`, which receives the Atomic Model to be deleted and remove it from the simulation.
- `reorganizationOperator()`, which reorganizes the architecture under simulation generating a new architecture configuration.

The MS4ME Controller also needs that each Atomic Model that represents a constituent, a mediator or a stimuli generator has a field named “typeCons” with your time name (e.g. Sensor, Transmitter, StimuliGeneratorSensor). All Atomic Models generated by ASAS already have this field.

Figure 5 shows the communication with the elements on MS4ME and the PLASMA Lab.

Figure 6 shows the necessary activities to accomplish the INIT request. To do this, the MS4ME driver should establish a connection to the Simulator running on the PLASMA. After that, the simulator sends an INIT request to the controller, and the controller requests the current Coupled Model in simulation for the MS4ME Simulation Interface and the controller remake the Coupled Model undoing all possible changes on initial architecture configuration, finally returning an *Init* object.

The diagram depicted in Figure 6 shows the necessary activities to accomplish the NEW_TRACE request. To do this, the Simulator uses the connection already established on INIT request and sends the request. After that, the controller requests for the current simulation to the simulation

$Mode \rightarrow ByTrace \mid Aleatory$

$ByTrace \rightarrow$ 'in trace' $Trace$ 'execute operators' $Changes$ $(ByTrace)$?

$Aleatory \rightarrow$ 'for each trace execute' $Amount$ 'aleatory operators using' $ProbOperator$

$ProbOperator \rightarrow Operator$ 'with probability' $Probability$ $(ProbOperator)$?

$Changes \rightarrow Operator$ (',' $Operator)^*$

$Operator \rightarrow Addition \mid Removal \mid Reorganization$

$Addition \rightarrow$ 'add' $Name$

$Removal \rightarrow$ 'remove' $Name$ $RemovalType$

$Reorganization \rightarrow$ 'reorganization'

$RemovalType \rightarrow \{ConstituentName\}$ 'by name' $\mid \{ConstituentType\}$ 'by type'

$Trace \rightarrow integer$

$Amount \rightarrow integer$

$Probability \rightarrow float$

Figure 4 Expected syntax of user-defined operators file.

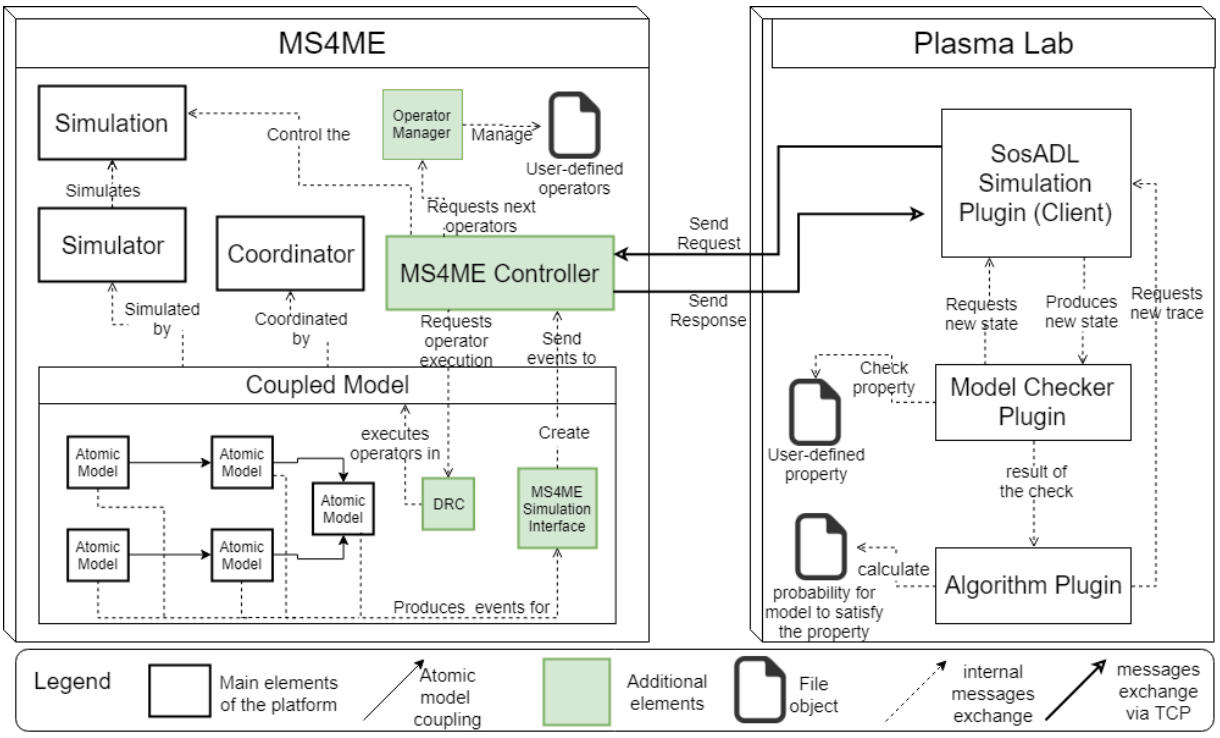


Figure 5 Approach architecture.

interface and restarts the simulation. The Restarting process is performed by the Simulation object in MS4ME that re-initialize its fields, the simulation trace, and re-initializes all the Atomic Models in the Coupled Model. In the re-initialization process, the Atomic Models send *NewNode* events to the simulation interface that forwards them to the controller. The controller also requests the simulation interface the current Coupled Model to trigger the *NewLink* events. With the simulation restarted, the controller requests to the Operator Manager the next operators to be performed on the architecture and the controller sends the request of each operator so that the DRC can execute it, and finally returns a *TraceStart* object to the Simulator.

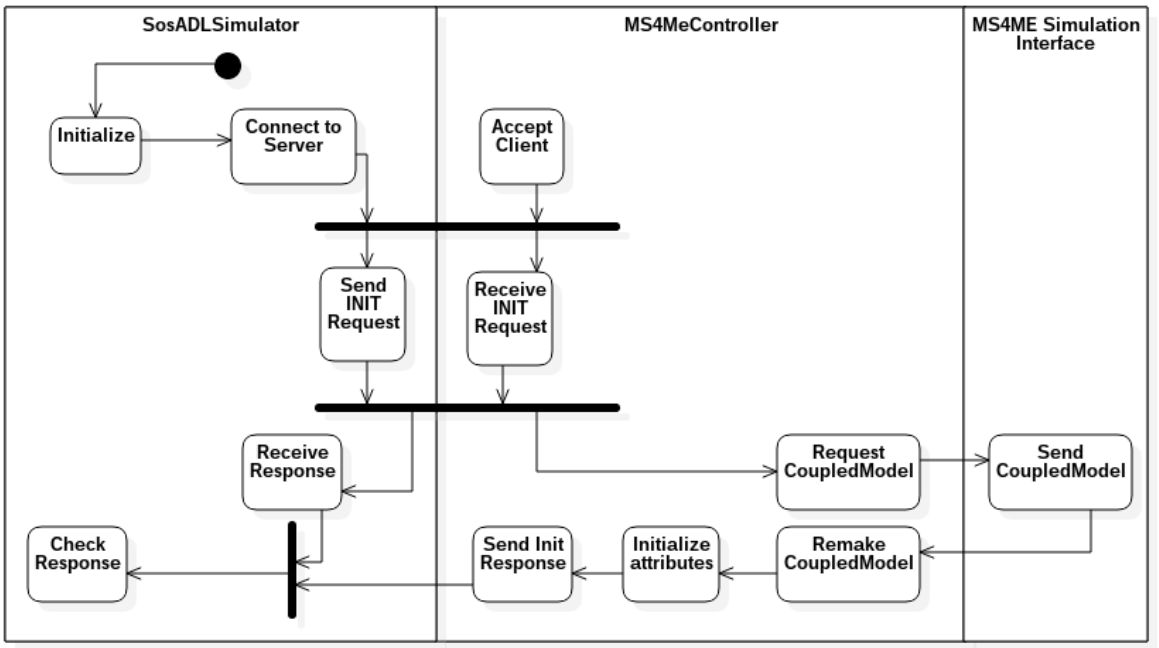


Figure 6 Init request activity diagram.

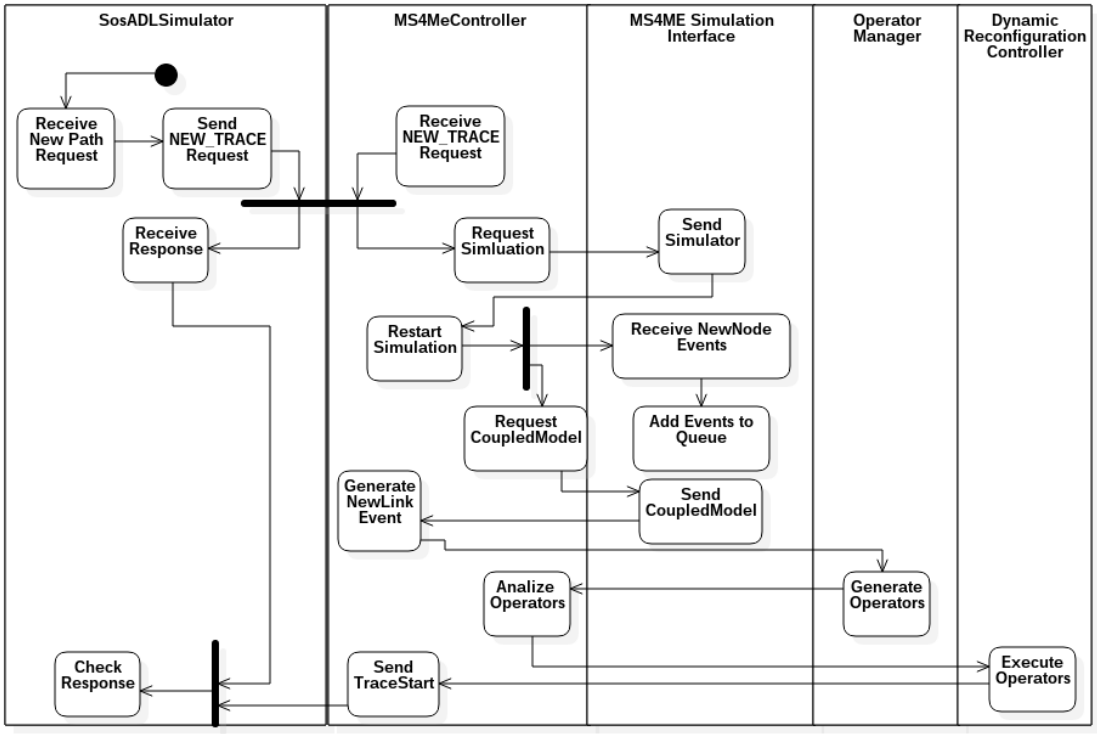


Figure 7 New trace request activity diagram.

Simulator sends only a NEW_STATE request to the controller if it has no event to process. If it has it, it processes that event. Once the Simulator sends a request to the controller, the controller needs to request the current simulation to the simulation interface and execute one interaction at a time until one of these interactions despairs events. With these events, the controller sends a response of type Events containing all the events produced. With these events, Simulator takes the first of these events and processes it. The activity diagram referring to this process can be seen in Figure 8.

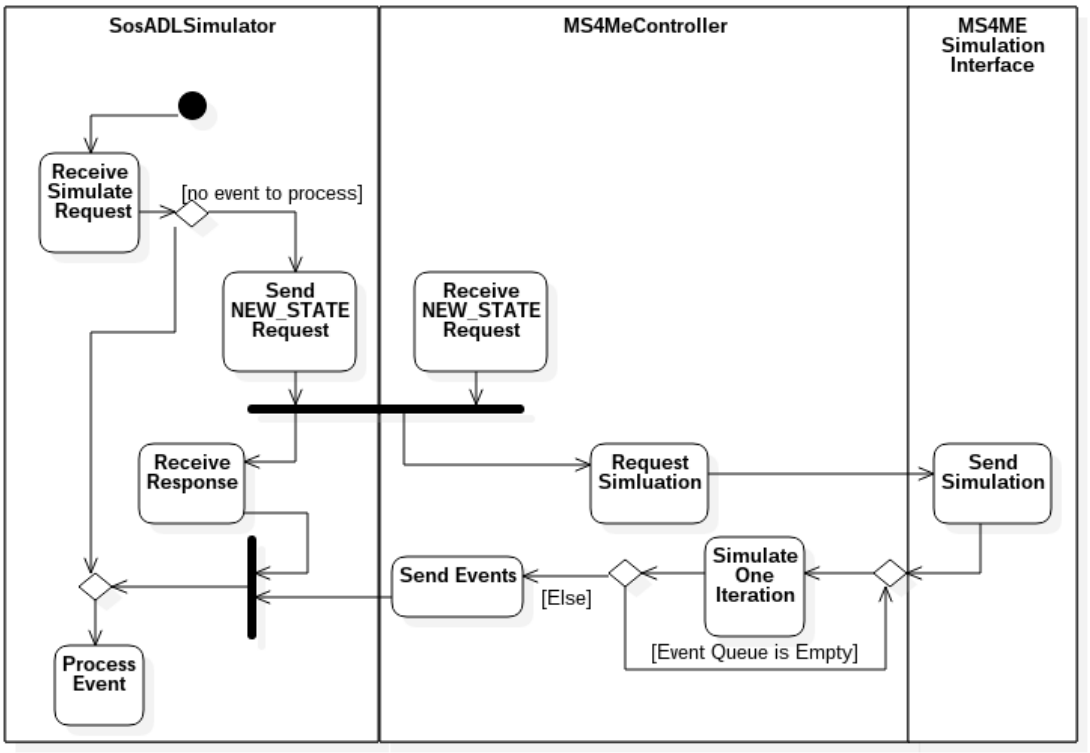


Figure 8 New state request activity diagram.

How to cite this article: W. Manzano, E. Y. Nakagawa, and V. V. Graciano Neto (2023), An Infrastructure for Automated Formal Verification of SoS Architectural Models using Simulation and Statistical Model-Checking, *Journal of Software: Evolution and Process*, *2023;00:1–6*.