A LIGHTWEIGHT, FEEDBACK-DRIVEN RUNTIME VERIFICATION METHODOLOGY

by

Christopher James Lynch
B.S. May 2011, Old Dominion University
M.S. August 2012, Old Dominion University

A Dissertation Submitted to the Faculty of
Old Dominion University in Partial Fulfillment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY

MODELING AND SIMULATION

OLD DOMINION UNIVERSITY
August 2019

ABSTRACT

A LIGHTWEIGHT, FEEDBACK-DRIVEN RUNTIME VERIFICATION METHODOLOGY

Christopher James Lynch
Old Dominion University, 2019
Director: Saikou Y. Diallo

Runtime verification facilitates error identification during individual simulation runs to increase confidence, credibility, and trust. Existing approaches effectively convey history, state information, and flow-of-control information. These approaches are common in practice due to shallow learning curves, lower mathematical requirements, and interpretation aided by observable feedback that provides context to the time and location of an error. However, runtime techniques lack consistent representation of the models' requirements and the attention-demanding process of monitoring the run to identify and interpret errors falls on the user. As a result, these techniques lack consistent interpretation, do not scale well, and are time intensive for identifying errors.

To address these shortcomings, this dissertation develops a lightweight, feedback-driven runtime verification (LFV) methodology consisting of three components: simulation platform instrumentation; user-defined specifications; and viewing simulation runs. The LFV methodology provides a formal specification that facilitates clear and consistent mappings between model components, simulation specifications, and observable feedback. These mappings are defined by simulation users, without requiring knowledge in formal mathematics, using the information available to them about how they expect a simulation to operate. Users specify values to simulation components' properties to represent acceptable operating conditions and assign feedback to represent any violations. Any violations within a run trigger the

corresponding feedback and direct users' attention toward the appropriate simulation location while tracing back to the assigned specification. A formal specification adds transparency to error specification, objectiveness to evaluation, and traceability to outcomes.

A two-group randomized experiment compares the LFV against a traditional runtime method of animation with operational graphics to provide initial insight into the effectiveness of the LFV. A statistically significant increase in precision (i.e. the proportion of correctly identified errors out of the total errors identified) and recall (i.e. the proportion of correctly identified errors out of the total errors present) is observed for participants' using the LFV. The use of attention grabbing feedback alongside clearly defined specifications results in more concise error identification.

The LFV opens new research areas for runtime verification of large-scale and hybrid simulations, occluded simulation components, and exploring the role of different feedback mediums in support of verification.

This dissertation is dedicated to my wife, Jenny, whose support and understanding made this dissertation a reality.

ACKNOWLEDGMENTS

A great many people contributed to the success of this dissertation through their constant support, advice, and guidance. I extend many, many thanks to my committee members for their patience and years of guidance on my research. As my major advisor, Dr. Saikou Diallo was instrumental in the development of this research, meeting the ODU IRB requirements, and the shaping of this document. Dr. Jose Padilla contributed extensive time and effort reviewing multiple drafts of this document, survey designs, and presentations. Dr. John Sokolowski, Dr. Roland Mielke, and Dr. Navonil Mustafee each contributed volumes of critical comments and insightful guidance that helped in scoping this research and achieving the research objectives over the past four years.

Thank you to my friends that participated in the journey within the Computational Modeling and Simulation department. Significant appreciation goes to Hamdi Kavak and Daniele Vernon-Bido for our weekly meetings that helped me in maintaining momentum and focus towards the development and completion of this research.

A special thanks to all of my colleagues at the Virginia Modeling, Analysis, and Simulation Center (VMASC) for years of lively discussions on model development, simulation implementation, verification and validation approaches, multi-paradigm modeling, analysis techniques, and teaching simulation concepts.

Anthony Barraco provided background knowledge and support on the CLOUDES platform that was necessary in integrating the LFV Java and JavaScript code with the platform to allow for testing. This allowed for the creation of the Use Cases verified using the LFV that were developed as part of this study. Dr. Ross Gore and Menion Croll both provided helpful advice and guidance from the Computer Science perspective on overcoming programming challenges in efficiency and data volume during the instantiation of the LFV.

Thank you to the many people that participated in the pilot study, the post-pilot discussions, and in the full study. All of the data provided by these activities greatly strengthened the findings from this research.

NOMENCLATURE

A	Set of outputs
B	Set of feedback mediums
C	Set of model components
$E_M(t) = \bigcup_{e=1}^{\lvert E \rvert} E_e(t)$	Set of entities in the model at time *t*
$E_e(t) \subseteq E_M(t)$	Set of entities of type *e* in the model at time *t*
$E_N(t) \subseteq E_M(t)$	Set of entities in the current node at time *t*
$EM_e(t)$	Total entities of type *e* currently in the model at time *t*
$EN_{e,n}(t)$	Total number of entities of type *e*, at the current node *n*, at time *t*
P	Set of model component properties
R	User defined runtime verification specifications (subset of *V*)
$RM_r(t)$	Total resources of type *r* currently in the model at time *t*
$RN_{r,n}(t)$	Total number of resource *r*, at the current node *n*, at time *t*
S	Set of possible runtime violations
T	Set of time
TD(t)	Total number of departures from all exit nodes at time *t*
$TR_r(t)$	Total number of resource type *r* currently released at time *t*
$TS_r(t)$	Total number of resource type *r* currently seized at time *t*
V	Set of verification specifications
W	Set of actual violations occurred for a simulation run (subset of *S*)
ϕ	Set of operators

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1
INTRODUCTION

Models and simulations connect people to problems, systems, and phenomena of interest to enhance our understandings and interpretations of their corresponding causes, outcomes, and mechanisms. A *model* serves as the blueprint for a simplified representation of the phenomena of interest at the necessary level of abstraction to address the questions driving the study. The model provides the intended design, specifications, and behaviors of the phenomena of interest. A *simulation model* is the executable implementation of the model that produces observable behaviors and outcomes for the model. These in turn provide the foundations for learning and interpretation through experimentation. *Simulations* are the individual instances of the executed simulation model for a given set of inputs, random number generator seed, and run length and their corresponding sets of outputs. As models are the simplifications of the phenomena of interest and the simulation runs are generated from these simplifications, simulation models undergo processes to determine that the implementation matches the intended design and that the simulation's outcomes are an acceptable representation of the modeled reality. These processes fall under the purview of Verification and Validation (V&V)[1].

The accepted definitions that distinguish the roles of verification and validation within the Modeling and Simulation (M&S) discipline are: verification determines that the simulation has been implemented correctly (answering the question of whether the simulation works as intended); and validation determines if the correct model has been built (answer the question of whether the model answers the right questions). V&V occurs iteratively throughout the development and implementation process of a simulation model, as illustrated through numerous

[1] IEEE Transactions and Journals style is used in this thesis for formatting figures, tables, and references.

modeling frameworks [1-3]. Testing can occur often and only shows that the simulation did not fail a test under the conditions utilized for the test. Passing a verification or validation test does not prove that the simulation is free of errors or that it is a perfect representation of the modeled phenomena.

The verification process includes identifying the presence of errors within simulation models, isolating their sources, and correcting them [4-8]. Errors can originate from numerous occurrences, including: mistakes in implementing a simulation model's code; mistakes in constructing the model structure through the simulation platform's user interface (UI); and mistakes in initializing the parameters and input values of the simulation. Errors can lead to cascading effects throughout a simulation which can help to reveal its existence at multiple points within the simulation. Cumulatively, multiple components within the simulation being effected can make it easier to identify that an error exists, but can complicate the identification of its origin point. The observable effects resulting from an error represents its manifestations [9] and may appear at one or more multiple locations other than its source. Conceptually, this mimics the definition given by Ammann and Offutt [10] that software errors are manifestations of software faults, with software faults being the tangible instantiations of design mistakes.

Due to the volume of terms in use, the relevant terminologies pertaining to simulation verification used in this research are defined as follows:

Definition 1.1: *A* ***simulation*** *is the execution of an implemented simulation model.*

Simulation allows for conducting experiments with the purpose of understanding the represented system's behaviors [11]. Models can be static (time independent) or dynamic (time dependent) with respect to the connections between the advancement of the model and the advancement of the simulated time [12-14]. Whether a model is dependent upon time effects how and when potential errors can be identified. Models can also be classified as stochastic (multiple sets of outcomes for a given input configuration) or deterministic (the same outcomes given an input configuration) with respect to the presence of randomness within the system [7, 15, 16]. Randomness can require larger numbers of runs with the same input configuration in order to produce a run where an error is present.

Definition 1.2: *An* **error** *is any occurrence that violates its specifications.*

The above definition of error is consistent with the literature in that determining what constitutes an error relies on the intended use of the simulation [17-19] and pertains to the mismatch between what is expected versus what actual exists within the simulation [20, 21]. A model's intended use and its defined specifications can help direct verification efforts; however, different types of techniques are available dependent upon the test being applied.

Techniques for verifying simulations are commonly classified as static or dynamic and they range in their levels of mathematical and logical formality [18, 22]. Static techniques are commonly applied to a simulation's code without running the simulation and are well-suited for identifying syntactic and structural errors. Dynamic techniques are generally applied during the simulation's execution and are well-suited for identifying logical errors. Static techniques often provide the source of an error (since the code is directly being examined) while dynamic techniques are proficient at revealing the results of errors but not necessarily in revealing the source of the error directly.

Static techniques may often be better suited for identifying the root cause of an error; however, this comes at a tradeoff with higher time and resource requirements than dynamic techniques. Time and resource increases result from the higher levels of formal mathematical background required of the simulation tester (such as using model checking) and from larger numbers of people being needed to conduct a test (such as audits or code reviews). Techniques that are more mathematically informal, have lower learning curves, and are dynamic have been found to be more commonly applied in practice [23, 24]. The category of dynamic verification techniques applied at runtime is the central focal point of this research. As such, the definition of a runtime error follows.

Definition 1.3: *A* ***runtime error*** *is a violation of any specification that manifests during a simulation's execution.*

Under this definition, the ability to identify runtime errors involves (1) knowing what the model is expected to do and (2) having access to what the simulation is actually doing. The combination of these two elements allows for the identification of violations while the simulation is running. The absence of runtime errors only indicates that the model was not shown to violate any of its specifications under the given instantiation (i.e. under the parameters of the run) for the given time frame (i.e. a simulation run length of 20 weeks) for the given verification test (i.e. the model components that are being investigated under the test). As a result, the ability to conduct runtime verification relies on identifying an error's manifestations.

Definition 1.4: *The* ***manifestation of an error*** *is* an observable representation of the error's occurrence at some location and time during the simulation run.

This definition reflects temporal and spatial connections between an error and its observable representations during runtime. Note that the location of an error's manifestation and the source location of the error are not necessarily the same [9, 10]. The goal is to first identify that an error is present before isolating the exact error. The process of identifying the source of a suspected error is handled differently than identifying its existence and may rely more on static techniques. Spatial connections refer to the exact locations of the errors within the simulation's code, input space, simulation environment's icons (i.e. nodes, charts, consoles, etc.), and/or any visualizations provided; these do not refer to any geographical mappings between the simulated entities within the simulated environment. For runtime verification, within-run analyses utilizing descriptive statistics [25] can assist in identifying error manifestations by illuminating unexpected statistics within the run.

This work is concerned with identifying runtime errors by recognizing their manifestations. The following four criteria are required to allow for the identification of errors:

1. *Existence*: an error must exist;
2. *Occurrence*: when an error exists, it must occur;
3. *Observability*: when an error occurs, it must manifest in an observable form;
4. *Identification*: while an error is observable, it must be observed.

These points are related to the concepts of reachability, infection, and propagation within software engineering [10]. Reachability indicates that the location of the code containing the error must be reached. Infection indicates that the corresponding state of the code must be incorrect once reached. Propagation indicates that the infected state must propagate as an observable output.

Conducting runtime verification requires the ability to examine simulation components throughout execution. Simulation environments provides some values or statistics on screen by default, such as the number of resources available in Arena [26] or the number of entities waiting in a queue in CLOUDES [27]. Other desired values or statistics may not automatically display at runtime or values may display only in threshold or tier representations, such as 10+ or high/low. Therefore, runtime techniques commonly require the ability to instrument the environment so that the desired values can be accessed and made observable at runtime.

***Definition 1.5: Instrumentation** is the insertion of additional code or monitors into the executable model to analyze and reveal information during execution [18, 22].*

Instrumentation involves two distinct but complementary efforts: making the desired simulation components observable; and conducting the appropriate analyses so that the desired information can be revealed. Revealing the information requires considerations for how the information is conveyed to the model users, such as through numeric monitors, charts and plots, or sensory feedback (such as through visual, audial, or tactile feedback). Analysis considerations involve how simulation components are examined (i.e. the mathematical or logical checks to use), how to make determinations that errors exist, and handling any mathematical calculations or logical checks that are employed as part of these analyses. For social simulations, inserting redundant code assists in identifying errors and assists in locating their sources [28].

An absence of observed errors does not indicate any of the following:

- The absence of observed errors *does not* suggest that the simulation is absolutely error free even under the boundaries of the test, particularly for stochastic simulations;
- The absence of observed errors *does not* indicate that the simulation's conceptual models are correct;
- The absence of observed errors *does not* indicate absolute adherence to the simulation's conceptual models or reference models;
- The absence of observed errors *does not* suggest that the verification test applied was setup correctly (the design and application of a test may also need to be tested for correctness); and,
- The absence of observed errors *does not* suggest that the model is a valid representation of the system that it was created to represent.