Proceedings of the 2021 Winter Simulation Conference
S. Kim, B. Feng, K. Smith, S. Masoud, Z. Zheng, C. Szabo and M. Loper, eds.

A FRAMEWORK FOR SUPPORTING SIMULATION OF MBSE MODELS

Nicholas Engle
Michael E. Miller

Air Force Institute of Technology
2950 Hobson Way (AFIT/ENV)
Wright Patterson AFB, OH 45433 USA

ABSTRACT

Application of Model-Based Systems Engineering (MBSE) requires the integration of robust tools to provide system description and dynamic model execution. MBSE tools, such as Cameo Systems Modeler facilitate robust system descriptions. While these tools often have built-in simulation capabilities, they are often limited in scope and lack features for robust modeling of system behavior and performance. We present a prototype toolkit which extends SimPy, an open-source Python-based Discrete-Event Simulation library. Using custom SysML stereotypes for defining object, environment, and activity properties, our prototype extracts information from Cameo model files to build and run a graph-based Python discrete event simulation.

1 INTRODUCTION

Model-based Systems Engineering (MBSE) is being adopted across the United States Department of Defense and Industry. This process uses standardized graphical modelling languages, such as the Systems Modeling Language (SysML), to support design and communicate system requirements, structure, and behavior (Delligatti 2013). MBSE tools, such as Cameo Systems Modeler (CSM), support the development of large SysML models, tracking objects defined in models, associating objects with system behavior, and propagating changes between linked diagrams.

The SysML Activity Diagram defines a sequence of actions to be performed. These actions may be base-level actions, control actions, such as Forks, Joins, and Decision nodes, or nested activities. Actions and edges between actions may have custom-defined properties, and may be *allocated* to objects, permitting representation of a series of tasks performed between different actors. Activities are a natural target for simulation. Unlike designing our process directly in the simulation environment, SysML allows us to integrate the process into the larger system design, seamlessly modeling interactions between elements.

Integrating SysML tools and the Arena Discrete Event Simulation (DES) environment has been discussed previously (Batarseh and McGinnis 2012). For our application, integration with an open source environment permits the extension of the models to robustly consider human constraints, such as the human's available mental capacity, which are not supported by most traditional DES environments. Therefore, we chose to attempt integration with SimPy due to its relative simplicity, level of development, and utility. SimPy dates to 2002 and it has been re-implemented in several other languages (Julia, C#, R), providing community design support and allowing porting to these languages (SimPy 2020). SimPy's API is similar to standard python, making it familiar to Python developers. This allows the simulation section to be moderately agnostic of SimPy's API specifics, making it feasible to re-implement in new frameworks.

2 ENVIRONMENT DESCRIPTION

Due to the potential complexity of SysML models, this project uses a standard 'interface' for defining models to be simulated using the process shown in Figure 1. As shown, the model is defined in a SysML

tool, i.e., CSM, a custom data reader converts the resulting SysML to provide a simulation definition, and the model is executed in SimPy.

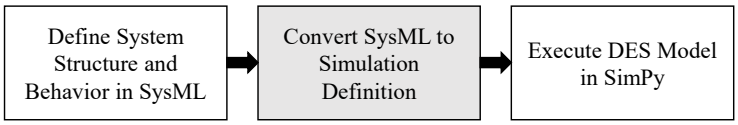

Figure 1. Process flow for DES definition and execution.

CSM uses Stereotypes to extend the definitions of core components, allowing user-defined versions of any element. The "Actor" block may be placed on any Block Definition Diagram and defines an actor that performs actions. This is where environmental parameters are defined, such as weather conditions or worn gear. The next stereotype extends default SysML actions allowing action specific parameters, such as nonstandard probability distributions, delays, or completion times and their associated parameters. Finally, the last stereotype extends the Activity Diagram to point the data reader towards relevant activities. This interface is intended to be general and not depend on one specific model structure. To support this, the data reader scans the whole model for relevant elements rather than expecting elements in specific locations.

The data reader extracts model data to Python-native containers which are provided to the graph builder. This creates a directed graph of the model with actions and control elements as *nodes,* and element connections as *edges*. Each of the basic nodes is represented as a Python class which holds associated data, outgoing edges, and a run() process representing a node's execution. In general, when a node's run() is invoked, it accepts an *Environment* object allowing access to resources shared between nodes, and utilities such as the current sim time. The run() process also accepts an *execution token*, storing trial-specific state information and statistics. Action nodes use this information and Actor-specific environmental data to calculate their time to complete and determine action pass or failure. If the action fails it may terminate the current trial or direct execution to an alternate edge.

While this is the standard execution of nodes, control elements require special definition. For example, *Fork* nodes allow execution to traverse two different paths before eventually re-joining at a corresponding *Join*. The Fork creates an execution token for each path, storing information necessary for the Join to pair tokens later. Paths may vary in time to complete, so the Join must "collect" execution tokens, waiting until all routes have re-combined before continuing. Decision nodes also require special attention. To allow for probabilistic path modeling, we use the SysML *Probability* parameter to assign each edge a weight. The Decision node scans its outgoing edges at runtime and randomly chooses a path based on the weights assigned to the individual paths.

3 RESULTS AND CONCLUSIONS

We demonstrate initial methods for connecting MBSE and DES tools the techniques using relatively modest profiles to constrain the MBSE model. Therefore, this work may provide templates for future work using other tools in the same disciplines. The authors intend to extend this work to provide robust human representations in digital engineering to support advanced user-system modeling.

4 DISCLAIMER

The views in this article are those of the authors and do not necessarily reflect the official policy or position of the Department of the Air Force, Department of Defense, nor the U.S. Government.

REFERENCES

Batarseh, O., and L.F. McGinnis. 2012. "System Modeling in SYSML and System Analysis in ARENA." In *Proceedings of the 2012 Winter Simulation Conference*, edited by C. Laroque, J. Himmelspach, R. Pasupathy, O. Rose, and A.M. Uhrmacher, 2924–35. http://ieeexplore.ieee.org.etechconricyt.idm.oclc.org/xpl/articleDetails.jsp?arnumber=6465229&tag=1.

Delligatti, Lenny. 2013. SysML Distilled: A Brief Guide to the Systems Modeling Language. Addison-Wesley.

SimPy. 2020. "SimPy History & Change Log." 2020. https://simpy.readthedocs.io/en/latest/about/index.html.