A Formal Approach to Modeling and Analyzing Human Taskload in Simulated Air Traffic Scenarios

Adam Houser, Lanssie Mingyue Ma , Karen Feigh *Member*, *IEEE* and, Matthew L. Bolton, *Member*, *IEEE*

***Abstract*—In complex systems, like the modern air traffic system, human operator taskload can have a profound influence on how well the system performs. However, because of the system complexity, it can be difficult to determine all of the situations where taskload issues can arise. Simulation and formal verification have been used separately to explore human taskload in complex systems. However, both have problems that limit their usefulness. In this paper, we describe a formal modeling architecture designed to enable the discovery of interesting human operator taskload conditions though the synergistic use of formal verification and simulation. This architecture formally represents original simulation constructs using computationally efficient abstractions that ensure the temporal and ordinal relationships between simulation events (actions) are represented realistically. Taskload for each agent is represented based on a priority queue model where only a limited number of actions can be performed or remember by a human at any given time. We provide on overview of this architecture, discuss its essential features, and describe the mathematical foundations needed for its instantiation. We present insights into its capabilities for finding interesting taskload conditions by formulating several checkable specification properties. The implications of this architecture are discussed in terms of the broader supported analysis method and direction for future work are explored.**

***Index Terms*—Formal methods, taskload, workload, simulation, human-automation interaction.**

I. Introduction

HUman operator taskload, a measure of the number of tasks a human operator is expected to perform at a given time, is critical to the safe and efficient operation of complex system such the air traffic system. This is because taskload is a good indicator of human operator workload [1], where excessive taskload/workload leads to human error and reduced performance. However, determining when taskload can become excessive and what the performance implications of that taskload are can be very challenging because of the many different people, machines, and environmental conditions that can interact during their operation. Running experiments or tests with real world systems and human subjects can be too time consuming and expensive to explore the many operating conditions that can occur. To address this, researchers have been building simulation environments such as Work Models that Compute (WMC) [2] that allow human operator taskload to be analyzed in a variety of air traffic simulations. While more flexible than human subject experiments and real world test, these types of simulation are not exhaustive and can thus still miss potentially dangerous or performance-critical operating conditions that did not happen to occur in one of the explored scenarios. Formal verification represents an analysis approach that specifically addresses this limitation of simulation.

Adam Houser is with the Department of Systems and Industrial Engineering, University at Buffalo, State University of New York, Amherst, NY

Lanssie Mingyue Ma is with the Daniel Guggenheim School of Aerospace Engineering in Computational Science and Engineering, Georgia Institute of Technology, Atlanta, GA

Karen Feigh is with the Daniel Guggenheim School of Aerospace Engineering, Georgia Institute of Technology, Atlanta, GA

Matthew L. Bolton is with the Department of Systems and Industrial Engineering, University at Buffalo, State University of New York, Amherst, NY 14260 USA e-mail: mbolton@buffalo.edu

A. Formal Verification

Formal verification is an analysis technique that falls with the discipline of formal methods. Formal methods are well-defined mathematical languages and techniques for the specification, modeling, and verification of systems [3]. Specification properties mathematically describe desirable system conditions. Systems are modeled using mathematically-based languages. Verification then mathematically proves whether or not the model satisfies the specification. Model checking is an automated approach to formal verification [4]. In model checking, a formal model describes a system as a state transition model: a set of variables and transitions between variable states. Desirable specification properties are usually represented in a temporal logic [5]. Verification is performed automatically by exhaustively searching a system's statespace to determine if these properties hold. If they do, the model checker returns a confirmation. Otherwise, a counterexample is produced, which shows how the specification violation occurred as a trace through the statespace of the model.

Formal verification has been used successfully to evaluate human-automation interaction in different capacities [6]. However, little work has been done to investigate human operator workload or taskload. While Mercer and Goodrich et al [7], [8] have investigated ways of formally modeling workload, they have not used it in formal verification analyses.

Further, while powerful, formal verification techniques like model checking suffer from combinatorial explosion, where the statespace grows exponentially as additional components are added to the model [4]. This can quickly lead to a situation where the model is too big to be verified. Model checking is also limited by the expressive power of its notations, where models cannot contain non-linear arithmetic or other typical programming constructs (such as loops or type casting). As such, formal verification, in contrast to simulation, scales badly and is limited in what system behavior it can consider.

B. Formal Verification and Simulation

Some degree of success has been found in using formal verification synergistically with simulation to exploit the

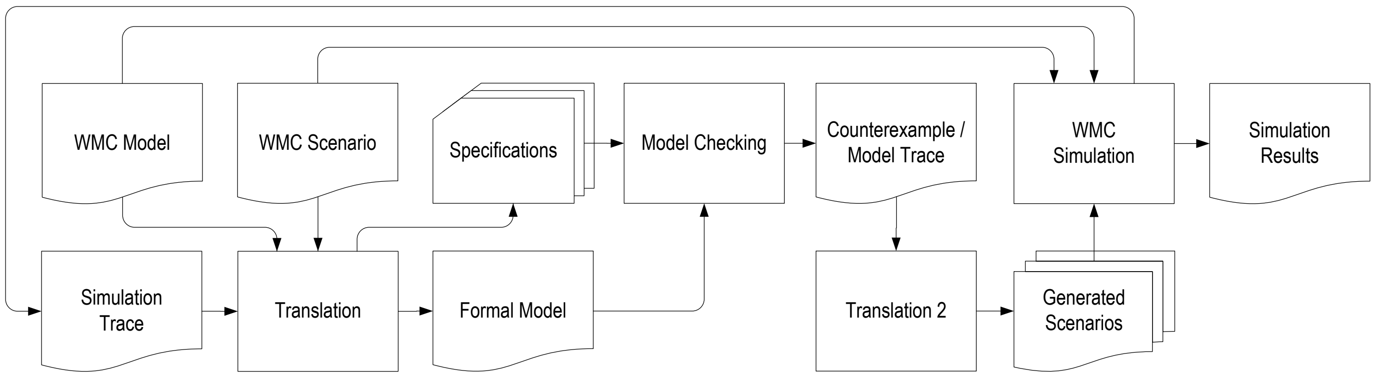

Fig. 1. Method for the synergistic use of WMC simulation and model checking.

exhaustive capabilities of model checking with the scalability of simulation. Specifically, formal verification is used selectively to evaluate bounded elements of a simulated system [9]–[11]. Of particular interest to this project is work that has used simulation traces as a means of creating formal models of a small enough scope that they avoid scalability problems (for example, see [12]–[14]). However, these analyses are limited in that they only check properties about the actual trace. Thus they do not account for any system behavior beyond what is already in the trace. There is therefore a need to more effectively use formal verification and simulation together to use formal verification to explore the space around simulation traces instead of just the traces themselves.

C. Our Method

We are attempting to develop a unique analysis approach that will allow the WMC simulation to be used synergistically with formal verification. Specifically, we want to give analysts the ability use the exhaustive capabilities of model checking to explore the region around a simulated air traffic scenario to find excessive human taskload conditions worthy of deeper, high-fidelity analysis with the simulation. To accomplish this, we are developing a method that uses the Symbolic Analysis Laboratory (SAL) [15] to model check the space around WMC simulation traces.

This method (see Fig. 1) works as follows:

1) A WMC work model (which describes the agents in a simulation along with the actions they perform and the resources they modify) and a scenario (which describes the initial conditions that represent a specific air traffic situation and future events that can occur) is run through a WMC simulation. The simulation produces a trace showing exactly how that scenario played out.
2) The work model, scenario, and simulation trace are then automatically translated into a formal model representing the simulation over a constrained period of time. In this model, the timing of actions (when they occur and how long they occur) can include variance to allow the model checker to explore the performance space around the modeled scenario. The translator also generates a set of specification properties designed to find interesting taskload conditions in the model.
3) A model checker is then used explore the formal model to generate traces illustrating violations of specifications.
4) The traces are then translated back into WMC scenarios for deeper analyses in its simulation environment.

D. Objectives

To be able to formally model WMC concepts with our method (Fig. 1), we needed a formally modeling architecture. The architecture needed to support all of the following: (a) *Modeling real time:* Because our method allows analysts to evaluate how variance in timing affects taskload, we need to be able to formally model real time; (b) *Modeling taskload:* WMC can support a priority-queue-based approach to modeling human taskload and control how human's switch between tasks and actions [16]. Thus, our architecture needs to be able to replicate the taskload and task switching behavior of WMC; (c) *Computational efficiency:* Because of the scalability limitations of model checking, the architecture must represent WMC concepts in a computationally efficient manner. In this paper, we describe a formal modeling approach that satisfies these requirements. To do this, we first discuss the relevant WMC concepts the architecture needs to encapsulate. We then discuss then discuss our architecture and how it has been realized. We also describe the specification properties that we can use to generate traces with this model. Finally, we discuss how we plan to use our architecture in future evaluations of air traffic scenarios.

II. WMC

Work Models that Compute (WMC) is a simulation framework that dynamically models complex, multi-agent concepts of operations and work domains [17]. WMC attempts to model the collective work of a set of agents [18]. It consists of two parts: models which describe the work of a given domain, i.e. a work model, and an engine to simulate any work model [17]. Each work model has three primary elements: agents, actions and resources. Resources are defined as a collection of specific elements of the work environment which can be sensed and manipulated by the agents. Actions manipulate resources and are linked to a specific agent and represent the work at its most atomic unit. The work model specifies the actions frequency, priority and duration of resources it needs or manipulates and which agents are involved [19]. Agents serve the dual purpose of organizing actions and adding a layer of dynamics to the prescribed action sequence by placing limits on the number of actions performed simultaneously and their priority [18].

A scenario/script pulls together all these elements of work models, agents, actions, and resources into a scenario simulation which can then generate an action trace as well as other higher-level metrics of interest. The simulation engine works on a hybrid timing mechanism that allows WMC to incorporate features of both continuous time and event-based simulation. This enables WMC to simulate both dynamical systems (such as aircraft dynamics) and event based agents (such as pilot models) [19], [20].

In the case of human agents, WMC has the capability to model human taskload [16]. Specifically, each modeled human agent has two priority queues: one representing agents that are *active* (currently being executed) and ones that are inactive. Inactive actions can have two classifications. Those that have never been been executed are designated as *waiting*. Actions that were previously active are *delayed*. The active queue has a limited capacity which results in actions transitioning between queues. If a human agent is assigned new actions, those action are put in the unassigned queue and given the *waiting* designation. If the there is room in the active queue, the highest priority actions (those with the highest explicit priority with the shortest execution time as determined by the action's resources) are moved to the active queue. If there are active actions with lower priorities than those in the inactive queue, those actions are set to *delayed* and moved to the inactive queue while the higher priority actions are made *active*. As actions are finished, they are removed from the active queue. Thus, with this infrastructure, taskload is represented be the number of actions in an agents queues.

III. Formal Modeling Architecture

To formally model these WMC concepts, we have created an abstract architecture (Fig. 2). In this, the formal model is represented by three synchronously composed modules: a scheduler, actions, and agents. A Scheduler module keeps track of modeled time, determines when actions are assigned to agents, and coordinates the behavior of the other modules based on the scheduler's status. The actions module is actually a collection of synchronously composed action modules that represent the actions from the WMC simulation. Similarly, the agents module is a collection of synchronously composed agent modules that represent the agents from the WMC simulation.

The actions and agents modules communicate with each other via arrays of action and agent data types (actions and agents from Fig. 2), where each action and agent module is associated with an instance of an action and agent datatype respectively. The action data type contains all of the following:

id: A unique action identification as an integer from 1 to *N*, where there are *N* total actions;
agent: The identification of the agent responsible for the action;
state: The priority queue location and status of the action: whether it is in *active*, *waiting*, or *delayed*, or *notAssigned*;
priority: The priority level of the action as a bounded integer;

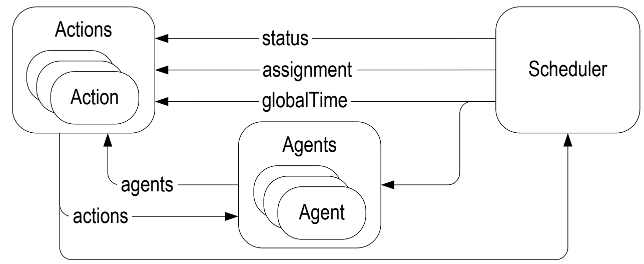

Fig. 2. Formal modeling architecture used to represent WMC concepts.

time: The time left for the action to finish executing (initial times are determined by the time it takes to set an action's resources in the original WMC model); and
update: The next time the action will be assigned (this is determined by the observed timings in the simulation trace);

The agent data type contains the following:

id: A unique agent identification as an integer from 1 to *M*, where there are *M* total agents;
activeCapacity: The agent's active priority queue capacity;
activeCount: The number of active actions the agent is responsible for (the number of actions in the agent's active priority queue);
minActive: The action data type for an active action the agent is responsible for that has the minimum priority (smallest *priority* and longest *time*) of all such actions; and
maxInactive: The action data type for a *waiting* or *delayed* action (an action in the *inactive* queue) the agent is responsible for that has the maximum priority (greatest *priority* and shortest *time*) of all such actions.

Our architecture does not explicitly represent the priority queues underlying WMC agents. Rather, the state of these can be inferred by reasoning over the array of action data types. To do this efficiently, we make use of λ calculus operations so that we can reason about sets [21] of actions. In this sense, a set is a mapping of action *id*s to Boolean values $actionset : actionID \rightarrow Boolean$. Such sets use λ operations to define this mapping. For example, the empty set $\varnothing = \lambda(i \in actionIDs) : \text{False}$. This can be interpreted as: for all possible values of action id *i*, *i* is not in the set (*i* maps to False).

Our architecture also abstracts away the WMC concept of resources in service of computational efficiency. However, what resources are being modified at any give time can still be inferred from what actions are executing at that time.

The following describes the details of each of the elements in our architecture.

A. Scheduler

The scheduler module is responsible for maintaining the clock and communicating the *globalTime* to the other modules. It also indicates when *notAssigned* actions who have *update* times at the current *globalTime* are ready to be executed via the *assignment* variable. Finally, the scheduler uses its *status* to coordinate the behavior of the other modules.

The scheduler's *status* transitions between its three states using the logic in Fig. 3. Specifically, it starts out *assigning*,

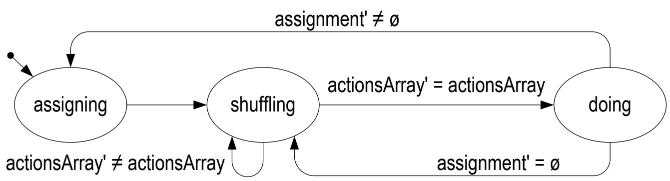

Fig. 3. State transition system representing the scheduler's *status*. Note that an $'$ on a variable indicates that variable's value in the next state. For example $actions' \neq actions$ is checking whether *actions* will change in the next state.

where it indicates which actions are ready to be executed. After *assigning*, it automatically transitions to *shuffling*. When the scheduler is *shuffling*, the action modules are able to reassign the execution state of assigned actions (move them between priority queues). While *shuffling*, the scheduler monitors the state of *actions* to see if any changes will occur in the next state. If any changes do occur, the scheduler remains *shuffling*. If there are no changes, the scheduler status transitions to *doing*. If the *status* is *doing* then, if in the next state there is nothing to assign, the *status* transitions to *shuffling*. Otherwise it transitions to *assigning*.

If the scheduler is *assigning* it communicates which actions are ready to be performed by computing a λ calculus set. This set is defined as

$$\begin{aligned} assignment = \lambda(i \in actionIDs) : & \\ & actions[i].update = globalTime \\ & \wedge actions[i].state = notassigned. \end{aligned} \tag{1}$$

This can be interpreted as the set of all action ids such that the associated actions are currently *notassigned* and have an *update* time equal to the current *globalTime*.

The scheduler uses timed automata [22], [23] to represent *globalTime* as a real valued quantity. If the the scheduler status is *doing* then *globalTime* is increased to the minimum value representing the time it takes to finish any of the *active* actions or the next *update* time of any of the *notAssigned* actions.

B. Agents

The agents module is a composition of synchronously composed agents, where each agent module manages the values of its corresponding agent data type. Conceptually, the agent module is responsible for keeping track of the number of its active actions (*activeCount*). It also provides information each action will need for moving between execution states (moving between priority queues) in the form of its *minActive* and *maxInactive* variables.

To compute *activeCount* an agent module will use the formula shown in (2) with $actionState = active$. Note that in the formal model, the code for doing this operation is automatically generated with a known bound on the number of possible actions (N). This operation is therefore linear and scales efficiently. It is also important to note that this same equation (2) can be used to compute the number of actions that are *waiting* and *delayed*.

To compute *minActive* and *maxInactive*, the agent first uses λ calculus to compute sets containing all of the action ids that satisfy the minimum active and maximum inactive criteria (*minActiveSet* and *maxInactiveSet* respectively). *minActiveSet* is computed as shown in (3) where, for all action ids i in a set of *actionIDs*, i is in the set if the action with $id = i$ is *active*, associated with the given agent, and has a priority less than or equal to all other *active* actions of the agent. *maxInactiveSet* is computed as shown in (4) where, for all action ids i in a set of *actionIDs*, i is in the set if the action with $id = i$ is *delayed* or *waiting*, associated with the given agent, and has a priority greater than or equal to all other *waiting* or *delayed* actions associated with the agent. With these sets computed, the actions *minActive* and *maxInactive* are selected non-deterministically from the action *id*s in *minActiveSet* and *maxInactiveSet* respectively. This allows for non-determinism in what actions will ultimately be active or inactive at any given time if the actions have the same priority.

C. Actions

Each action module in Actions is responsible for managing the values in the associated action data type based on the schedulers status and the global time. For any given state in the model, the action module behaves as follows:

- If *status* is *doing* and the action's *state* is *active* then the action's *time* is decremented based on the amount of elapsed time since the clock was last updated. If doing this means that the action has finished (that *time* becomes 0), the action's *state* is set to *notAssigned* and its *update* time is set to the action's next update time from the original simulation trace. To add non-determinism to the timing of actions, variance can be included in the update time.
- If *status* is *assigning* and the action is in the set of assigned actions then the action's *state* is set to *waiting* and the action's *time* is set. Non-deterministic amounts of variance can be added to the time in this assignment.
- If *status* is *shuffling* then:
  - If the action's state is *waiting* or *delayed* and it is equal to its agent's *maxInactive* action and either the action has a higher priority than its agent's *minActive* action or its agent's active capacity has not been reached, then the action's state is set to *active*.
  - If the action's state is *active* and it is equal to its agent's *minActive* action and the agent's active capacity has been exceeded then the action's state is set to *delayed*.

D. Analysis Capabilities

This architecture gives us the ability to model sections of simulation traces with included variance in the timing of actions. This is useful because it gives us the ability to reason about taskload in specification properties, where we can assert the absences of potentially problematic conditions for human agents. We can then use these with a model checker to generate counterexamples that will allow us to use our method (Fig. 1) to create WMC scenarios to examine the counterexample conditions in the simulation. For our current purposes, we are interested in specifications related to excessive workload and conditions where people may drop or forget actions.

For excessive workload, we are concerned with finding conditions where the human operator's active priority queue

$$
\begin{aligned}
\mathit{cardinality}(\mathit{actionState}) = &\begin{cases} 1, & \text{if } \mathit{actions}[1].\mathit{agent} = \mathit{agentID} \wedge \mathit{actions}[1].\mathit{state} = \mathit{actionState} \\ 0, & \text{otherwise} \end{cases} \\
&+ \ldots + \begin{cases} 1, & \text{if } \mathit{actions}[N].\mathit{agent} = \mathit{agentID} \wedge \mathit{actions}[N].\mathit{state} = \mathit{actionState} \\ 0, & \text{otherwise} \end{cases}
\end{aligned} \tag{2}
$$

$$
\begin{aligned}
\mathit{minActiveSet} = &\lambda(i \in \mathit{actionIDs}) : \mathit{actions}[i].\mathit{agent} = \mathit{agentID} \wedge \mathit{actions}[i].\mathit{state} = \mathit{active} \\
&\wedge \forall (j \in \mathit{actionIDs}) : \\
&\left(\begin{pmatrix} \mathit{actions}[j].\mathit{agent} = \mathit{agentID} \\ \wedge \mathit{actions}[j].\mathit{state} = \mathit{active} \end{pmatrix} \Rightarrow \begin{pmatrix} \mathit{actions}[i].\mathit{priority} < \mathit{actions}[j].\mathit{priority} \\ \vee \begin{pmatrix} \mathit{actions}[i].\mathit{priority} = \mathit{actions}[j].\mathit{priority} \\ \wedge \mathit{actions}[i].\mathit{time} > \mathit{actions}[j].\mathit{time} \end{pmatrix} \end{pmatrix} \right)
\end{aligned} \tag{3}
$$

$$
\begin{aligned}
\mathit{maxInactiveSet} = &\lambda(i \in \mathit{actionIDs}) : \mathit{actions}[i].\mathit{agent} = \mathit{agentID} \wedge (\mathit{actions}[i].\mathit{state} = \mathit{waiting} \vee \mathit{actions}[i].\mathit{state} = \mathit{delayed}) \\
&\wedge \forall (j \in \mathit{actionIDs}) : \\
&\left(\begin{pmatrix} \mathit{actions}[j].\mathit{agent} = \mathit{agentID} \\ \wedge \begin{pmatrix} \mathit{actions}[j].\mathit{state} = \mathit{waiting} \\ \vee \mathit{actions}[j].\mathit{state} = \mathit{delayed} \end{pmatrix} \end{pmatrix} \Rightarrow \begin{pmatrix} \mathit{actions}[i].\mathit{priority} < \mathit{actions}[j].\mathit{priority} \\ \vee \begin{pmatrix} \mathit{actions}[i].\mathit{priority} = \mathit{actions}[j].\mathit{priority} \\ \wedge \mathit{actions}[i].\mathit{time} < \mathit{actions}[j].\mathit{time} \end{pmatrix} \end{pmatrix} \right)
\end{aligned} \tag{4}
$$

is at capacity [16]. We can use linear temporal logic to assert that the active queue for a given human agent with $id = i$ will never reach capacity with

$$
\mathbf{G} \neg \begin{pmatrix} (\mathit{status} = \mathit{doing}) \\ \Rightarrow \begin{pmatrix} \mathit{agent}[i].\mathit{activeCapacity} \\ \neq \mathit{agent}[i].\mathit{activeCount} \end{pmatrix} \end{pmatrix}, \tag{5}
$$

where this can be interpreted as: for all paths through the model ($\mathbf{G}$) we never want it to be true that if the scheduler *status* is *doing* then agent *i*'s *activeCount* reaches or exceeds its *activeCapacity*. Note that we are only concerned with the capacity of an agent's queues when the scheduler's *status* is *doing* because, by design, queue capacities may be exceeded during nominal *assigning* and *shuffling* operations.

By adding a synchronous observer (see [24]), we can also have a model variable (*count*) that can count the number of clock update periods over which a full active queue is maintained. This can allow us to specify that an active queue should never remain full for over K updates of the clock as

$$
\mathbf{G} \neg (\mathit{count} \geq K). \tag{6}
$$

Thus, we can generate traces illustrating how a human agent can remain at maximum active capacity over K periods.

There are several different reasons why a human operator may fail to perform an action. First, if the human's working memory is exceeded (indicated by excessive actions that are *waiting* or *delayed*) he or she might forget an action. We can assert the absence of this condition as follows:

$$
\mathbf{G} \neg \begin{pmatrix} (\mathit{status} = \mathit{doing}) \\ \Rightarrow \left(\begin{pmatrix} \mathit{cardinality}(\mathit{waiting}) \\ + \mathit{cardinality}(\mathit{delayed}) \end{pmatrix} \geq \mathit{max} \right) \end{pmatrix}, \tag{7}
$$

where *max* is the maximum capacity of the agent's inactive queue.

A human may also forget an action is it remains in working memory (*waiting* or *delay*) for too long [25]. We can specify that this should never occur as

$$
\mathbf{G} \neg \begin{pmatrix} (\mathit{actions}[j].\mathit{state} \neq \mathit{notAssigned}) \\ \Rightarrow \left(\begin{pmatrix} \mathit{globalTime} \\ - \mathit{actions}[j].\mathit{update} \end{pmatrix} \geq \mathit{timeMax} \right) \end{pmatrix}, \tag{8}
$$

where *timeMax* is an analyst-specified waiting time that an action should not exceed.

IV. Discussion

The presented architecture allows use to formally represent all of the relevant WMC concepts while satisfying our objectives. (a) It uses timed automata to represent real time and allows for sensitivity analysis of WMC concepts based on variance in the timing of actions. (b) It allows taskload to be modeled by having the model reason over an array of actions, each with its own state and associated agent. (c) By using λ operations over the set of actions and by abstracting away WMC details un-important to the formal analyses, the architecture is computationally efficient. Additionally, a number of specification properties can be used to reason about taskload for use in creating counterexamples for later use in scenario creation. We currently have a working version of this architecture working with the WMC simulation and the infinite bounded model checker in SAL.

This work was only focussed on the architecture required to realize our method (Fig. 1). Future work will focus on completing the method and using it to evaluate realistic air traffic scenarios.

A. Translation Processes

We have already developed a prototype translation process that converts WMC trace, scenario, and work model information into a formal model that uses the architecture presented here. This method is currently being used to evaluate existing WMC models and scenarios. Future work will focus on refining this translation process to reduce the amount of human analyst intervention required. Additionally, in current efforts, all reverse translation (Translation 2 from Fig. 1) must be done manually. Future efforts will focus on automating this. As this project progresses, these translations will benefit from the use of a standardized WMC XML models.

B. XML Models

As of now, the current system evaluates SAL models generated from WMC action traces. As we move forward, we hope to create a system which has less reliance on action traces and offers a flexible mechanism for developing scenarios and simulations by means of a XML Specification for the work model and scenario. The XML Specification would create a universal starting point for modeling new systems of interest and allow rapid changes to existing work models. Having such an abstracted specification would support collaborators or general users who are unfamiliar with WMC or SAL but more familiar with XML markup. Additionally, an XML Specification would allow for XML files of work models and scenarios to be translated back and forth between WMC simulations and SAL experiments more easily. The XML Parser currently translates XML into WMC, and other directions are being developed now.

C. Realistic Application

We will use the completed method (Fig. 1) to evaluate realistic air traffic scenarios. In particular, we will examine the different scenarios from [26] which represented three aircraft arriving into Schiphol Airport RWY18R under different distributions of authority, autonomy, and responsibility between air and ground-based operators. In particular, the scenarios explored in [26] did not take the limitations of human operator taskload into account. Thus, our future efforts will use the formal architecture here and the associated formal verification from the method to generate scenarios that will allow the simulation to explore how performance changes based on different operator memory limitations (priority queue capacity restrictions) and different amounts of variance between the timing of actions.

V. Acknowledgement

This work was supported by the grant "Scenario-Based Verification and Validation of Autonomy and Authority" from the NASA Ames Research Center as sub-award number 67102 from the Georgia Institute of Technology.

References

[1] S. G. Hart and L. E. Staveland, "Development of NASA-TLX (Task Load Index): Results of empirical and theoretical research," *Advances in psychology*, vol. 52, pp. 139–183, 1988.

[2] A. R. Pritchett, S. Y. Kim, and K. M. Feigh, "Measuring human-automation function allocation," *Journal of Cognitive Engineering and Decision Making*, vol. 8, no. 1, pp. 52–77, 2014.

[3] J. M. Wing, "A specifier's introduction to formal methods," *Computer*, vol. 23, no. 9, pp. 8, 10–22, 24, 1990.

[4] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model checking*. Cambridge: MIT Press, 1999.

[5] E. A. Emerson, "Temporal and modal logic," in *Handbook of Theoretical Computer Science*, J. van Leeuwen, A. R. Meyer, M. Nivat, M. Paterson, and D. Perrin, Eds. Cambridge: MIT Press, 1990, ch. 16, pp. 995–1072.

[6] M. L. Bolton, E. J. Bass, and R. I. Siminiceanu, "Using formal verification to evaluate human-automation interaction in safety critical systems, a review." *IEEE Transactions on Systems, Man and Cybernetics: Systems*, vol. 43, no. 3, pp. 488–503, 2013.

[7] J. Moore, R. Ivie, T. Gledhill, E. Mercer, and M. Goodrich, "Modeling human workload in unmanned aerial systems," in *2014 AAAI Spring Symposium Series*. Palo Alto: AAAI, 2014, pp. 44–49.

[8] R. Stocker, N. Rungta, E. Mercer, F. Raimondi, J. Holbrook, C. Cardoza, and M. Goodrich, "An approach to quantify workload in a system of agents," in *Proceedings of the 14th International Conference on Autonomous Agents and Multiagent Systems*. Liverpool: IFAAMAS, 2015.

[9] A. J. Hu, "Simulation vs. formal: Absorb what is useful; reject what is useless," in *Proceedings of the Third International Haifa Verification Conference*. Berlin: Springer, 2008, pp. 1–7.

[10] J. Yuan, J. Shen, J. Abraham, and A. Aziz, "On combining formal and informal verification," in *Computer Aided Verification*. Springer, 1997, pp. 376–387.

[11] G. Gelman, K. M. Feigh, and J. Rushby, "Example of a complementary use of model checking and agent-based simulation." Piscataway: IEEE, 2013.

[12] A. Yasmeen, K. M. Feigh, G. Gelman, and E. L. Gunter, "Formal analysis of safety-critical system simulations," in *Proceedings of the 2nd International Conference on Application and Theory of Automation in Command and Control Systems*. IRIT Press, 2012, pp. 71–81.

[13] D. A. Stuart, M. Brockmeyer, A. K. Mok, and F. Jahanian, "Simulation-verification: Biting at the state explosion problem," *IEEE Transactions on Software Engineering*, vol. 27, no. 7, pp. 599–617, 2001.

[14] X. Chen, H. Hsieh, F. Balarin, and Y. Watanabe, "Automatic trace analysis for logic of constraints," in *Proceedings of the Design Automation Conference*. IEEE, 2003, pp. 460–465.

[15] L. de Moura, S. Owre, and N. Shankar, "The SAL language manual," Computer Science Laboratory, SRI International, Menlo Park, Tech. Rep. CSL-01-01, 2003.

[16] A. Pritchett and K. Feigh, "Simulating first-principles models of situated human performance," in *Proceedings of the IEEE First International Multi-Disciplinary Conference on Cognitive Methods in Situation Awareness and Decision Support*. Piscataway: IEEE, 2011, pp. 144–151.

[17] A. R. Pritchett, "Simulation to assess safety in complex work environments," J. D. Lee and A. Kirlik, Eds. New York: Oxford University Press, 2013, ch. 22, pp. 352–366.

[18] A. R. Pritchett, K. M. Feigh, S. Y. Kim, and S. K. Kannan, "Work models that compute to describe multiagent concepts of operation: Part 1," *Journal of Aerospace Information Systems*, vol. 11, no. 10, pp. 610–622, 2014.

[19] G. E. Gelman, "Comparison of model checking and simulation to examine aircraft system behavior," Ph.D. dissertation, Georgia Institute of Technology, 2012.

[20] A. R. Pritchett, S. Y. Kim, and K. M. Feigh, "Modeling human–automation function allocation," *Journal of Cognitive Engineering and Decision Making*, vol. 8, no. 1, pp. 33–51, 2014.

[21] G. Smith and L. Wildman, "Model checking z specifications using sal," in *ZB 2005: Formal Specification and Development in Z and B*. Springer, 2005, pp. 85–103.

[22] R. Alur and D. L. Dill, "A theory of timed automata," *Theoretical computer science*, vol. 126, no. 2, pp. 183–235, 1994.

[23] B. Dutertre and M. Sorea, "Timed systems in SAL," SRI International, Tech. Rep. NASA/CR-2002-211858, 2004.

[24] J. Rushby, "The versatile synchronous observer," in *Specification, Algebra, and Software*, S. Iida, J. Meseguer, and K. Ogata, Eds. Springer, 2014, pp. 110–128.

[25] D. C. McFarlane and K. A. Latorella, "The scope and importance of human interruption in human-computer interaction design," *Human-Computer Interaction*, vol. 17, no. 1, pp. 1–61, 2002.

[26] M. IJtsma, A. R. Pritchett, and R. P. Bhattacharyya, "Computational simulation of authority-responsibility mismatches in air-ground function allocation," p. 6 pages, 2015.