Houser, A., Ma, L. M., Feigh, K., & **Bolton, M. L.** (ND). Using formal methods to reason about taskload and resource conflicts in simulated air traffic scenarios. *Innovations in Systems and Software Engineering*. DOI: 10.1007/s11334-017-0305-2, 15 pages. In press.

Innovations Syst Softw Eng manuscript No.
(will be inserted by the editor)

Using Formal Methods to Reason About Taskload and Resource Conflicts in Simulated Air Traffic Scenarios*

Adam Houser, Lanssie Mingyue Ma, Karen M. Feigh, and Matthew L. Bolton,

Received: XXX / Accepted: XXX

Abstract In complex environments, like the modern air traffic system, interactions between human operators and other system agents can profoundly impact system performance. System complexity can make it difficult to determine all of the situations where issues can arise. Simulation and formal verification have been used separately to explore the role of humans in complex systems. However, both have problems that limit their usefulness. In this paper, we describe a method that allows interesting conditions related to human taskload and resource conflicts between agents to be discovered and evaluated in high fidelity through the synergistic use of formal verification and simulation. The core of this method is based on a formal modeling architecture that represents original, agent-based 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 using a priority queue model where only a limited number of actions can be performed or remembered by a human at a given time. Resources affected by agent behaviors are associated with actions so that resources can be reasoned about at the action level. We discuss our method and its formal architecture. We describe how the method can be used to find taskload and resource conflict conditions through the use of formal, checkable specification properties. We then use a simple air traffic example to demonstrate the ability of our method to find interesting taskload and resource conflict conditions around a simulation trace. The implications of this method are discussed and directions for future work are explored.

1 Introduction

A number of problems are associated with human-automation interaction in complex systems [2, 17, 53]. High demands on human resources can lead to excessive taskload; work that fails to engage the human (such as monitoring) can lead to a loss of vigilance; and poorly designed interfaces may be incompatible or contradictory to the human operator's task. Further, the complexity of a system can make it difficult for humans to track system state. This can lead to situations where the system does things the human does not expect or human actions result in unintended outcomes, both of which can lead to large-scale system disasters [49]. All of these conditions can result in human error and system failures. As systems become more autonomous, these types of problems are expected to become worse [17]. In this work, we are predominantly concerned with issues related to taskload and concurrency issues that can lead to human operator confusion.

Human operator taskload is a measure of the number of tasks a human operator is expected to perform at a given time [43, 44]. Taskload has shown itself to be a good indi-

*We presented an earlier version of this manuscript at the 2015 International Conference on Complex Systems Engineering [30]. This new manuscript makes substantial contributions beyond what was reported in [30]. Specifically, we present additional specification properties for generating different types of simulation scenarios. We also report results showing how the model checking analyses connects with the agent-based simulation. This new paper also features an extended discussion.

A. Houser and M. L. Bolton
Department of Industrial and Systems Engineering
State University of New York at Buffalo
Amherst, NY 14260
Tel.: (716) 645-2359
Fax: (716) 645-3302

M. L. Bolton E-mail: mbolton@buffalo.edu

L. M. Ma and K. Feigh
Daniel Guggenheim School of Aerospace Engineering,
Georgia Institute of Technology
Atlanta, GA 30332

cator of human operator mental workload (the total mental effort being expanded by a person at a given time) in the air traffic system [25, 29, 43, 44, 60]. In this sense, it is important that participants in this system avoid exorbitant levels of taskload to prevent 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. However, due to the many different operating conditions that can occur, experiments and tests with real systems and human subjects are infeasible.

In complex systems that contain humans, physical elements, and computational elements (agents) in a distributed environment, confusion about the value of shared resources can have profound implications for system safety. Specifically, if the human cannot keep track of the system state, he or she may behave erroneously or be surprised by system behavior and behave unpredictably [41, 52]. These types of concurrency problems can be thought of in terms of resources. Specifically, in systems with concurrency, agents can often have shared resources (variables and/or data) that they can access and/or modify. If a resource is modified concurrently by two agents, or modified by one agent while others are accessing it, a human agent may become confused and behave erroneously because a resource value does not match his or her expectations.

To address these problems, researchers have been building simulation environments such as Work Models that Compute (WMC) [47] that allow human operator taskload and system performance to be analyzed in a variety of air traffic simulations. Such simulations are more flexible than human subject experiments and real world tests. They are also able to represent complex system concepts in high fidelity. However, they follow a traditional experimental approach and are thus not exhaustive and can still miss potentially dangerous or performance-critical operating conditions that did not happen to occur in one of the explored scenarios. Researchers have also been exploring how formal verification [63] such as model checking [16] (a means of performing formal verification based on exhaustive/complete searches of systems models) can be used to prove whether or not concurrency issues can produce confusing or dangerous conditions in systems that rely on human-automation interaction [7, 61]. By virtue of using proofs, these analyses are exhaustive and will thus not miss conditions that can manifest in the evaluated models. However, this completeness comes at the expense of scalability, which can severely limit what systems can be analyzed.

In this work, we strove to develop a new method for using simulation and formal verification synergistically in a way that exploits the advantages of both approaches. Specifically, this approach gives analysts the ability to use model checking's exhaustive search capabilities to explore the region around a simulated air traffic scenario to find excessive human taskload conditions and resource conflicts that may have previously been undiscovered. Discovered conditions can then be more deeply explored using the high-fidelity simulation analysis. By exploiting each method's strengths, this work makes a significant contribution in that it allows formal analyses to be used with systems for which they were not previously possible. In this paper, we describe how this method was realized. First, we discuss the background necessary to understand our method. We then discuss the formal modeling architecture we developed for encapsulating WMC concepts. We describe the specification properties that we can use with the model to generate traces. Finally, we demonstrate the capabilities of our method by showing how verification results can be used to create simulation scenarios for an air traffic application.

2 Background

2.1 WMC

Work Models that Compute (WMC) is a simulation framework that dynamically models complex, multi-agent concepts of operations and work domains [45]. WMC attempts to model the collective work of a set of agents [46]. It consists of two parts: a work model that describes the work of a given domain, and an engine that simulates the work model [45]. Each work model comprises 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, are linked to a specific agent, and represent the work at its most atomic unit. The work model specifies each action's frequency, priority, duration of resources it needs or manipulates, and which agents are involved [28]. Agents serve the dual purpose of organizing actions and adding a layer of dynamics to the prescribed action sequence by placing limits on both the number of simultaneously performed actions and their priorities [46].

A scenario pulls all elements of work models, agents, actions, and resources into a simulation. This can be used to generate an action trace and 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 dynamic systems (such as aircraft dynamics) and event-based agents (such as pilot models) [28, 48].

WMC provides an excellent framework for the work discussed in this paper. It allows for the modeling of humans and automated agents, while accounting for the resources each requires and modifies over the course of their work. It also accounts for human operator taskload. Further, while

traditional simulations represent actions as if they are the sole responsibility of specific agents, WMC assigns actions to the work model. Actions are then executed by whichever agents are available. This allows for a full investigation of work as a fully defined object.

2.2 Modeling Taskload in Aerospace Systems

The current version of WMC does not support the ability to evaluate human operator taskload. However, earlier simulations have researched this capability using the concepts around which WMC is built [43, 44]. In this formulation, each modeled human agent has two priority queues: one representing actions that are *active* (currently being executed) and one representing actions that are *inactive*. Inactive actions can have two designations. Those that have never been executed are designated as *delayed*, and actions that were previously active but are now inactive are designated as *interrupted*. The active queue has a limited capacity which results in actions transitioning between queues. If a human agent is assigned new actions, those actions are put in the inactive queue and given the *delayed* designation. If 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 lower-priority actions are moved to the *inactive* queue and designated as *delayed*, and the higher priority actions are moved up into the *active* queue. As actions are finished, they are removed from the *active* queue and rescheduled for later (if they occur again) or to never occur again. Within this infrastructure, taskload can be described as the complete set of actions in all of an agent's queues. While this approach to taskload has its limitations [35], it has proven itself useful in a number of aerospace applications [25, 29, 43, 44, 60].

2.3 Formal Verification

Formal verification is an analysis technique that falls within the discipline of formal methods. Formal methods are well-defined mathematical languages and techniques for the specification, modeling, and verification of systems [63]. Specification properties mathematically describe desirable system conditions. Systems are modeled using mathematically-based languages, and verification then mathematically proves whether or not the model satisfies the specification. Model checking is an automated approach to formal verification [16], whereby 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 [24]. 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. This shows how the specification violation occurred as a trace through the statespace of the model.

Formal verification has been successfully used to evaluate human-automation interaction in a number of different capacities [7, 61]. Most relevant to this discussion is the research that has focused on workload and mode confusion.

There is a significant body of literature that focuses on using formal methods to find potential mode confusion in human-automation interaction [7]. However, this work has predominantly used models of human-machine interfaces [12–14, 34, 36], concurrently executing human mental models and automation models [11, 20, 21, 50, 51, 54], or concurrently executing human task knowledge and automation models [8–10, 62]. As such, these analyses have almost exclusively focused on systems with single human operators and have not accounted for the interaction of multiple humans and automated agents in a distributed system.

Comparatively fewer papers have explored how human workload and taskload can be considered in formal methods. The sole contributions have come from Mercer and Goodrich et al. [40, 56], who have researched ways of formally modeling workload. However, they have not used these methods in formal verification analyses.

No matter the analysis subject, formal verification techniques like model checking suffer from combinatorial explosion, where the statespace grows exponentially as additional components are added to the model [16]. This can lead to models that are too big or take too long to be verified. Because of this scalability limitation, complex system behaviors must often be represented abstractly. This can limit the types of system behaviors and interactions that can be considered in a given analysis.

2.4 Formal Verification and Simulation in Combination

Success has been found in using formal verification synergistically with simulation to exploit the exhaustive capabilities of model checking with simulation's ability to represent system in higher fidelity. Specifically, formal verification has been used selectively to evaluate bounded elements of a simulated system [26, 27, 31, 65]. Of particular interest to this project is work that has used simulation traces as a means of creating formal models of a scope small enough to avoid limits imposed by scalability [15, 57, 64]. However, these analyses are limited in that they only check properties about the actual trace and thus do not account for any system behavior beyond what is already contained within it. There is significant potential in using model checking with simulation traces, where the model checking analyses will allow

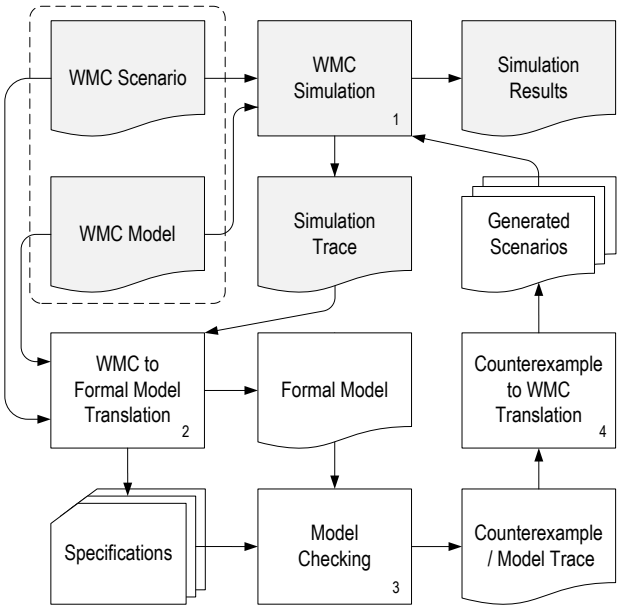

Fig. 1 Flow diagram illustrating the our method for the synergistic use of WMC simulation and model checking. Shaded shapes represent elements that originally existed as part of WMC. Numbers in processes correspond to steps from the list on page 4.

analysts to explore a more complete space around a trace and thus find potentially interesting or dangerous operating conditions that were missed.

3 Our Method

In this work, we attempted to develop a method that gives analysts the ability to use model checking's exhaustive search capabilities to explore the region around a simulated air traffic scenario to find excessive human taskload conditions and/or resource conflicts that may not have been discovered otherwise. We further wanted to give analysts the ability to analyze the discovered scenario condition more deeply using the high-fidelity air traffic simulation.

Our method, which allows WMC simulation to be used synergistically with formal verification, is shown in Fig. 1. This method 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) are run through a WMC simulation. The simulation produces a trace showing exactly how that scenario evolved.
2. The work model, scenario, and simulation trace are then translated into a formal model representing the simulation over a constrained period of time (a time window). In WMC, the time at which actions occur and the duration of those actions are critical to the manifestation of taskload. Thus, in the formal model, the timing of actions (when they occur and for how long they occur) is explicitly represented. This representation can include analyst-defined variance, nominally on the order of one to three seconds, 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 used to 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.

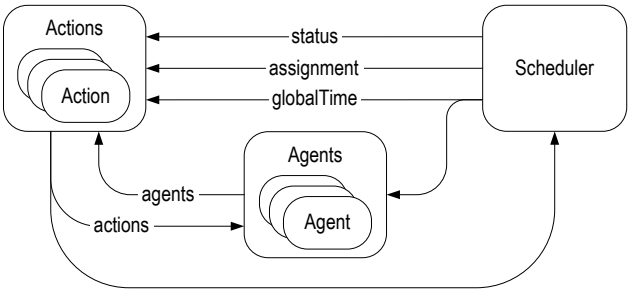

Fig. 2 Formal modeling architecture used to represent WMC concepts.

Note that this method was designed to be employed iteratively. That is, an analyst would gradually move a window of time through a scenario, starting at the beginning, over multiple analyses. This allows the effects of changes in the scenario that happen early its time line to play out and be considered in subsequent model checking sessions.

3.1 Formal Modeling Architecture

To formally model WMC concepts with our method (Fig. 1), we needed a formal modeling architecture. The architecture needed to support all of the following: (a) *Modeling real-valued time:* Because our method allows analysts to evaluate how variance in timing and nondeterminism in action prioritization affects taskload, we needed the capability to formally model real-valued time; (b) *Modeling taskload:* WMC can support a priority-queue-based approach to modeling human taskload and how humans switch between tasks and actions [44]. Thus, our architecture needed 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.

The architecture for accomplishing this is shown in Fig. 2. In this, the formal model is represented by three synchronously composed modules: a Scheduler, Actions, and Agents. The Scheduler keeps track of modeled time, determines when actions are assigned to agents, and coordinates

Table 1 Variables that Define the Action Data Type

Variable	Description
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*, *delayed*, or *interrupted*, or *notAssigned* (the state of an action that has yet to be assigned or has been finished and rescheduled).
priority	The priority level of the action as a bounded integer.
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).
update	The next time the action will be assigned (this is determined by the observed timings in the simulation trace).

the behavior of the other modules based on the scheduler's status. The Actions module is actually a collection of synchronously composed action submodules that represent actions from the WMC simulation. Similarly, Agents is a collection of synchronously composed agent submodules that represent 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). Each action and agent is associated with an instance of its respective data type. The action data type contains all of the variables shown in Table 1, while the agent data type is defined by the variables in Table 2.

Our architecture does not explicitly represent the priority queues underlying WMC agents. Rather, the state of these can be inferred by performing operations over the array of action data types. To do this efficiently, we make use of λ calculus to reason about sets [22, 55] 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 can be represented as

$$\varnothing = \lambda(i \in actionIDs) : \text{False}.$$

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 abstracts away the WMC concept of resources in service of computational efficiency. However, what resources are being modified at any given time can be inferred from the actions that are executing (information that is readily available from the WMC model and scenario).

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

3.1.1 Scheduler

The Scheduler module is responsible for maintaining the clock and communicating the *globalTime* to the other modules. It also indicates when *notAssigned* actions with *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.

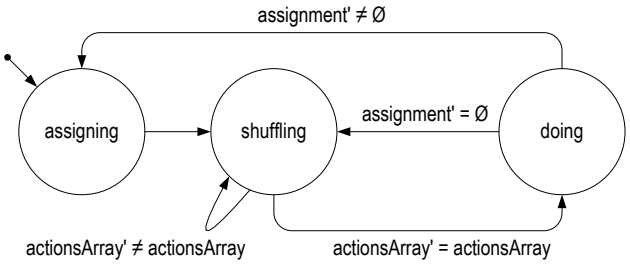

Fig. 3 State transition system representing the scheduler's *status*. Note that an $'$ on a variable indicates that variable's value in the next state. For example $actions' \neq actions$ is checking whether *actions* will change in the next state.

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

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

Table 2 Variables that Define the Agent Data Type

Variable	Description
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 active action the agent is responsible for that has the minimum priority (smallest *priority* and longest *time*) of all such actions.
maxInactive	The *delayed* or *interrupted* 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.

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

The Scheduler uses timed automata [1, 23] to represent *globalTime* as a real-valued quantity. If the Scheduler status is *doing* then *globalTime* is increased to the minimum value in the set of times associated with how long it will take to finish any of the *active* actions and the next *update* times of any of the *notAssigned* actions.

3.1.2 Agents

The Agents module is a composition of synchronously composed agent submodules, where each agent manages the values of its corresponding agent data type. Conceptually, each agent 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 submodule uses the formula in (2) with $actionState = active$. 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 in the *delayed* and *interrupted* priority queues, even though these are not explicitly represented in the formal model.

To compute *minActive* and *maxInactive*, the agent first uses λ calculus to compute sets containing all of the action *id*s that satisfy the minimum active and maximum inactive criteria (*minActiveSet* and *maxInactiveSet*, respectively). The *minActiveSet* is computed as shown in (3) where, for all action *id*s 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 associated with the agent. *maxInactiveSet* is computed as shown in (4) where, for all action *id*s i in a set of *actionIDs*, i is in the set if the action with $id = i$ is *interrupted* or *delayed*, associated with the given agent, and has a priority greater than or equal to all other *delayed* or *interrupted* actions associated with the agent. With these sets computed, the action's *minActive* and *maxInactive* are selected 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.

3.1.3 Actions

Each action within the Actions module is responsible for managing the values in the associated action data type in response to the Scheduler's status and the global time. For any given model state, each action behaves as follows:

- If *status* is *doing* and that action's *state* is *active*, then the action's *time* is decremented based on the amount elapsed 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 *delayed* and the action's *time* is updated. Non-deterministic amounts of time variance can also be added in this assignment.
- If *status* is *shuffling*, then:
  - If the action's state is *delayed* or *interrupted* 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 equal to its agent's *minActive* action, and the agent's active capacity has been exceeded, then the action's state is set to *interrupted*.

3.2 Specification Properties and Analysis Capabilities

Our architecture gives us the ability to model sections of simulation traces with included variance in the timing of actions. This is useful because it allows us to reason about taskload and resource conflicts in specification properties that allow us to assert the absences of specific conditions we are interested in generating traces to explore. Specifically, with model checking, we can then use these properties to generate counterexamples showing exactly how the conditions of interest occur. As such, this enables us to use our method (Fig. 1) to create WMC scenarios to examine the conditions found in the counterexample in the simulation. For our current purposes, we are interested in specifications that concern taskload conditions and potential resource conflicts that can result from concurrency between agents.

There are multiple taskload conditions we are potentially interested in discovering. To find conditions where the human operator is working as hard as they can, we want to see a condition where the operator's active priority queue is at its capacity [44]. To find this, 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 the following specifi-

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

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

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

cation property:[1]

$$FindActiveLoad \models \mathbf{G} \left(\begin{pmatrix} (status = doing) \\ \Rightarrow \begin{pmatrix} agent[i].activeCount \\ < agent[i].activeCapacity \end{pmatrix} \end{pmatrix} \right). \tag{5}$$

This can be interpreted as: for all paths through the model ($\mathbf{G}$), if the scheduler's *status* is *doing*, then agent *i*'s *activeCount* should be less than *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.

If the human's working memory meets or exceeds its capacity, he or she might forget an action. This condition can manifest when there are excessive actions that are *delayed* or *interrupted*. We can use the following specification to find counterexamples where the number of *delayed* or *interrupted* actions meet or exceed capacity:

$$FindInactiveLoad \models \mathbf{G} \left(\begin{pmatrix} (status = doing) \\ \Rightarrow \begin{pmatrix} cardinality(delayed) \\ + cardinality(interrupted) \\ < agent[i].inactiveCapacity \end{pmatrix} \end{pmatrix} \right). \tag{6}$$

[1] Note that specification property patterns are presented with a name, followed by a $\models$, followed by the specification property logic.

In this, $agent[i].inactiveCapacity$ is used as the maximum capacity of $agent[i]$'s *inactive* queue. Note that this parameter is not used to define the behavior of the model (as with the parameters in Table 2). Rather, this value must be defined in the specification itself.

To find conditions where capacity is exceeded, we can check the following:

$$FindOverload \models \mathbf{G} \left(\begin{pmatrix} (status = doing) \\ \Rightarrow \begin{pmatrix} cardinality(delayed) \\ + cardinality(interrupted) \\ \leq agent[i].inactiveCapacity \end{pmatrix} \end{pmatrix} \right). \tag{7}$$

A lack of taskload can also be a problem because it can indicate that the person is not being utilized by the system. Further, humans that are not being engaged by the system may experience a decrement in vigilance which can result in human error [37].

If an original simulation scenario contains excessive taskload, an analyst may want to find variations of the scenario that result in lower taskload. To find counterexamples where a human never reaches their maximum load, we can

use the following specification patterns:

$$FindNoLoad \models \mathbf{G}\neg\left(\begin{array}{l}\left(\begin{array}{l} status = doing \vee status \neq doing \\ \wedge\left(\begin{pmatrix} cardinality(delayed) \\ +cardinality(interrupted)\end{pmatrix} < agent[i].inactiveCapacity\right)\end{array}\right) \\ \mathbf{U}\,(globalTime \geq Never)\end{array}\right). \quad (8)$$

This asserts that globally ($\mathbf{G}$) we never want it to be the case that either the *status* is *doing* or the *status* is not *doing* with the cardinality of the inactive queue less than its capacity until ($\mathbf{U}$) the end of the model simulation time. Similarly, we can find counterexample where a human never exceeds their maximum load with:

$$FindNoOverload \models \mathbf{G}\neg\left(\begin{array}{l}\left(\begin{array}{l} status = doing \vee status \neq doing \\ \wedge\left(\begin{pmatrix} cardinality(delayed) \\ +cardinality(interrupted)\end{pmatrix} \leq agent[i].inactiveCapacity\right)\end{array}\right) \\ \mathbf{U}\,(globalTime \geq Never)\end{array}\right). \quad (9)$$

Variations of *FindNoLoad* [Eq. (8)] and *FindNoOverload* [Eq. (9)] can also be used to find instances of active load that never reach capacity or never exceed it by replacing $agent[i].inactiveCapacity$ with $agent[i].activeCapacity$ and $cardinality(delayed) + cardinality(interrupted)$ with $cardinality(active)$. Thus, an analyst who wishes to find counterexamples with load at or below certain levels can replace any of the capacity variables in Eqs. (8) and (9) with a specific number. This can also be used to find instances of low load that could suggest vigilance decrement and error.

The WMC scenario contains information that relates resources to actions. This can be used to find situations where the concurrent performance of actions could cause resource conflicts. In our work, we are concerned with two different types of resource conflicts. First, in a get-set conflict, if an action sets a resource at the same time that another action gets it, it will not be clear what value the receiving action will assume. This could result in human operator confusion. Similarly, in a set-set conflict, if two actions set resource values at the same time, then it will not be clear what value the resource has and the human(s) involved may become confused. For both types of resource conflicts, we can generate a counterexample demonstrating a conflict between two potentially conflicting actions with

$$FindResourceConflict \models \mathbf{G}\neg\begin{pmatrix} actions[j].state = Active \\ \vee actions[k].state = Active\end{pmatrix}. \quad (10)$$

This asserts that globally ($\mathbf{G}$) we never want a given action ($actions[j]$) to execute (be *Active*) at the same time as another action $actions[k]$.

Finally, a human may forget an action if it remains in working memory (*delayed* or *interrupted*) for too long [38]. We can find where this occurs for a given action with:

$$FindExcessiveDelay \models \mathbf{G}\left(\begin{array}{l}(actions[j].state \neq notAssigned) \\ \Rightarrow\left(\begin{pmatrix} globalTime \\ -actions[j].update\end{pmatrix} < timeMax\right)\end{array}\right). \quad (11)$$

In this, *timeMax* is an analyst-specified waiting time that an action should not exceed. This can be interpreted as for all paths through the model ($\mathbf{G}$), it should always be true that if an action has been assigned to an agent (the action's state is *notAssigned*), then the time since its last update should not exceed *timeMax*.

3.3 Implementation

We have implemented our method (Fig. 1) as a desktop computer application. The program automatically process WMC scenarios, generates the formal model and specification properties, and converts counterexamples result obtained from verifying the properties with a model checker back into a WMC scenario.

In this implementation, a user will first run the WMC simulation to develop a simulation trace. The application then processes the WMC simulation, its scenario, and the simulation trace. The application displays the simulation trace to the analyst who then selects a time frame (a span of time defined by a minimum and maximum) that will be the focus of the model checking analysis. Once the time frame is selected, the application automatically parses the relevant portion of the trace to identify: (a) All of the actions performed during their time frame; (b) The agents associated with the actions; and (c) And all of the resources that are gotten by or set be each of the actions.

The analyst then reviews the list of agents to identify their type (human or basic). If an agent is basic, it represents an automated element of the system. It is thus given an "unlimited" active priority queue (a queue with a maximum equal to the total number of actions in the analysis). If an agent is human, then the analyst can set the size of the agent's active and inactive queues. The analyst can also remove agents from the analysis. This can be a useful feature if automated agent behavior is not a factor in the analysis and the analyst wants to help control for scalability.

When agent editing is completed, the program will modify the list of available actions to remove any actions associated with excluded agents. An analyst can then edit the priorities assigned to actions, the amount of time they take to perform, as well as set variance on action update times and performance times.

When this is completed, the program generates a model checker input file based on the formal modeling architecture (Fig. 2) in the notation of the Symbolic Analysis Laboratory (SAL) [18] for use with its infinite bounded model checker [19].[2] In this file, generated versions of the specifications from section 3.2 are created. Specifications based on *FindActiveLoad*, *FindInactiveLoad*, *FindOverload*, *FindNoLoad*, and *FindNoOverload* [Eqs. (5)–(9)] are generated for each agent included in the analysis. A *FindResourceConflict* property [Eq. (10)] is generated for each pair of actions where a get-set or set-set conflict could occur.

Once the desired specification has been checked with the model checker and a counterexample produced, an analyst can process the counterexample with our application. This analyzes the counterexample and identifies what time each of the actions needs to be performed at in the simulation to achieve the behavior from the counterexample. The application then generates WMC scenario code (C++), which is used to modify the original WMC scenario. This reschedules each of the actions to achieve the new behavior. The WMC simulation can then be rerun to analyze the impact of this behavior. If desired, an analyst can iteratively use our method with a simulation scenario to progressively investigate a scenario over its entire run.

4 Testing

To test our method, we wanted to show first we could use our different classes of specification properties to find the intended simulation conditions. Second, we wanted to show that we could use counterexamples generated with the method to produce the desired behavior in the simulation. For these tests, we use an air traffic control application. Specifically, we employ scenarios from AAR Studies (Authority, Autonomy, and Responsibility Studies; a series of simulations) [32, 33]. These scenarios simulate three aircraft attempting to land on runway RWY18R at the Amsterdam Schiphol Airport under the supervision of an air traffic controller with different distributions of authority and responsibility between the aircraft, the aircraft pilots, and the air traffic controller. For the purpose of the test reported here, we examined a balanced distribution (termed FA3-FA3), which defined a split of responsibility between both parties. This maintains the idea that while both ground and aircraft have their own responsibilities, they maintain full authority for their responsibilities. It also ensures that there is no mismatch between who is completing the action and who is responsible for the success of the action. This ensures our tests are only concerned with actions associated with the landing tasks as opposed to actions associated with the monitoring of other agents' behaviors.

When this simulation scenario was originally run through WMC, it generated a trace with 199,204 simulation actions that occurred over 676.33 seconds of simulated time. Our testing was concerned with determining that our new method was capable of analyzing and modifying WMC traces, and not necessarily analyzing an entire WMC simulation. Thus, the tests only focused on analyzing the first five seconds of the original simulation scenario, checking properties against a formal model of that period, and determining that the modified scenario replicated the behavior found in the counterexamples generated from the checking of the properties. Note that five seconds was chosen for this analysis because, when the original trace was examined, actions tended to cluster together. The first five seconds of the simulation encapsulated the first cluster of actions. This involved the air traffic controller managing all three aircraft's progress to waypoints and clearing the aircraft for descent while the pilots direct their aircraft to the waypoint.

Although the time span considered in our test was small, the model had significant complexity from the other elements of the system. Specifically, our analyses included all four human agents from the original scenario: the pilots from aircraft 1, 2, and 3 as well as the air traffic controller. Further, in this analyzed segment, there were nine human actions: six for the air traffic controller, and one for each pilot. The air traffic controller had separate actions for managing the progress of each aircraft to the waypoint and for clearing each aircraft for descent. Each aircraft pilot was capable of directing his or her aircraft to the waypoint.

Because the original scenarios did not contain priority queue limit information, these had to be assigned for the formal model. For this, each agent was given an active queue with a capacity of two and an inactive queue with a capacity of three (giving each agent a memory capacity of seven plus or minus two [39]). The duration of actions were treated as if they would take one second each to perform.[3] No timing variance was employed in our tests. However, the nondeterminism that could result from similarities in priorities between actions was accounted for.

The SAL file created from the scenario contained generated properties for *FindActiveLoad*, *FindInactiveLoad*, *FindOverload*, *FindNoLoad*, and *FindNoOverload* [Eqs. (5)–(9)] for each agent. Three *FindResourceConflict* specifications using Eq. (10) were generated (all based on potential get-set conflicts): one for each aircraft, where the conflict occurs when the air traffic controller's actions for managing

[2] SAL was chosen for this work because of the expressiveness of input language [18]. This included the ability to model real time, its support of both synchronous and asynchronous composition of modules (though only synchronous composition was ultimately used), and its inclusion of lambda operations.

[3] Note that these timings are not necessarily realistic. The presented tests were concerned with demonstrating the capabilities of our method, not with the realism of the air traffic scenario. More realistic scenarios will be the subject of future work. See section 5.

an aircraft's progress towards the waypoint and the pilot's action for directing the corresponding aircraft to the waypoint occur at the same time.

Simulations and verifications were performed on a computer workstation with a 3.6 gigahertz Intel Xeon processor with 128 gigabytes of RAM. Verifications were run on the Linux Desktop using SAL's infinite bounded model checker [19]. SAL's infinite bounded model checker requires a depth (a bound) on the number of transitions considered in a given analyses. For large bounds, verification can take a significant amount of time. Thus, for all verifications, depths were started out at 10 and were iteratively increased by 5 until either a counterexample was found or the analyst was satisfied that the property would not be violated.

4.1 Finding Counterexamples

The generated specifications have three distinct forms: those for finding instances of load [derived from Eqs. (5)–(7)], those for finding instances of a lack of load [based on Eqs. (8) and (9)], and those for finding resource conflicts [created from Eq. (10)]. Thus, while every property type has been tested for its efficacy, the results discussed here focus on one property of each form. Further, because the pilots in the given modeled section only had one action each, it made little sense to consider issues related to their load. Thus load-based specifications were only checked for the air traffic controller.

To start, we attempted to check whether the air traffic controller could ever be overloaded [by checking Eq. (7)]. When verification was performed, a counterexample was found at depth 10 (verification time: 16.56 seconds). This showed that overload could occur in the scenario as originally represented in the WMC scenario.

To find a condition where overload could be avoided, we reran the verification with Eq. (9). This produced no counterexample at depths 10 (verification time: 5.45 seconds) and 15 (verification time: 85.81 seconds). However, at depth 20 (verification time: 479.74 seconds), a counterexample was produced. This showed how actions could be spaced out so as to avoid air traffic controller overload. This scenario is explored in more depth in the subsequent section.

Finally, we attempted to check whether the get-set resource conflict could occur for aircraft one, where the conflict occurs when the air traffic controller's actions for managing an aircraft's progress towards the waypoint and the pilot's action for directing the corresponding aircraft to the waypoint occur at the same time. This produced a counterexample at depth 10 (verification time: 10.7 s). This showed that these two actions could produce a resource conflict in the original scenario.

Collectively, these tests demonstrate the methods ability to identify specification violations and thus generate counterexamples. These counterexamples can be used to create new scenarios.

4.2 Modifying the Simulation Scenario with Counterexamples

Next, we wanted to test that our method could accurately be used to generate a new scenario illustrating the action sequence from a counterexample. Because only the specification that generated a counterexample with no overload [Eq. (9)] represented an action sequence that was different from the original scenario, this was used. Thus, we used our automated method to create a new scenario (a modified version of the original) where the actions were spaced out as specified in the counterexample. A manual examination of the resulting action trace revealed that the rescheduled actions were being performed at the times specified in the counterexample and that these timings avoided overload.

5 Conclusions and Discussion

In this paper, we described a formal modeling architecture designed to enable the discovery of interesting human operator conditions that relate to human taskload and resource conflicts between agents though the synergistic use of formal verification and simulation. The presented method allows us to formally represent all of the relevant WMC concepts while satisfying our objectives. (a) It uses timed automata to represent real-valued time and allows for sensitivity analyses 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 with a different agent; and (c) By using λ operations over the set of actions and by abstracting away WMC details unimportant to the formal analyses, the architecture is computationally efficient. A number of specification properties can be used to reason about taskload for use in generating counterexamples. Finally, we showed how these specification properties could be used to find scenarios to drive WMC simulations to interesting conditions.

Previous work that has used formal verification with simulation has either focused on analyzing limited aspects of a simulation [26, 27, 31, 65] or on only model checking the actual simulation traces [15, 57, 64]. As such, this work makes a significant contribution in that it introduces a new way of using simulation and formal verification synergistically by allowing model checking to be used to explore the space around simulation scenarios. This is a major breakthrough for several reasons. From a formal methods perspective, this approach allows formal verification to be used in the analysis of complex systems that would have been too

big and/or complex to be evaluated in the past. From a simulation perspective, our method allows for a much more complete analysis of the space around a simulation trace than was previously possible. This should allow analysts to use high-fidelity simulations to find potential problems that may not have been found otherwise.

While researchers have investigated how to formally model workload [40, 56], these have been used in simulation environments, not formal verification. Thus, our developments are significant because it constitutes the first effort to account for human operator taskload in formal verification analyses. By using lambda calculus sets as the means of modeling taskload, we have developed a representation that should be computationally tractable in a number of different situations. Given the importance that taskload and workload play in human performance and system safety, this advance will enable formal verification to be used for human factors engineering purposes in new and important ways.

The presented research focused primarily on the technical challenges of realizing our method. It thus neglects a full application of the method in a realistic application. There are also ways that the method could be improved. These issues are explored below.

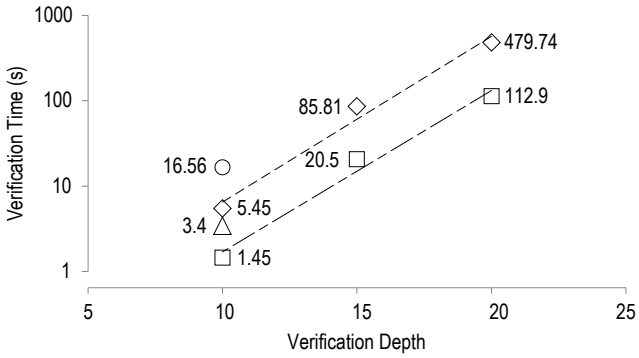

Fig. 4 Semi-log plot showing the verification times for the taskload properties at different depths. ○ and ◇ represent the original verification times [for Eqs. (7) and (9) respectively] where all four human agents are included in the formal model. △ and □ represent the verification of the same properties (respectively), for formal models that only include the air traffic controller. Dotted lines are used to show the exponential trend line for verifications conducted for multiple depths.

5.1 Scalability

An examination of the verification results shows that times increase exponentially with the search depths (see Fig. 4). This is a potentially limiting factor for our analyses as verifications could take so long to run that they would be prohibitive for use in a long, complex simulation scenario.

For properties that relate to taskload, each agent in our formal modeling architecture operates independently of the others. Thus, for these types of specifications, verifications for each agent could be run separately. The resulting new action times from each analysis can then be integrated into the new scenario.

Figure 4 illustrates how the verification times were reduced by only considering the air traffic controller in the formal model for all of the verifications that related to taskload in our application. This shows that, while limiting the number of agents in the model does not avoid exponential increases in verification times, it does substantially reduce the verification times. This will inevitably help in the analysis of scenarios where verification time becomes an issue. Future work should investigate this feature in more depth. It should also explore other options for improving model scalability.

Although not discussed extensively here, our method was designed to be used iteratively, where an analyst can: use the method to modify one portion of a scenario, generate a new scenario, run it, observe the effect, and apply the method on a latter part to further modify the scenario. In this way, small time windows can be used effectively to explore longer scenarios while avoiding scalability issues. Further, the iterative approach to the deployment of the method should allow the impact of minor change early in the scenario to be fully realized in subsequent scenario runs.

It is important to note that even with all of the agents included in the verification analyses, the longest verification time we observed was 479.47 seconds (Fig. 4). In our analyses, the average WMC run took ∼650 seconds. Thus, though the formal verification analyses can take a significant amount of time, they are effectively considering an infinite number of simulation scenarios. As such, the dual use of formal verification and simulation can be viewed as having a scalability advantage in that model checking is allowing multiple scenarios to be explored more efficiently than could be achieved by only running multiple simulations. For example, for the analyses presented in this paper, the scenarios discovered from running the WMC simulation scenario and then applying our model checking process required less computational time than it would have taken to run two full simulation runs. If only simulation were being employed, many more simulations would be required to find the same types of conditions, if they were ever found at all.

5.2 Abstraction

Because the exact timing of actions was critical to the way that taskload was computed (this would be even truer if variance in action timing was included), we modeled it explicitly in our method. However, this is likely a major factor in the verification times reported in the previous section. It is conceivable that abstraction techniques could be used to remove the need to represent real time from the model and thus enable the use of more traditional symbolic model check-

ing. In fact, automated techniques exist for learning such abstractions from hybrid models that contain both discrete and continuous components [58, 59]. Future work should explore how abstraction techniques could be used to potentially improve the scalability of our approach.

5.3 Improvements to the Translation Processes

While automated, the translation processes still require some manual analyst intervention to identify the sizes of priority queues (if they are not part of the WMC model), timing variances, and priorities. Further, when translating a counterexample back into a WMC scenario, our method will automatically create WMC code that must be manually placed in a WMC scenario cpp file, and the WMC C++ source must be recompiled. In future work, we hope to rely less on these manual human operations. To accomplish this, we hope to use XML descriptions of the work models and scenarios. 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 XML files of work models and scenarios to be translated back and forth between WMC simulations and SAL experiments without manual placing of source code and WMC compilation. The XML Parser currently translates work model and scenario XML files into WMC and WMC scenarios into XML. Current XML files are parsed and XML statements are subsequently generated and populate the WMC template. We envision the capability for WMC to XML Specifications to parse WMC C++ files and dynamically generate XML tag statements during runtime to populate XML template files.

5.4 Additional Sources of Variance

Our method only allows for variance of action timing and sequencing due to non-determinism in action prioritization. This allowed us to use formal verification to test the taskload and resource conflict properties we were interested in. However, it does not allow for variations in the actions that are performed due to human error or changes in the environmental conditions. These could also impact taskload and resource conflicts by introducing additional actions or time constraints. Work within the extended formal methods literature has focused on how human error and anomalous system conditions can be generated in formal models so that verifications can assess their impact on system performance [3–7, 42]. Future work should investigate how these approaches could be adapted for use in our method.

5.5 Expanded Application

The AAR Studies application presented here showed that our method was capable of finding the desired conditions around simulation traces and generating new simulation scenarios from those results. While the total amount of time represented in the model was small (5 seconds), the method was able to handle scenarios with 4 separate agents with 9 possible actions between them and produce results in a reasonable amount of time.

While the results presented here were capable of testing the intellectual contributions of our work, the analyses do not give us any particular insights into the larger performance of the represented air traffic control scenario. Further, the balanced FA3-FA3 distribution we tested is only one of many possible distributions of authority and responsibility offered by the AAR Studies scenarios. Future work with the method will explicitly focus on using the new capabilities of the method to more deeply and completely explore how taskload and resource conflicts could impact air traffic control performance under different distributions of authority and responsibility.

Acknowledgement

This work was supported by the grant “Scenario-Based Verification and Validation of Autonomy and Authority” from the NASA Ames Research Center under award number NNX13AB71A.

References

1. Alur R, Dill DL (1994) A theory of timed automata. Theoretical Computer Science 126(2):183–235
2. Bainbridge L (1983) Ironies of automation. Automatica 19(6):775–780
3. Bolton ML, Bass EJ (2008) Using relative position and temporal judgments to identify biases in spatial awareness for synthetic vision systems. The International Journal of Aviation Psychology 18(2):1050–8414
4. Bolton ML, Bass EJ (2009) Comparing perceptual judgment and subjective measures of spatial awareness. Applied Ergonomics 40(4):597–607
5. Bolton ML, Bass EJ (2013) Generating erroneous human behavior from strategic knowledge in task models and evaluating its impact on system safety with model checking. IEEE Transactions on Systems, Man and Cybernetics: Systems 43(6):1314–1327
6. Bolton ML, Bass EJ, Siminiceanu RI (2012) Generating phenotypical erroneous human behavior to evaluate humanautomation interaction using model check-

ing. International Journal of Human-Computer Studies 70(11):888–906
7. Bolton ML, Bass EJ, Siminiceanu RI (2013) Using formal verification to evaluate human-automation interaction in safety critical systems, a review. IEEE Transactions on Systems, Man and Cybernetics: Systems 43(3):488–503
8. Bolton ML, Jimenez N, van Paassen MM, Trujillo M (2014) Automatically generating specification properties from task models for the formal verification of human-automation interaction. IEEE Transactions on Human-Machine Systems 44:561–575
9. Bredereke J, Lankenau A (2002) A rigorous view of mode confusion. In: Proceedings of the 21st International Conference on Computer Safety, Reliability and Security, Springer, London, UK, pp 19–31
10. Bredereke J, Lankenau A (2005) Safety-relevant mode confusions–modelling and reducing them. Reliability Engineering and System Safety 88(3):229–245
11. Buth B (2004) Analyzing mode confusion: An approach using FDR2. In: Proceeding of the 23rd International Conference on Computer Safety, Reliability, and Security, Springer, Berlin, pp 101–114
12. Butler RW, Miller SP, Potts JN, Carreño VA (1998) A formal methods approach to the analysis of mode confusion. In: Proceeding of the 17th Digital Avionics Systems Conference, IEEE, Piscataway, pp C41/1–C41/8
13. Campos JC, Harrison M (2001) Model checking interactor specifications. Automated Software Engineering 8(3):275–310
14. Campos JC, Harrison MD (2011) Modelling and analysing the interactive behaviour of an infusion pump. In: Proceedings of the Fourth International Workshop on Formal Methods for Interactive Systems, EASST, Potsdam
15. Chen X, Hsieh H, Balarin F, Watanabe Y (2003) Automatic trace analysis for logic of constraints. In: Proceedings of the Design Automation Conference, IEEE, pp 460–465
16. Clarke EM, Grumberg O, Peled DA (1999) Model checking. MIT Press, Cambridge
17. Committee on Autonomy Research for Civil Aviation; Aeronautics and Space Engineering Board; Division on Engineering and Physical Sciences; National Research Council (2014) Autonomy Research for Civil Aviation: Toward a New Era of Flight. National Academy of Sciences, Washington DC
18. de Moura L, Owre S, Shankar N (2003) The SAL language manual. Tech. Rep. CSL-01-01, Computer Science Laboratory, SRI International, Menlo Park
19. De Moura L, Owre S, Rueß H, Rushby J, Shankar N, Sorea M, Tiwari A (2004) Sal 2. In: International Conference on Computer Aided Verification, Springer, pp 496–500
20. Degani A (2004) Taming HAL: Designing interfaces beyond 2001. Macmillan, New York
21. Degani A, Heymann M (2002) Formal verification of human-automation interaction. Human Factors 44(1):28–43
22. Derrick J, North S, Simons T (2006) Issues in implementing a model checker for Z. In: International Conference on Formal Engineering Methods, Springer, pp 678–696
23. Dutertre B, Sorea M (2004) Timed systems in SAL. Tech. Rep. NASA/CR-2002-211858, SRI International
24. Emerson EA (1990) Temporal and modal logic. In: van Leeuwen J, Meyer AR, Nivat M, Paterson M, Perrin D (eds) Handbook of Theoretical Computer Science, MIT Press, Cambridge, chap 16, pp 995–1072
25. Feigh KM, Pritchett AR, Mamessier S, Gelman G (2014) Generic agent models for simulations of concepts of operation: part 2. Journal of Aerospace Information Systems
26. Gelman G, Feigh KM, Rushby J (2013) Example of a complementary use of model checking and agent-based simulation. In: IEEE International Conference of Systems Man and Cybernetics, IEEE, Piscataway, pp 900–905
27. Gelman G, Feigh K, Rushby J (2014) Example of a complementary use of model checking and human performance simulation. IEEE Transactions on Human-Machine Systems 44(5):576–590
28. Gelman GE (2012) Comparison of model checking and simulation to examine aircraft system behavior. PhD thesis, Georgia Institute of Technology
29. Hart SG, Staveland LE (1988) Development of NASA-TLX (Task Load Index): Results of empirical and theoretical research. Advances in psychology 52:139–183
30. Houser A, Ma LM, Feigh K, Bolton ML (2015) A formal approach to modeling and analyzing human taskload in simulated air traffic scenarios. In: 2015 International Conference on Complex Systems Engineering, pp 1–6
31. Hu AJ (2008) Simulation vs. formal: Absorb what is useful; reject what is useless. In: Proceedings of the Third International Haifa Verification Conference, Springer, Berlin, pp 1–7
32. IJtsma M, Hoekstra J, Bhattacharyya RP, Pritchett A (2015) Computational assessment of different air-ground function allocations. In: Eleventh USA/Europe Air Traffic Management Research and Development Seminar, 10 pages
33. IJtsma M, Pritchett AR, Bhattacharyya RP (2015) Computational simulation of authority-responsibility mismatches in air-ground function allocation. In: Proceedings of the 18th International Symposium on Aviation

Psychology, Write State University, Dayton, 6 pages
34. Joshi A, Miller SP, Heimdahl MP (2003) Mode confusion analysis of a flight guidance system using formal methods. In: Proceedings of the 22nd Digital Avionics Systems Conference, IEEE, Piscataway, pp 2.D.1-1–2.D.1-12
35. Loft S, Sanderson P, Neal A, Mooij M (2007) Modeling and predicting mental workload in en route air traffic control: Critical review and broader implications. Human Factors 49(3):376–399
36. Lüttgen G, Carreño V (1999) Analyzing mode confusion via model checking. In: Proceeding of Theoretical and Practical Aspects of SPIN Model Checking, Springer, Berlin, pp 120–135
37. Mackworth JF (1964) Performance decrement in vigilance, threshold, and high-speed perceptual motor tasks. Canadian Journal of Psychology 18(3):209–223
38. McFarlane DC, Latorella KA (2002) The scope and importance of human interruption in human-computer interaction design. Human-Computer Interaction 17(1):1–61
39. Miller GA (1956) The magical number seven, plus or minus two: Some limits on our capacity for processing information. Psychological Review 63(2):81
40. Moore J, Ivie R, Gledhill T, Mercer E, Goodrich M (2014) Modeling human workload in unmanned aerial systems. In: 2014 AAAI Spring Symposium Series, AAAI, Palo Alto, pp 44–49
41. Palmer E (1995) "Oops, it didn't arm"- A case study of two automation surprises. In: Proceedings of the 8th International Symposium on Aviation Psychology, Wright State University, Dayton, pp 227–232
42. Pan D, Bolton ML (ND) Properties for formally assessing the performance level of human-human collaborative procedures with miscommunications and erroneous human behavior. International Journal of Industrial Ergonomics DOI http://dx.doi.org/10.1016/j.ergon.2016.04.001
43. Popescu V, Clarke J, Feigh KM, Feron E (2011) ATC taskload inherent to the geometry of stochastic 4-d trajectory flows with flight technical errors. CoRR abs/1102.1660
44. Pritchett A, Feigh K (2011) 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, IEEE, Piscataway, pp 144–151
45. Pritchett AR (2013) Simulation to assess safety in complex work environments. In: Lee JD, Kirlik A (eds) The Oxford handbook of cognitive engineering, Oxford University Press, New York, chap 22, pp 352–366
46. Pritchett AR, Feigh KM, Kim SY, Kannan SK (2014) Work models that compute to describe multiagent concepts of operation: Part 1. Journal of Aerospace Information Systems 11(10):610–622
47. Pritchett AR, Kim SY, Feigh KM (2014) Measuring human-automation function allocation. Journal of Cognitive Engineering and Decision Making 8(1):52–77
48. Pritchett AR, Kim SY, Feigh KM (2014) Modeling human–automation function allocation. Journal of Cognitive Engineering and Decision Making 8(1):33–51
49. Reason J (1990) Human Error. Cambridge University Press, New York
50. Rushby J (2002) Using model checking to help discover mode confusions and other automation surprises. Reliability Engineering and System Safety 75(2):167–177
51. Rushby J, Crow J, Palmer E (1999) An automated method to detect potential mode confusions. In: Proceedings of the 18th Digital Avionics Systems Conference, IEEE, Piscataway, pp 4.B.2–1–4.B.2–6
52. Sarter NB, Woods DD (1995) How in the world did we ever get into that mode? Mode error and awareness in supervisory control. Human Factors 37(1):5–19
53. Sheridan TB, Parasuraman R (2005) Human-automation interaction. Reviews of Human Factors and Ergonomics 1(1):89–129
54. Sherry L, Feary M, Polson P, Palmer E (2000) Formal method for identifying two types of automation-surprises. Tech. Rep. C69-5370-016, Honeywell, Phoenix
55. Smith G, Wildman L (2005) Model checking Z specifications using SAL. In: ZB 2005: Formal Specification and Development in Z and B, Springer, pp 85–103
56. Stocker R, Rungta N, Mercer E, Raimondi F, Holbrook J, Cardoza C, Goodrich M (2015) An approach to quantify workload in a system of agents. In: Proceedings of the 14th International Conference on Autonomous Agents and Multiagent Systems, IFAAMAS, Liverpool
57. Stuart DA, Brockmeyer M, Mok AK, Jahanian F (2001) Simulation-verification: Biting at the state explosion problem. IEEE Transactions on Software Engineering 27(7):599–617
58. Tiwari A (2003) Hybridsal: Modeling and abstracting hybrid systems. Tech. rep.
59. Tiwari A, Khanna G (2002) Series of abstractions for hybrid automata. Hybrid Systems: Computation and Control pp 425–438
60. Vela A, Feigh KM, Solak S, Singhose W, Clarke JP (2012) Formulation of reduced-taskload optimization models for conflict resolution. IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans 42(6):1552–1561
61. Weyers B, Bowen J, Dix A, Palanque P (eds) (2017) The Handbook of Formal Methods in Human-Computer Interaction. Springer, Cham

62. Wheeler PH (2007) Aspects of automation mode confusion. Master's thesis, Massachusetts Institute of Technology, Cambridge
63. Wing JM (1990) A specifier's introduction to formal methods. Computer 23(9):8, 10–22, 24
64. Yasmeen A, Feigh KM, Gelman G, Gunter EL (2012) 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, pp 71–81
65. Yuan J, Shen J, Abraham J, Aziz A (1997) On combining formal and informal verification. In: Computer Aided Verification, Springer, pp 376–387