Formal Analysis of Safety-Critical System Simulations

Ayesha Yasmeen*, Karen M. Feigh†, Gabriel Gelman† and Elsa L. Gunter*
yasmeen@illinois.edu, karen.feigh@gatech.edu, g.gelman@gatech.edu, egunter@illinois.edu
*University of Illinois, 201 N Goodwin Avenue, Urbana, IL 61801, USA, +1(217)265-6118
†Georgia Institute of Technology, 270 Ferst Drive, Atlanta, GA 30332, USA, +1(404)385-7686

ABSTRACT

Safety-critical systems are often large and complex. Usually it is not physically or economically feasible to operate these systems under all variant environmental conditions to analyze possible behaviors. Simulating system behaviors under various different environmental conditions and operator guidance patterns provides a cost-effective method of system analysis. In this work we demonstrate how we can formally encode and analyze voluminous simulation traces of safety-critical systems to assess safety and effectiveness requirement conformation. We provide methodology for trace reduction to help obtain tractable yet meaningful formal encoding of the traces. Our methodology is flexible in the sense that one single trace can be analyzed from the point of view of many different properties without having to incur the cost of regenerating the trace. Experimental analysis of a simulation trace can help obtain valuable insights into possible reasons for nonconformance with system requirements. We present our implementation results for traces ensuing from aircrafts attempting to perform Continuous Descent Approach for landing at airport runways. Our work demonstrates that, with the help of faithful abstractions we can obtain valuable insights about simulated traces by the formal verification procedures irrespective of the size of the simulation traces. Formal verification methodology allows for intuitive, expressive yet succinct formulation of system requirements. The combination of simulation trace generation and formal verification provide feedback that may (i) assess the appropriateness of the requirement specifications, (ii) suggest possible infidelity in the simulation modules and (iii) even delineate design error of the original safety-critical system.

Categories and Subject Descriptors

D.2.4. [**Software Engineering**]: Software/Program Verification—*Formal methods, Model checking*; F.4.1. [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*Temporal logic*; I.6.6. [**Simulation and Modeling**]: Simulation Output Analysis; I.6.6. [**Simulation and Modeling**]: Model Validation and Analysis

General Terms

Human Factors; Reliability;Verification.

Keywords

Simulation, formal verification, trace analysis, automated model generation, model checking, model reduction, safety-critical systems, temporal properties

INTRODUCTION

Failure of safety-critical systems can lead to tremendous even fatal consequences. Safety-critical systems thus must be thoroughly analyzed before deployment. Real life safety-critical systems are often very large and complex. Different approaches exist towards ensuring high integrity of safety-critical systems. Among these simulation is quite prominent. Simulation provides a detailed system model including almost all known factors and behaviors of the system. Simulation of safety-critical systems is advantageous for behavior analysis, as often the real systems cannot be used to analyze against many possible alternative environmental conditions. Simulation generates traces for system domain specialists who may get some automated rendition of the data like visualization. Another prominent system verification technique is formal verification. Formal verification entails obtaining an abstract model of the original system and then assessing that abstract model for safety and effectiveness requirement conformance. The abstract models are often variants of finite state machines and the desired properties are often specified using some form of declarative logic like temporal logic [12]. A great advantage of declarative logic is that is allows a succinct yet intuitive method of specifying requirements. Formal declarative specification of system requirements can be more easily expressible and understandable than descriptions of complex procedural modules created to assess conformance to system requirement. Automated formal tools ease the task of verifying a system against logical property statements. Formal verification is capable of checking each model against complex properties that may need to hold for all possible behaviors of a system. But formal verification methodologies often suffer from the fact that their computationally expensive capabilities demand immense abstractions of the system. Formal models need to become more abstract as systems grow in size and complexity. Properties likewise can become more abstract. Simulations, on the other hand, contain more faithful detail about possible system behavior, but only of one example behavior. Formal verification thus allows for highly expressive property analysis at the cost of system abstraction. Simulation provides a more faithful record of system states for one particular behavior. Usually there is no mechanism for asserting properties about the ensuing traces. Simulation cannot provide any requirement conformation guarantee. Formal verification can provide understanding about safety, effectiveness and fairness requirements where their results are inversely affected by the degree of abstraction from the actual system. We bring these two seemingly diverse directions of research together to provide improved understandings of system behavior. To obtain this enhanced analysis, we will first simulate a system to obtain as realistic information as possible. Then we will perform formal verification of the simulated traces to obtain formal, fine-grained yet intuitive analysis of the system behavior. Our goal is to enhance the information that can be learned from simulated behaviors. We provide formal methodology to aid in assessing system simulation traces for desired property conformations.

Safety-critical systems are usually combinations of numerous modules where each module implements complex algorithms. Traces generated by the simulation process takes the focus away from the internal workings of safety-critical systems. Instead it requires one to focus on actual data resulting from the interactions among the system modules and environmental components. Each simulation trace stands for a particular combination of environmental and system interactions. Formal analysis of safety-critical systems under all possible variant environmental conditions is impractical in most cases due to both the expanse of possible environment variations and the intricacy and size of just the system itself. Given a set of prominent and interesting environmental conditions, system behavior in conjunction with each one of these possible conditions can be simulated and then formally analyzed. The analyses results will then provide a reasonable understanding of the robustness of the system itself. For example, human operators are prominent members of any safety-critical system's environment. Instead of analyzing against all possible deviant human action choices, we can analyze against representative and well established patterns of human operator deviations from their recommended behavior to assess robustness of the system against human operators [2, 16]. Formal simulation trace analysis will also have the benefit that analysis of one particular trace can be performed against many different properties. If the analysis results indicate possible anomaly, then the property can be made more rigid or flexible and the analysis can be performed again without requiring regeneration of the simulated trace or the formal encoding. Thus, if necessary, we can use exploratory properties to determine possible sources of error. In the case of system designers being incapable of exactly formulating requirement properties of the simulated behaviors, our methodology will allow them to start from verification against a simple coarse-grained property specification. They can then refine that property in successive iterations to finally achieve the exact fine-grained desired property for the simulation traces. Moreover, if each simulation trace can be annotated by the software module or operator agent that controls that particular simulation step, then formal analysis will even be able to reason about the agents possibly responsible for anomalous behavior. Thus, formal simulation analysis will provide a powerful and flexible formal simulation analysis methodology.

Formal modeling techniques require specification statement, model generation and then model verification. We extend this methodology for analyzing simulation traces. Safety-critical systems tend to have infinite range and continuously evolving data types leading to a state explosion problem for verification tools resulting in high degrees of abstraction to ensure computability within the computational and memory capacities of today's computing powers. Simulation traces contain immense sequences of snapshots of these parameters. We provide a methodology by which we can formally process such immense collections of data snapshots. Our methodology yields understandings about whether the immense collections of simulated data conform to complex and temporal system requirements. We have applied our technique to verify simulated traces of aircrafts attempting to land at airports using Continuous Descent Approach (CDA) [6]. The simulation traces are generated using the Work Models that Compute (WMC) simulator [13]. We have gained insights into how refinements of original, naively formulated safety properties are required in real life. We have obtained evidence of possible modeling infidelity of the simulation modules themselves. We have confirmed with details a categorization of simulated traces as safe and unsafe. We also show how we can determine adherence of simulated human pilot behaviors to their recommended behaviors to ensure safe operation by the aircraft automation.

The rest of the paper is organized as follows: in the next section we present other works related to our work. We present our methodology for formal verification of simulation traces in the section titled FORMAL VERIFICATION. We present the scenario we have considered in this work in the section titled SCENARIO. Then we present how we have applied our methodology for verifying the automations involved for this scenario in the section titled TRACE ANALYSIS. We present our findings towards the end of that section. Finally we present our future goals and concluding remarks in the last section.

RELATED WORKS

Various research has been undertaken to bring together the informative single trace generation capability of simulation tools and the universal trajectory analysis methodology of formal verification. A closely related work is that of Stuart *et al.* In [14] they combine the strengths of simulation and formal verification to reduce the state space explosion issue. They first use simulation to generate a trace prefix. Formal state space exploration is then performed starting from that trace prefix. However, the properties that they can handle are not as expressive as temporal logic. And also in the case of complex systems like aircrafts, a simulation trace prefix itself can be extremely large, let alone possible trace extensions. Girard and Pappas present a verification technique for infinite state systems in [7] where exhaustive state space exploration verification methodology is approximated by a finite number of simulations. We similarly suggest formal temporal property analysis on top of obtaining a representative collection of simulated traces. Narayanan and McIlraith present a technique for modeling, simulating and analyzing systems through Petri Nets in [11]. They provide property conformance through state space intersection checking. Kemper and Tepper present stochastic simulation model analysis techniques in [10]. They provide algorithms to distinguish progressive behavior from repetitive behavior in a trace. They also provide a trace reduction algorithm to extract a minimal progressive fragment of a trace. Another related research direction is that of monitoring simulation process. Simulation monitoring may be inlined or performed post-hoc. If monitoring is inlined in the simulation process then that requires advanced precise formulation of the desired property. Any change in the property will result in performing the expensive simulation again. Post-hoc simulation monitoring usually involves innovative and informative visual display. Runtime verification via monitoring is related in the sense that runtime monitoring monitors a trace execution for desired property conformation while the trace is getting generated [1]. Runtime verification of simulation tool was not considered in this work as this would have required integrating numerous complex runtime monitoring modules with the simulation software. The CDA simulator is a complex enough software with considerable computational complexity. Complex integration with runtime monitoring tools would further add to that complexity. Additionally any change in the monitoring approach or the property would have to trickle down to relevant simulator modules. Also the most prominent runtime verification tools require Java platform, whereas the simulator used in this work has been developed in C++.

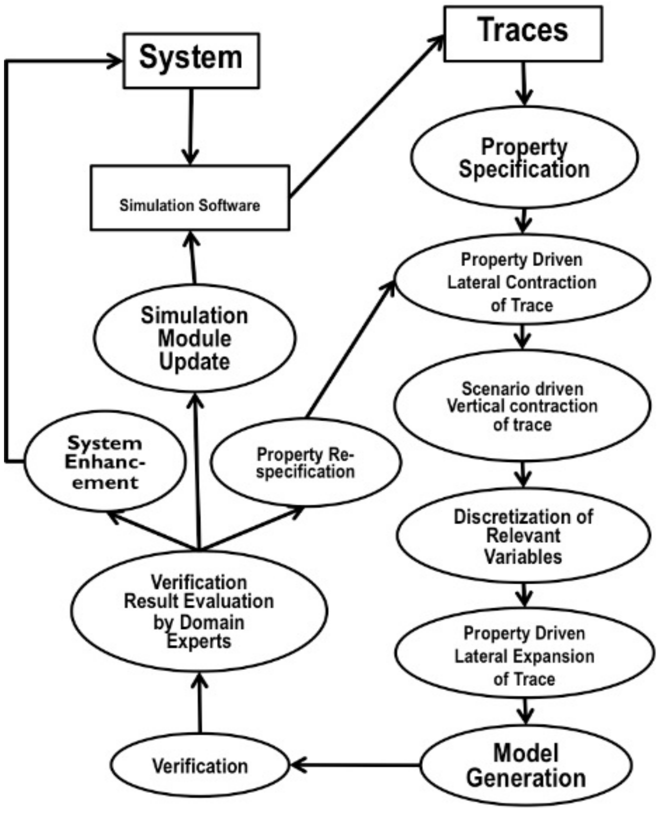
System
Traces
Simulation Software
Property Specification
Property Driven Lateral Contraction of Trace
Simulation Module Update
Scenario driven Vertical contraction of trace
System Enhancement
Property Re-specification
Discretization of Relevant Variables
Verification Result Evaluation by Domain Experts
Property Driven Lateral Expansion of Trace
Verification
Model Generation

Figure 1. Methodology for Formal Trace Verification

Chen *et al.* present a simulation trace analysis methodology in [5] where they show how simulation traces can be checked against properties expressed in the Logic of Constraints (LOC). LOC is a logic created to express simulation trace properties. However, temporal logic properties are more expressive than LOC properties if the system is augmented with necessary record keeping variables. They also provide a simulation monitoring technique in [4] where the simulation trace generation process is monitored against LOC properties. All these works provide simulation trace analysis methodologies where the system requirement is specified in terms of sets of safe states or simple logical formulae. We contend that having the expressiveness of temporal properties provides a more intuitive mechanism to system designers to provide their understanding of the system requirements instead of delineating all possible states that satisfy system requirements and goals.

FORMAL VERIFICATION

The formal system verification process involves collection of desired properties, encoding of the system in a suitable formalism, performing abstractions to the model as necessary and then using some existing verification tool to verify conformance to the aforementioned properties. We augment this methodology as depicted in Figure 1 for analyzing simulation traces. We focus on automated model checking tools for formal verification in this work. In the figure, the System stands for the safety-critical system under consideration. The design and specification of the system drives the simulation software. Usually the system is modeled using a collection of simulation modules. The simulation engine executes the simulation modules possibly under some specific conditions of the system environment. The execution of simulation modules generates traces that are the focus of this work.

Property Specification

Simulation traces are sequences of collections of values for system parameters embodied in simulation variables. The set of system parameters available in a trace provides an understanding of the possible declarative logic propositions at our disposal for formulating system requirements like safety and effectiveness. The set of available propositions are predicates on sets of simulation variables. Using these propositions, one can then formulate the desired properties in a declarative logic. The declarative logic can vary from simple predicate logic to Linear Temporal Logic (LTL) and even Computation Tree Logic (CTL). For this work we focused on LTL properties, as we are always concerned about one linear execution path contained in a simulation trace. The syntax of a linear temporal logic formula is as follows: $\varphi = p \mid \neg\varphi \mid \bigcirc\varphi \mid \Diamond\varphi \mid \Box\varphi \mid \varphi_1 \mathcal{U} \varphi_2$. Here p stands for atomic propositions that may hold of a state; $\neg\varphi$ indicates that the property φ does not hold of the trace starting in the current state; $\bigcirc\varphi$ indicates that in the *next* state the property φ will hold. $\Box\varphi$ indicates that φ will hold at the current state and at all states in future; and $\Diamond\varphi$ indicates that eventually φ will hold at some point in the future. $\varphi_1 \mathcal{U} \varphi_2$ indicates that the property φ_1 will continue to hold along a path until the property φ_2 holds. $\mathcal{U}$ is considered to be a *strong* until property as φ_2 must hold at some point in future.

Trace Reduction

Complex scenarios like aviation systems often yield extremely large traces involving values for large collections of parameters. Formal models generated from complex and large scenarios can easily become intractable for verification tools. Moreover, most of the parameters in such scenarios usually range over infinite sets of possible values. And even if specific ranges of values can be considered for the sake of simplicity, the enormous volume of steps contained in a simulation trace can still be computationally expensive. Thus for verification purposes, trace reduction is an essential step. However, this abstraction is not the usual system abstraction where complex, rigorous details of system modules are abstracted away. Our abstractions are performed on the traces ensuing from the simulation model of the system where the simulation models are usually not very abstract. We show three essential steps of trace reduction for simulation trace verifications. These trace model reduction steps will help generate formal models that will be tractable yet meaningful. Let $T(\Sigma)$ be a trace where each trace step contains values for each variable v in the set Σ.

Lateral Contraction of Trace, $\mathcal{L}$

The requirement specification step holds the key to this model reduction step. Usually requirements are specified in terms of predicates involving a very small subset of the different parameters tracked at every time step by a simulation engine. Thus at any point in time, we can focus just on the values assigned to that specific set of trace parameters. As a simple model reduction step we can then focus on just these variables of interest and possibly other variables that directly influence these variables in the formal model generation process: $\mathcal{L}(T(\Sigma)) = T(\Sigma')$ where $\Sigma' \subseteq \Sigma$. Let Π be the set of simulation variables appearing in a set $\mathcal{P}$ of LTL properties. Then, a simple lateral contraction step $\mathcal{L}(T(\Sigma))$ will yield a trace $T(\Pi)$ where $\Pi \subseteq \Sigma$. Our assumption is that the LTL formulae that will get asserted on the trace models will be built with propositions over values contained in state variables. Lateral contraction of traces does not affect the values contained in state variables, but simply separates out interesting pieces of a trace. Since we assert properties on the contracted traces that we want to hold in the excerpted finite segment of a given trace, lateral contraction of traces does not adversely affect LTL property satisfaction in our case.

Discretization of Simulation Variables, $\mathcal{D}$

Continuous data like floating point data is handled by very few model checking tools. Even integral data with large, possibly infinite range of values can pose problems for model checking tools due to increase in state space calculation complexity. As a result we suggest discretization of variables as an essential model reduction step. Both continuous and integer variables can be scaled to a manageable discrete range of values. $\mathcal{D}(T(\Sigma)) = T(\Sigma')$ where $\Sigma' = \{\mathcal{D}(v) | v \in \Sigma\}$. The discretization function $\mathcal{D}$ may truncate and scale the parameters to values in some specific range. Discretization of variables comes at the cost of loss of granularity both in the abstract models and the meaningful properties that can be asserted on them.

Vertical Contraction of Trace, $\mathcal{V}$

Safety-critical system traces can often be extremely large. However, if the scenario permits, we can only focus on some specific slices of the trace. The slices can belong to consecutive or disjoint parts of the trace. Let $\mathcal{V}(T(\Sigma)) = T'(\Sigma)$. Then we must have $\forall j_1, j_2. \exists k_1, k_2. T'_{j_1} = T_{k_1} \wedge T'_{j_2} = T_{k_2} \wedge j_1 \leq j_2 \Rightarrow k_1 \leq k_2$.

Lateral Expansion of Trace, $\mathcal{E}$

Requirement properties for traces are often expressed not only in terms of the current value of simulation variables but also on historical values or the change of value occurring in the variables of interest. Thus trace models may very well need to be annotated with additional variables to keep track of changes occurring in the simulation variables: $\mathcal{E}(T(\Sigma)) = T(\Sigma')$ where $\Sigma \subseteq \Sigma'$. Usually, $\Sigma' = \Sigma \cup \bigcup_{\forall \sigma \in \Sigma} \sigma'$ where σ' stands for a history variable introduced for a simulation parameter σ.

Formal Model for Simulation Traces

In order to perform formal trace verification, we need to first translate the traces into formal models. We convert each simulation trace into a finite state automata where combinations of the values of the relevant simulation variables collectively form the state space. Each variable update leads to a transition from one state to another. We propose direct translation of a simulation trace into the input language for the verification tool of choice. We need to generate a formal model for a large sequence of interactions among agents and automations contained in the simulation traces. Vital statistics of the automated systems are represented by appropriate simulation variables in the simulation trace. They can be directly translated into appropriate formal variables. The interactions among the agents and automations can be easily modeled as communications among the relevant entities. For example, asynchronous unbuffered communication channels can be used to model a display dial of an automation. Synchronous unbuffered communication or asynchronous buffered communication can be used to model communication among human operators or among human operators and automations. Although the translation procedure from raw information to formal model may vary from scenario to scenario, the idea remains the same: (i) identify system variables, important agents and their actions involved a scenario, (ii) ascertain the system variables affected by those actions, (iii) model communications involving relevant variables using relevant actions by the agents, (iv) model the traces with appropriate variable updates and inter-agent communications. Every variable update action will modify the state of the formal model and appropriate collection of properties can then be verified against that state space. Each action of an agent is actually a collection of resource updates or communications. Thus each agent action can be modeled as functions using combinations of appropriate translations for the resource updates and communications involved.

Verification and Analysis of Verification Results

The formal model can be analyzed against specified properties using existing verification tools and the results can then be conveyed to the appropriate domain experts. The domain experts may decide that the system model failed to conform

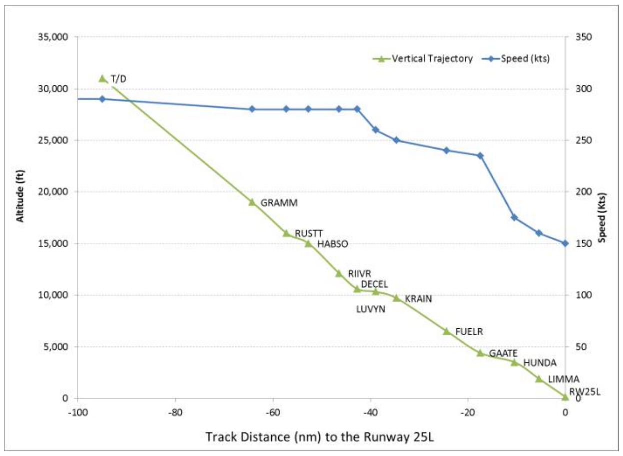

Figure 2. Vertical profile of the nominal (continuous descent) arrival and approach scenario

to some requirement as a result of possible errors in the translation procedure, flawed or overly restrictive property specification, erroneous simulation module or even worse, a design flaw in the original system. These results can lead to changing the formalization procedure, property re-specification, simulation module update or even modification of the original system. If there is a possible error in the formal encoding then it can be corrected without changing the simulation trace. If the desired property turns out to be unrealistic then we can analyze the same simulation trace with respect to many different variations of the property without incurring the cost of generating the trace or the formal model repeatedly. If simulation model change or original system flaw elimination is required then the entire process can be performed again to obtain a new formal encoding to verify. However, given the flat structure of the simulation traces and encoding methodology suggested by us, an automated translator from simulation traces to formalisms can be easily developed. Thus formal trace verification can help improve the integrity of a safety-critical system even before it gets deployed. We now present the example scenario we worked with to show how our methodology works in practice.

SCENARIO

The scenario used for analysis in this paper is that of a Boeing 747-400 aircraft executing a continuous descent approach in to LAX following the RIIVR TWO (http://155.178.201.160/d-tpp/1202/00237RIIVR.PDF) standard arrival route to runway 25L. Unlike a standard arrival, a continuous descent approach does not include any level segments. Instead, at the top of descent, the pilot pitches the aircraft over, reduces thrust and attempts to land the aircraft following a continuous descent path without leveling out or adding additional thrust. The goal of such a descent is to reduce emissions and noise at lower altitudes in the vicinity of airports. The vertical approach path (see Figure 2) is described by the RIIVR TWO ARRIVAL standard arrival route and a standard approach procedure. This arrival is suitable for aircrafts arriving from the northeast into LAX. The arrival and

Function Allocation	Description	
FA1	Highly-automated	Pilot using LNAV/VNAV with air traffic instructions directly processed by the flight deck automation.
FA2	Mostly-automated	Pilot using LNAV/VNAV with pilot receiving air traffic instructions and programming the autoflight system.
FA3	Mixed-automated	Pilot updating the vertical autoflight targets and receiving air traffic instructions, and the FMS commanding the lateral autoflight targets.
FA4	Mostly-manual	Pilot programming the autoflight targets and receiving air traffic instructions.

Table 1. Modeled Function Allocations

Lateral/Vertical Location	ATC Instruction	Pilot's Action FA 1	Pilot's Action FA 3
At Start / FL310		Verify T/D Point	Verify T/D Point
Towards T/D point	Altitude Clearance to 19,000ft	Verify Crossing Restriction Receive Altitude Clearance Dial Altitude Selector	Verify Crossing Restriction Receive Altitude Clearance Dial Altitude Selector
At T/D point		Confirm Active Waypoint Confirm Altitude Target Confirm Speed Target	Confirm Active Waypoint Dial Altitude Selector Dial Vertical Speed Selector Push Vertical Speed Switch Dial Speed Selector Push Speed Switch
Towards GRAMM	Altitude Clearance to 12,100ft	Verify Crossing Restriction Receive Altitude Clearance Dial Altitude Selector	Verify Crossing Restriction Receive Altitude Clearance Dial Altitude Selector
Towards RUSTT / At FL180		Perform Approach Checklist	Perform Approach Checklist
Towards RIIVR / About 13,000ft	Hand off Aircraft	Respond to Handoff	Respond to Handoff
At RIIVR, DECEL, LLUVYN, KRAIN		Confirm Active Waypoint Confirm Altitude Target Confirm Speed Target	Confirm Active Waypoint Dial Altitude Selector Dial Vertical Speed Selector Push Vertical Speed Switch Dial Speed Selector Push Speed Switch
Towards FUELR	Altitude Clearance to 150ft (Runway)	Verify Crossing Restriction Receive ILS Clearance Dial Altitude Selector	Verify Crossing Restriction Receive ILS Clearance Dial Altitude Selector
At RWY 25L			Simulation Ends

Table 2. Differences in Pilot Actions between FA1 & FA3

approach phases are initiated when the aircraft reaches the top of descent (T/D) point, which is the position for starting a descent to achieve an optimal fuel usage, an expected time of arrival, or both as calculated by the aircraft automation. The simulation starts with the aircraft flying at 31,000ft and approximately 30nm from the T/D point towards the waypoint GRAMM and ends when the aircraft reaches 150 feet on final approach. The approach as shown in Figure 2 represents the ideal case of the arrival and approach phases. In this scenario, the air traffic controller clears the aircraft at appropriate times to lower altitudes successively (19,000ft, 12,100ft, 9700ft, 4,400ft, and 150ft). The air traffic controller clears the aircraft by providing descent instructions that may specify the entire arrival route, a certain waypoint, or simply a lower altitude. However, this does not imply the aircraft can fly arbitrarily to the cleared altitude. The aircraft is still required to maintain the indicated waypoints and associated altitude and/or speed restrictions. If the next clearance is not given after the aircraft reaches the currently-cleared altitude, the aircraft must level off and stay at the cleared altitude until the new clearance is issued by the air traffic controllers.

Similar to many other civil transport aircrafts, B747-400 has many different ways in which the pilot can fly it. Each of these methods results in a different distribution of vital functions between the onboard automation, i.e. flight management system (FMS) and the crew. This distribution of responsibility is called function allocation. We will describe this scenario using four different function allocations: Highly Automated, Mostly Automated, Mixed, Mostly Manual. Table 1 provides their definitions. Table 2 illustrates the differences in the pilot actions which need to be modeled between the Highly-automated (FA1) and the Mixed-automated (FA3) modes for some waypoints. The first column of Table 2 indicate the approximate location at which these actions happen with respect to the lateral and longitudinal profile. The second column indicates if the air traffic controller triggered the actions. For example, in the second row, before the aircraft reaches the top of descent point (T/D), the air traffic controller will clear the aircraft to 19,000ft. Without this clearance, the pilot may not initiate a descent even if they pass the point where the aircraft would nominally begin its descent. Pilot actions include activities such as *verify*, *dial*, *push*, receive, and confirm. *Confirm* indicates that the pilots check that the automation has been correctly programmed with the flyight plan. *Verify* indicates that the pilots confirm that they are aware of an upcoming event. In addition to these actions triggered by air traffic controller actions or locations along the flight route, additional actions are modeled for the pilot and the aircraft. Additional actions for pilots include managing or monitoring waypoint progress, monitoring heading, monitoring altitude, monitoring descent airspeed, monitoring expected location of where the aircraft would be laterally when it reached an altitude restriction, deploying flaps, deploying landing gear, deploying or retracting speed brakes.

The variants of nominal conditions used in this study are: tail wind between $20,000ft$ and $12,000ft$ at either $30kts$ or $80kts$. Tail winds impact the aircraft descent by making it shallower, thus making it more difficult to follow the prescribed vertical path without over speeding. When the aircraft detects that its current configuration will deviate too significantly ($< 10kts$) from the prescribed speed, it will request the pilot to apply drag. A request for additional drag triggers the pilot to apply speed brakes until the speed deviation decreases sufficiently($< 5kts$). Depending on the function allocation, in addition to using speed brakes to slower down the aircraft, the automation or the pilot also attempts to minimize the vertical deviation from the prescribed flight path. Maintaining the vertical flight path is important in congested airspace because other flight corridors pass very close to the one that the aircraft is traversing and transgressing into as it increases the risk of collision.

WORK MODELS THAT COMPUTE (WMC)

This scenario is simulated using the WMC framework developed by the Cognitive Engineering Center at the Georgia Institute of Technology to address several concerns in the design of multi-agent socio-technical systems, particularly addressing the impact on function allocation on human automation interaction. The WMC simulation framework extends models used in qualitative Cognitive Work Analysis (CWA) [15] to a form suitable for computational simulation. WMC seeks to model systems of interest using a consistent, systematic perspective of the collective work performed by a team of agents. The constructs of the framework represent *actions* describing work processes, *resources* describing the state of

States Relevant to the Action	Actions of the Pilots	Cognitive Control Mode		
		Opportunistic	Tactical	Strategic
States of Aircraft				
Configuration	Confirm Configuration Change		Periodically	Anticipated
Position	Monitor Altitude	As Required	Periodically	Anticipated
	Monitor Vertical Deviation		Periodically	Anticipated
	Monitor Distance to Waypoint		Periodically	Anticipated
	Verify TOD Location			Anticipated
	Verify Crossing Restriction			Anticipated
	Monitor Green Arc		Periodically	Anticipated
	Confirm Target Altitude		Periodically	Anticipated
	Confirm Target Airspeed		Periodically	Anticipated
Direction	Monitor Heading Trends		Periodically	Anticipated
	Monitor Waypoint Progress		Periodically	Anticipated
	Confirm Active Waypoint		Periodically	Anticipated
Speed	Monitor Descent Airspeed	Airspeed	Periodically	Anticipated
	Reduce Airspeed for Late Descent			Anticipated
States of Environment				
Communication	Confirm Data Communication		Periodically	Anticipated
	Request Clearance			Anticipated

Table 3. Monitoring actions included within each cognitive control mode and their timing

the system, *decision actions* for strategy selection, and represent *work* as a collection of *actions* grouped into *functions* at multiple levels of abstractions. These actions are executed by *agents*. Strictly speaking WMC is a hybrid continuous-time, event-based simulator where specific timing constructs are included. A full description of WMC can be found at [13].

WMC allocates actions to *agents*. Agents execute and manage their actions included within each function. Different agents may execute different number of simultaneous actions. Additionally, human agents are known to control their behavior in differential ways. We have implemented three different types of behavior for human agents (strategic, tactical, and opportunistic) following the corresponding descriptions of cognitive control modes (CCM) given by [8] and summarized in Table 3.

The basis of the WMC simulation framework are work models that are hierarchical work description decomposition into atomic actions. The simulation engine scans through the work model and loads the actions into the WMC simulation engine's action list. The simulation engine orders actions by their next update time. The action with the nearest next update time is placed on the top of the list and executed by being passed to the appropriate agent. During execution, the action declares a new next update time and the action list is sorted. The process repeats with the action with the closest next update time being executed next. Thus the time step in WMC is not fixed, but is based on the needs of the actions. In this work model, the actions governing the aircraft dynamics are updated most frequently (~20Hz) and the pilot actions less frequently (~0.1Hz).

OUTPUTS GENERATED BY WMC

The CDA scenario has two primary agents: the pilot and the automated aircraft. Because of the centrality of actions in WMC simulations, unlike standard agent-based simulations, WMC simulation traces consist of a list of every action which is executed and the inputs passed to the action. Conversely most agent-based simulations only generate traces of the state variables and not the explicit mechanisms which produce them. WMC simulation traces consist of actions, agents and resources which are all linked to the time of execution. Thus, WMC simulation traces are suitable for analysis of both the system parameters and the agent actions. From now on, we will use the abbreviations SC*XX*FA*Y*ZZZ to stand for various different parameterized CDA scenarios that we handle. Here SC01 stands for everything nominal, SC31 and SC33 indicate $30kts$ and $80kts$ tail wind between 20,000ft and 12,000ft. FA*Y* will encode function allocation like FA1 or FA2. ZZZ can be STR, OPP or TAC indicating the pilot cognitive mode. The following line of trace describes an action performed by the pilot in the scenario SC33FA3OPP:

```
Time:296.969921 Action:monitorAltitude Agent: pilot Attribute: 3
Got resource CDAAC_altitude Value(s) 31000.00211 Got resource
flight_phase Value(s) 1 Got resource waypointtocompare Value(s)
TOD LAT:34.34575721 LON:-116.5349904 ALT:31000 SPD:350
NUT: 356.969921
```

A trace step starts with the time, the agent involved and the action that was executed by that agent. In the example above 297 seconds into the simulation, the pilot monitors the altitude. The attribute describes the kind of action that was performed. There are three types of actions for human agents: monitoring, teamwork, taskwork. Then for each resource that was either accessed, i.e. *gotten* or *set* by this action the value is recorded. Getting a resource means that the agent reads the value of a resource for a particular action. For example, CDAAC_altitude is the resource for the altitude of the plane which is 31,000 feet. In the real world this would correspond to the pilot checking the current altitude and storing this value (the resource value) in their short term memory. The last resource that is read is the next waypoint with its latitude, longitude, altitude and speed values. As this is a monitor action, no resources are actually set. The above monitoring action happens in the opportunistic CCM every 60 seconds (in other CCMs this is different). Accordingly the next update time (NUT) of the action is set to 60s (a relative time) and is displayed at the end of the trace.

TRACE ANALYSIS

We now present our experience of implementing our methodologies as described in Section FORMAL VERIFICATION for analyzing the traces generated by WMC for the various CDA scenarios as described in last section.

Desired Properties

A very important step in any verification procedure is identification of the important requirements of the system. In the CDA scenario, we need to verify whether the actions of the pilots, the air traffic controllers and the automation or aircraft dove-tail together to perform CDA to land the aircraft. Each aircraft is provided with a specific flight plan whose later part consists of the plan to perform CDA. These flight plans are essentially ordered collections of waypoints. These waypoints inform the pilots and the aircraft as to the path they need to follow to reach their destination. Associated with each waypoint is a collection of constraints, indicating among other things latitude, longitude, altitude and speed the aircraft should reach next. Although, due to environmental conditions like adverse tail wind, an aircraft may not always follow the path specified by waypoints precisely, they need to be within a certain safe threshold of deviation from the guidelines provided by the waypoints. In this work we will analyze whether simulated traces of an aircraft always conform to the waypoints on its journey to its destination. Thus one property that we want for the CDA scenarios is that: at any point in time, the altitude of the aircraft is within the range specified by the current and previous waypoint. Also, we want to verify that the altitude is monotonically decreasing. We also want to verify that the pilot adheres to the actions he needs to undertake at different waypoints as described in Table 2. For the sake of simplicity we focus only on the altitude aspect of waypoints in this work.

- **Effectiveness Property**: The aircraft should be able to ultimately descend to 150 feet. This property is actually a sanity check. $\Diamond(\mathsf{currAltitude} \leq 150)$. Here currAltitude indicates the altitude of the aircraft at the current point in time.
- **Safe Operation Property**: This property checks that the aircraft does not deviate from the path suggested by the given list of waypoints and also that the aircraft does not increase its altitude instead of decreasing as suggested by the waypoints. $\Box((\mathsf{currAltitude} \geq \mathsf{currWayPointAltitude}) \wedge (\mathsf{currAltitude} \leq \mathsf{prevWayPointAltitude}) \wedge (\mathsf{currAltitude} \leq \mathsf{prevAltitude}))$. We can even go further and combine this property with the effectiveness property to state that the aircraft should be able to descend properly if the aircraft always stays within the prescribed waypoints: $(\Box((\mathsf{currAltitude} \geq \mathsf{currWayPointAltitude}) \wedge (\mathsf{currAltitude} \leq \mathsf{prevWayPointAltitude}) \wedge (\mathsf{currAltitude} \leq \mathsf{prevAltitude})) \wedge \Diamond(\mathsf{currAltitude} \leq 150)$.
- **Pilot Behavior Properties** We can also assess conformance to recommended behavior of flying the aircraft to perform CDA by the pilot. For example we can perform the sanity check that eventually each and every waypoint is chosen by the pilot. The norm is that the pilot switches to the next waypoint in the flight plan when the distance to the current waypoint is less than 0.2 nautical mile. Let us use the proposition $\mathsf{WPSelected_i}$ to indicate selection of the i-th waypoint in the flight plan by the pilot. Then we can verify that $\Diamond\mathsf{WPSelected}_i$. Also we can verify that if a waypoint is selected then it is eventually reached by the aircraft: $\mathsf{WPSelected}_i \Rightarrow \Diamond(\mathsf{currAltitude} \leq \mathsf{waypointAlititude}_i)$. We can also check that the first waypoint gets chosen by the pilot and that remains the current waypoint under consideration until the next waypoint in the flight plan gets chosen. For the flight plan segment between the waypoints RIIVR and KRAIN, we can have a (simplified) property like: $\mathsf{currwaypoint} = \mathsf{RIIVR} \quad \mathcal{U}\,(\mathsf{currwaypoint} = \mathsf{DECEL} \quad \mathcal{U}\ (\mathsf{currwaypoint} = \mathsf{LUVYN} \quad \mathcal{U} \quad \mathsf{currwaypoint} = \mathsf{KRAIN}))$. Here currWayPoint stands for the currently chosen waypoint. However, even this property can be rigid in the sense that if the pilot misses one particular waypoint, we cannot analyze the system for the rest of the waypoints. So, a more flexible property can be used for each waypoint in the flight plan. Let us consider the waypoint RIIVR: $(\Diamond(\mathsf{currwaypoint} = \mathsf{RIIVR}))\wedge ((\mathsf{currwaypoint} = \mathsf{RIIVR}) \Rightarrow ((\mathsf{currwaypoint} = \mathsf{RIIVR})$

$\mathcal{U}$ (currwaypoint = DECEL))). We use $A \Rightarrow B$ to stand for A implying B: if A is true then so must B. This property states that eventually the waypoint RIIVR will be chosen by the pilot and that will remain the waypoint of choice until the next waypoint DECEL is chosen. This property will allow us to debug the system with respect to each segment delineated by the flight plan. This example property reasserts the importance of our work. One single simulation trace can be generated and formally encoded, then verified repeatedly against many different requirement specifications. This process even allows for adaptive development of requirement properties. Also we can analyze the system stepwise to unearth possible reasons underlying inconsistencies while attempting to characterize sources of error. Use of existing verification tools simplify this step for us. We can also check the pilot actions in the scenarios against their expected actions for different function allocation modes as described in Table 1. For example, when the aircraft is being maneuvered in mixed automated FA3 mode or mostly manual FA4 mode, the pilot should be updating vertical auto flight targets. However, in automated modes like FA1, the pilot should not be updating the targets manually: $(((\text{FAmode} = \text{FA3}) \vee (\text{FAmode} = \text{FA4})) \Rightarrow \Diamond(\text{dialedVSSelector})) \wedge (\neg((\text{FAmode} = \text{FA3}) \vee (\text{FAmode} = \text{FA4})) \Rightarrow \Box(\neg(\text{dialedVSSelector})))$.

Now that we have delineated the properties that each trace should maintain, we can move on to modeling the traces themselves. Our choice of software verification tool is the model checker SPIN(`www.spinroot.com`) [9]. The input language for SPIN is Promela. We first present the methodology for encoding the traces into Promela.

Model for WMC Traces

We first present our modeling technique where no contraction of the simulation traces are performed before encoding into formalisms. This will help illustrate the necessity of model reduction for trace analysis. The various CDA scenarios analyzed by the WMC simulation engine have three principle categories of entities. We present the encoding method for resources and actions in Promela:

- **Resources:** Each resource is essentially a container of information. Each resource is thus mapped to a suitable Promela variable. Since Promela is not equipped with floating point values, all floating point resources are truncated to be integers. The information contained in a resource is updated by a responsible agent. The information contained in a resource may also get transmitted to a requesting agent. Thus two dedicated channels are assigned to each resource for communication of the resource information. For example the current altitude of an aircraft is tracked using the resource CDAAC_altitude. This resource is declared as follows with an initial value of 31,000 feet:
```
int CDAAC_altitude = 31000 ;
channel CDAAC_altitudeCH [1] of { int };
channel CDAAC_altitudeRCH [1] of { int };
```
The first channel is dedicated to transmitting the value contained within the resource. For example, a pilot may request to see the current altitude of the aircraft. The second channel is dedicated to receiving new altitude value. For example, the aircraft automation may measure and set a new value of the altitude the aircraft is currently flying at.

- **Actions:** Each action either reads the existing value or writes a new value to a set of resources. Thus each action is represented as processes that update the relevant resources. For example, let us consider the action named: monitorWaypointProgress that is executed by the pilot. The pilot checks the distance to the current waypoint and sets several variables. Partial Promela encoding for this action pertaining to getting the distance to the current waypoint and setting the vertical deviation by the pilot is presented below.
```
int monitorWaypointProgressdistCurrentWaypoint = 0;
proctype monitorWaypointProgressGETSdistCurrentWaypoint(){
    distCurrentWaypointCH?monitorWaypointProgressdist-
    CurrentWaypoint ;}
proctype monitorWaypointProgressSETSverticaldeviation (
    int newval){verticaldeviationRCH!newval ; }
```
In this code, the first process monitorWaypointProgress-distCurrentWaypoint corresponds to obtaining the value of the distance from the current waypoint by the pilot. This process name is obtained by joining the name of the action monitorWaypointProgress and the resource name distCurrentWaypoint. The value obtained from reading a resource is stored in a variable owned by the corresponding action. The name of the variable is determined by concatenating the names of the action and the resource. In this case the value of the resource distCurrentWaypoint is stored in the variable named monitorWaypointProgress-distCurrentWaypoint.

Trace Model:

We have a translator module written in the scripting language python that translates a simulation trace into Promela. The translated encoding needs to have a preamble with the necessary setup of environment through type and variable declarations. The translator first determines the datatype that is used for each resource. The translator module automatically adds required codes for declaring appropriate datatypes for waypoints and flight plans. String type data is represented by enumerated datatypes.

The action executed by an agent in a WMC simulation step gets translated into a collection of relevant function calls for setting or obtaining values for resources. Let us consider an arbitrary line of trace from the scenario SC01FA1STR:

```
Time:0 Action:CDAAC_fly Agent: CDAAC Attribute: 0 Set resource
CDAAC_altitude Value(s) 31000 Set resource CDAAC_groundspeed
Value(s) 350 Set resource CDAAC_vspeed Value(s) -0 Set resource
CDAAC_heading Value(s) 237.8671593 Set resource CDAAC_latitude
Value(s) 34.605994 Set resource CDAAC_longitude Value(s) -
116.035139
```

The CDA executor automation part of the aircraft, namely the agent CDAAC, executes the action CDAAC_fly so that required parameters are set to let the aircraft perform a CDA landing. Thus several parameters like CDAAC_altitude and

CDAAC_latitude get set by this action. Let us consider parts of the corresponding Promela code:

```
run CDAAC_flySETSCDAAC_altitude(31000) ;
run receiveCDAAC_altitude() ;
```

The first line of code stands for the pilot setting the value of the resource CDAAAC_altitude to 31000. Then the corresponding receiving process for the resource is called so that the resource can be modified accordingly. Similar encodings are performed for all the resources modified by the action. For resources that get read by an agent, we add two process calls: one standing for the resource sending its data and the other symbolizing reading the data by the agent. For example,

```
run sendaltitude_clearance() ;
run manageWaypointProgressGETSaltitude_clearance() ;
```

The first process call stands for the resource altitude_clearance sending out its value over the send channel dedicated for itself. The second process call encodes capturing the data sent by the resource altitude_clearance by the action manageWaypointProgress.

Each and every line of trace is translated in this same manner. The resultant Promela module can now be verified against properties as described earlier. However, at this point, we found that the Promela encodings for the CDA traces were sufficiently large to cause SPIN to suffer from segmentation fault. For example, for the SC01FA1STR scenario, the Promela file generated had 1,274,385 lines of code. The trace file for this scenario had 125,832 lines of trace. Thus there is an almost ten-fold size inflation introduced by the translation procedure. Additionally, at any point in time SPIN can only handle 255 processes. Each resource in our translation requires two process definitions. Each action requires process definitions for each resource it handles. Our translation was stretching SPIN to its limits through the numerous process definitions. CDA traces contain lots of non-integral simulation variables. Their range of possible values are large and contain negative values. Thus they cannot be represented meaningfully using the single byte Promela datatype *byte*. They need to be, at the least, represented by variables of the two byte long integer datatype *short*. Thus the size of memory required to store each individual state grew significantly. Also, we need to store historical snapshots of various resources. Each variable introduced to capture historical values of a resource adds to the size of each state. Thus if we translate the entire trace file faithfully by considering each of the eighty-seven resource updates and transmission events then the generated file becomes quite untenable for SPIN. To date, we have attempted complete analysis with SAL (`http://sal.csl.sri.com/`) and SPIN. Both tools were incapable of handling such large files resulting in segmentation faults with the computing resources that are at our disposal. The reason behind this is that model checkers, irrespective of being explicit or implicit (symbolic) state model checkers, need to perform state space computation. The size of the traces combined with all the different parameters associated with operating an aircraft and the all the different values they can take on renders the resultant state space too large to compute, store and keep track of. Space and memory limitations provided to these tools can quickly restrict their computing prowess while processing large state spaces.

Abstractions for Trace Model

We now present the abstractions we made to obtain tractable Promela models for SPIN.

- **Lateral Contraction via Parameter selection:** There are numerous parameters present in the traces. However, we are focusing on verifying waypoint conformance in this work. Each waypoint is a five part data: (a) name, (b) altitude, (c) longitude, (d) latitude, and (e) speed of waypoint. Hence we abstracted away all the parameters of the scenario that did not refer to the altitude, latitude, longitude or speed of the aircraft. Hence we were left with the flight plan, the current waypoint, the current altitude, current latitude, current longitude and current speed parameters. This reduction from eighty seven to six parameters tremendously helped in reduction of the computational time. We however needed to add a history variable for each of these parameters so that we could assert properties about changes in the information contained in them.
- **Vertical Contraction via Constraining to Relevant Trace Slice:** The simulated traces tracks the journey of the aircraft through thirteen waypoints. Although ideally we would like to analyze the entire CDA flight plan, in this work we focused on a representative segment of the flight plan. We focused on the behavior of the aircraft under influence of the pilots and air traffic controllers between the waypoints RIIVR and KRAIN. This seemingly small segment was sufficient to lead to the discovery of interesting issues.
- **Data Abstraction by Discretization of Continuous Value Parameters:** Many aviation scenario parameters are continuously valued like time, latitude, longitude, altitude etc. As mentioned before, we discretized these parameters to be integers. Additionally, we scaled some parameters to be represented by 8-bits of data. For example, the speed of the aircraft ranges between 350 and 150. As a result we discretized the speed by scaling it to a value between 0 and 255.

Additionally at every timestamp we decided to deal with only setting the value of a resource by an agent as getting the value of a resource does not cause any status change. Currently for the same SC01FA1STR scenario, the size of the Promela file is 45,420 lines of code. Compared with the 1,274,385 lines of Promela code that we used to have in the beginning, we have now obtained around 96% reduction in Promela code size. Omission of the latitude, longitude and speed resources further reduced the Promela code size to 18,469 lines.

Verification Results and Discussion

We now present our findings along with the time for computing them in Table 4. The verification was performed by a MacBook Pro laptop equipped with 2.5 GHz Intel microprocessor using 8 GB 1333 MHz DDR3 memory. Our analysis results tell us that hypothetical expectations about simulation

Scenario (Function allocation/ CCM)	Waypoint KRAIN reached?	Rigid waypoint conformance along path?	Flexible waypoint conformance along path?	Altitude monotonically decreases?	Altitude decreases to cleared altitude level only?	All four waypoints chosen by pilot?	Pilot manually adjusts flight targets?	Time (seconds)
Sc01FA1Str	Yes	No	Yes	No	Yes	Yes	No	260.86
Sc01FA2Str	Yes	No	Yes	No	Yes	Yes	No	260.30
Sc01FA4Str	Yes	No	Yes	No	Yes	Yes	Yes	369.11
Sc31FA2Str	Yes	No	Yes	No	Yes	Yes	No	216.73
Sc31FA3Str	Yes	No	Yes	No	Yes	Yes	Yes	200.89
Sc31FA4Str	Yes	No	Yes	No	Yes	Yes	Yes	201.21
Sc33FA3Opp	Yes	No	Yes	No	Yes	No	Yes	235.46
Sc33FA3Str	Yes	No	Yes	No	Yes	Yes	Yes	208.49
Sc33FA4Opp	Yes	No	Yes	No	Yes	No	No	69.00
Sc33FA4Str	Yes	No	Yes	No	Yes	No	No	81.97

Table 4. Analysis of CDA traces generated with various different environment conditions

traces may not always be met by real simulation traces. In actual operation of the aircraft, the aircraft may deviate from the desired path without violating safety of other aircraft. For example, the aircraft is always expected to fly on the path defined by two adjacent waypoints: the current and the previous waypoint. Usually the pilot observes that the aircraft is within 0.2nm of the current waypoint and updates the waypoint to the next one. However, the aircraft still needs to descend to the altitude suggested by the previous waypoint. Thus there is a period of time where the aircraft is supposed to be attempting to reach the current waypoint but in reality is still working on achieving the previous waypoint. This observation led to a relaxed property stating that the aircraft always adheres to at least the previous waypoint. Another property we checked was whether the aircraft always monotonically decreases its altitude. However, this property does not hold in most of the variations. In SC01FA1STR, the aircraft had overshot the desired altitude for waypoint LUVYN by almost a hundred feet, then it had increased its altitude to achieve the desired altitude, and even then overshot the required altitude by almost 40 feet. This led the domain experts to offer two explanations: (a) the simulation module encoding the autopilot is flawed and needs to be modified to eradicate this atypical and possibly impractical behavior, (b) the property needs to be relaxed to include possibility of pilot operating the aircraft within a safe threshold of the path suggested by the waypoints. We relaxed the property to check that at the very least, when a pilot is overshooting a desired altitude, they do not overshoot by an unsafe altitude difference and also while it tries to correct the altitude it does not increase the altitude by an unsafe amount. However, the experts agree that the write-up of the simulation module needs to be inspected. We verified some pilot behavior expectations like: whether he adheres to all four waypoints in the flight plan, whether he attempts to manually configure the aircraft while in different function allocation modes. We found that in the opportunistic mode, the simulated pilots do not monitor waypoint progress at all, suggesting infidelity in the simulation modules. Also we found that between RIIVR and KRAIN waypoints, in the fully automated mode, the pilot does not manually adjust all flight parameters as presented in Table 2.

CONCLUSION

Safety-critical systems often tend to be continuously evolving infinite state systems. These systems need to be highly reliable and fault-tolerant. Simulation of such systems yields voluminous traces for a *single* behavior trajectory at a time. Formal verification can help ascertain possible deficiencies during the design and development process of safety-critical systems. Formal system verification however can only be performed with suitable abstractions as they need to consider *all* possible behavior trajectories in order to make decisions about desired property conformance. In order to find a middle ground, we suggest obtaining simulation traces for interesting, representative sets of environment conditions and formally analyzing each simulated trace for requirement conformation. This methodology allows for asserting succinct temporal formulae over the traces. Our implementation of the methodology for an aviation scenario demonstrates that with the help of LTL model checking tools, LTL property verification of traces can offer valuable insights about automated system behaviors. We were able to assess conformance of safety-critical system simulation traces to safety properties. Deviations of simulation traces from expected behaviors were judged to be caused by stringent desired properties in some and caused by possible infidelity in the simulation tool in some other cases. The LTL model checkers provide counter examples to facilitate determining the trace step sequence that led to violation of a desired property. Since a single simulation trace model can be checked against many properties by LTL model checkers, we can analyze one single simulation trace from the perspective of many different requirements. This provides a significant advantage over runtime verification via monitoring where a trace generation procedure can only be monitored against one single property. Violation of that particular property requires rerun of the simulation tool to assess the same trace against any other property. In future we intend to augment this methodology by implementing an LTL model checking algorithm suitable for finite simulation trace behaviors that will assimilate the expressive behavior of LTL properties, yet will operate on concrete trace elements instead of formal model of the traces for state space exploration. This method will hopefully eradicate the computational burden of traditional model checkers. We also intend to analyze simulation traces via offline runtime trace monitoring using the JavaMOP runtime monitoring framework [3] in future. This would involve translation of the simulation traces into Java modules, training of the monitoring as to events of interest in the simulation traces, expressing the desired LTL properties in terms of the events of interest, and then performing LTL property monitoring using the JavaMOP tool. This approach will also retain the benefit of keeping the verification modules separate from the simulation modules.

ACKNOWLEDGMENTS

This work was funded in part by NASA Contract number NNA10DE79C and NSF Award number 0917218. The content is solely the responsibility of the authors and does not necessarily represent the official views of the NASA and NSF.

REFERENCES

1. Proceedings of Runtime Verification Conferences: RV'01 to RV'10.
2. Bolton, M. L., and Bass, E. J. Evaluating human-automation interaction using task analytic behavior models, strategic knowledge-based erroneous human behavior generation, and model checking. In *IEEE International Conference on Systems, Man and Cybernetics (SMC), Alaska, USA* (2011), 1788–1794.
3. Chen, F., and Rosu, G. Java-MOP: A Monitoring Oriented Programming Environment for Java. In *TACAS*, N. Halbwachs and L. D. Zuck, Eds., vol. 3440 of *Lecture Notes in Computer Science*, Springer (2005), 546–550.
4. Chen, X., Hsieh, H., Balarin, F., and Watanabe, Y. Automatic Generation of Simulation Monitors from Quantitative Constraint Formula. In *DATE*, IEEE Computer Society (2003), 11174–11175.
5. Chen, X., Hsieh, H., Balarin, F., and Watanabe, Y. Automatic trace analysis for logic of constraints. In *DAC*, ACM (2003), 460–465.
6. Clarke, J.-P., Brown, J., Elmer, K., Ho, N., Ren, L., Tong, K.-O., and Wat, J. Continuous descent approach: Design and flight test for Louisville International Airport. *Journal of Aircraft 41*, 5 (2004), 1054–1066.
7. Girard, A., and Pappas, G. J. Verification using simulation. In *HSCC*, J. P. Hespanha and A. Tiwari, Eds., vol. 3927 of *Lecture Notes in Computer Science*, Springer (2006), 272–286.
8. Hollnagel, E. *Human reliability analysis: Context and control*. Academic Press, London, UK, 1993.
9. Holzmann, G. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.
10. Kemper, P., and Tepper, C. Automated trace analysis of discrete-event system models. *IEEE Trans. Software Eng. 35*, 2 (2009), 195–208.
11. Narayanan, S., and McIlraith, S. A. Simulation, verification and automated composition of web services. In *WWW* (2002), 77–88.
12. Pnueli, A. The temporal logic of programs. In *FOCS* (1977), 46–57.
13. Pritchett, A. R., Feigh, K. M., Kim, S. Y., and Kannan, S. Work models that compute to support the design of multi-agent socio-technical systems. *IEEE Transactions on System Man and Cybernetics.* (2011).
14. Stuart, D. A., Brockmeyer, M., Mok, A. K., and Jahanian, F. Simulation-verification: Biting at the state explosion problem. *IEEE Trans. Softw. Eng. 27* (July 2001), 599–617.
15. Vicente, K. J. *Cognitive Work Analysis: Toward Safe, Productive, and Healthy Computer-Based Work*. Lawrence Earlbaum Associates, Mahwah, N.J., 1999.
16. Yasmeen, A., and Gunter, E. Robustness for Protection Envelopes with Respect to Human Task Variation. In *IEEE International Conference on Systems, Man and Cybernetics (SMC), Alaska, USA* (2011).