Electronic Communications of the EASST
Volume 53 (2012)

Proceedings of the 12th International Workshop on Automated Verification of Critical Systems (AVoCS 2012)

Formal Specification and Verification of a Coordination Protocol for an Automated Air Traffic Control System

Yang Zhao Kristin Yvonne Rozier

15 pages

Guest Editors: Gerald Lüttgen, Stephan Merz
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Formal Specification and Verification of a Coordination Protocol for an Automated Air Traffic Control System

Yang Zhao[1] * **Kristin Yvonne Rozier** [2]

[1] University of California, Riverside, USA. zhaoy@cs.ucr.edu
[2] NASA Ames Research Center, Moffett Field CA, 94035, USA. Kristin.Y.Rozier@nasa.gov

Abstract: Safe separation between aircraft is the primary consideration in air traffic control. To achieve the required level of assurance for this safety-critical application, the Automated Airspace Concept (AAC) proposes three levels of conflict detection and resolution. Recently, a high-level operational concept was proposed to define the cooperation between components in the AAC. However, the proposed coordination protocol has not been formally studied. We use formal verification techniques to ensure there are no potentially catastrophic design flaws remaining in the AAC design before the next stage of production.

We formalize the high-level operational concept, which was previously described only in natural language, in NuSMV and perform *model validation* by checking against LTL/CTL specifications we derive from the system description. We write LTL specifications describing safe system operations and use model checking for *system verification*. We employ *specification debugging* to ensure correctness of both sets of formal specifications and *model abstraction* to reduce model checking time and enable fast, design-time checking. We analyze two counterexamples revealing unexpected emergent behaviors in the operational concept that triggered design changes by system engineers to meet safety standards. Our experience report illuminates the application of formal methods in real safety-critical system development by detailing a complete end-to-end design-time verification process including all models and specifications.

Keywords: model validation, specification debugging, model checking, safety-critical system

1 Introduction

Safe distance between commercial aircraft must always be guaranteed to prevent potential aircraft collisions. The Automated Airspace Concept (AAC) [13] is designated to ensure the safe separation of these aircraft within a given airspace sector. The AAC is a high-level generic framework proposed as a candidate for the Next Generation Air Traffic Control System (NextGen), which is currently under development at NASA. To mitigate the complexity of calculating aircraft trajectory resolutions and enable separation assurance in real time, the AAC divides the task of separation assurance between three independent resolution systems that handle long-term, near-term, and urgent predicted loss of separation between aircraft, in addition to juggling

* Work contributing to this paper was supported in part by NASA's Airspace Systems Program and by the National Science Foundation under Grant CCF-1018057.

other tasks, such as fielding trajectory requests from controllers and pilots. It is essential that we verify the interactions between the set of independent components of the AAC result in emergent total-system behaviors that are safe, and that pilots reliably receive the correct routes from the correct components.

Formal verification has been gradually accepted as an important step in the design flow of safety-critical systems. Model checking is one of the most widely used formal verification methods. We use the model checker NuSMV [8, 9] since it is well-documented [27], open-source[1], and frequently used in industry [7,16,20,22,28,32]. We need to build a *system model* of the operational concept [12–14, 24] in formal semantics and also describe the behaviors required for safe system operation, the *model verification specification*, using temporal logic. NuSMV is then able to prove that the system model conforms to the model verification specification or return counterexamples corresponding to possible executions in the operational concept that violate that specification. Previously, significant effort has been devoted to increasing the automation and efficiency of the model checking step. However, formalizing safety-critical systems that are not inherently built on formal semantics becomes a bottleneck. We argue that system model validation and specification debugging are necessary tools to build confidence when formalizing a high-level conceptual design, because inconsistencies between the model and the real system and erroneous specifications can lead to spurious verification results.

We illuminate a complete end-to-end design-time verification process on the full-scale AAC coordination protocol, presenting our work on formally specifying, validating, and model checking from three aspects. The first aspect is the *formalization methodology*. We detail a process for deriving a formal model of the AAC operational procedure building upon existing literature and input from human experts via about 100 person-hours of interviews with system architects. We extract *model validation specifications* from the operational concept and construct *model verification specifications* by codifying the expectations of system designers, extracted via conversation, in temporal logic; all models and specifications are available online. [2]

The second aspect is *model and specification debugging*, which addresses a practical challenge in the above model checking procedure: how to ensure that the formalization of the system model correctly reflects the designer's intentions. Counterexamples returned from the model checker may not reveal problems in the operational concept itself, but instead correspond to imprecisions in the system model. Our solution to this problem is to create both the system model and a set of temporal logic properties (our model validation specification) from the description of operational concept as two ways of formalizing the designer's intentions. We validate our system model by model checking this set of specifications, which directly correspond to statements in the designer's description of how the system is constructed. Note that this is distinct from verification, which involves model checking the system model against specifications encoding the system safety requirements that describe how the system should behave. The reported counterexamples pinpoint any inconsistencies between these two and we manually identify whether the problem lies in the model or the specification. We exemplify our process of specification debugging, including LTL satisfiability checking [29] and counterexample-guided specification

[1] The source code and documentation are available for download from: http://nusmv.irst.itc.it/.

[2] Both NuSMV models, all specifications, and the commands we ran to check them are available at http://ti.arc.nasa.gov/m/profile/kyrozier/AAC/AAC.html.

debugging, to increase confidence in the correctness of our specifications. This iterative process greatly increases our confidence in the consistency between the system model, the formal specifications for both model validation and model verification, and the designer's intention.

The third aspect, *model checking*, completes our system verification process by checking the system model against the model verification specifications. We detail two emergent unexpected behaviors of the AAC that were pinpointed by counterexamples and discuss possible modifications to mitigate them. System designers identified these two cases as particularly illuminating and changed the system design in response to our counterexamples. We also discuss the lessons resulting from our experience and provide insights that are instructive for the formal specification, validation, and verification via model checking of other industrial systems.

The remainder of this paper is organized as follows. We highlight related work in Section 2. We introduce the full AAC operational concept and the merits of formalizing it in Section 3. In Section 4, we detail our system modeling techniques and their effects on the verification process. Section 5 discusses our specification debugging techniques. Section 6 presents important counterexamples describing unexpected emergent behaviors along with modifications to the system design to combat them. Section 7 concludes and points out future work.

2 Related Work

The three AAC components that calculate resolution maneuvers have undergone verification through simulation as well as ongoing component-level verification utilizing formal methods [5, 15, 25]. However, there is an important verification aspect that has never been addressed: the cooperation between components. Verification efforts to date have concentrated on single components in the AAC; each component has been rigorously evaluated individually under the assumption that if each component of the system behaves correctly that the system as a whole will also function as desired. Employing simulation or testing on the AAC as a whole is formidable, and too time-consuming to provide real-time results. Evaluating different potential system configurations, as we do here, was not previously possible.

Although [23] pointed out weaknesses of model checkers in handling complex data structures and arithmetic operations, model checking is still a powerful tool for verifying safety-critical protocols and several case studies on model checking various aerospace systems have been published [6, 21, 31]. Our study is carried out early in the design phase before any system implementation is available. Our system model and specifications for model validation and model verification are generated directly from natural language, while [31] generates the system model from the source code, and [6, 21] generate NuSMV models automatically from another formal language.

Specification debugging has drawn much attention [2, 17, 26]. These methods analyze the *consistency*, *completeness* and *safety* of formal specifications. For example, [2, 26] presented a platform for formal analysis of hardware requirements called "RAT," which has been applied in the analysis of aerospace systems [4]. Using this framework, instead of creating a system model, we only need to generate the model validation specification in Section 4.2 as the formal specification of the operational concept; RAT is able to check this specification against our model verification specification. However, the consistency between the formal specification under verification and the designer's intention is not guaranteed within this framework. Moreover, our

work is able to create a validated system model that can be used as a prototype for the future system implementation.

We demonstrate the simultaneous creation of the system model and model validation specifications; our approach is to validate these two parts against each other using model checking. Previous work [5, 6, 23, 31] often requires one or both of these two parts as the input to do analysis and verification. The chicken-and-egg relationship between the system model and the formal specifications describing system requirements provides a challenge when employing formal methods at design-time. We present the process of formalizing both the model validation and model verification specifications, like [21], and, as a highlight, we also present our specification debugging scheme, including LTL satisfiability checking. Little information about the effectiveness of LTL satisfiability checking in practical model checking is available in previous publications. We are the first to publish the complete details of such an end-to-end verification process, including the code for NuSMV models and specifications. Our models are open to the public for future research to help shed some light on the problem of initial, design-time formalization.

3 The Automated Airspace Concept

The central task of the AAC is to maintain safe separation and provide collision avoidance in the airspace. The AAC is able to detect a potential loss of separation (LOS) in the future, referred to as a *conflict*, and resolve conflicts by generating resolution maneuvers for the aircraft involved and sending these resolutions securely to pilot(s). Pilots are expected to carry out resolutions in a timely manner. To simplify communication and coordination, it is desirable to implement a single system capable of detecting and resolving all possible conflicts and collisions. However, the complexity of this system would be formidable and satisfactory response times are not achievable with currently available hardware. Thus, the AAC incorporates a compositional design, in which different components are responsible for handling short-, and near-term conflicts, and collision avoidance. Figure 1 illustrates the infrastructure of the AAC.

The strategic separation layer, referred to as the *AutoResolver*, addresses conflicts from three to 20 minutes in the future. The AutoResolver is implemented in software running on a central computer on the ground, taking in the trajectory of each aircraft in the airspace, detecting any conflicts, and outputting resolution maneuvers for any aircraft involved in such conflicts. There is no direct data link between the AutoResolver and the aircraft; the controller is in charge of

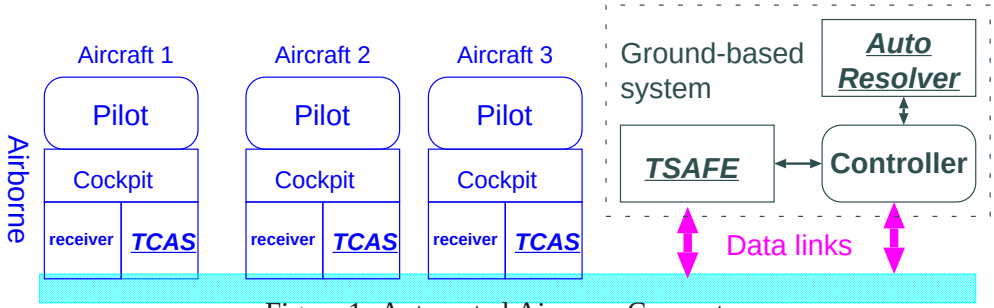

Figure 1: Automated Airspace Concept

sending the resolutions generated from the AutoResolver.

The tactical separation layer, known as the *Tactical Separation Assured Flight Environment* (*TSAFE*), addresses conflicts projected to occur less than 3 minutes in the future. TSAFE is also implemented in software running on a ground computer, but using a different algorithm from the AutoResolver. TSAFE can automate conflict resolution on the tactical layer, allowing the controller to focus more on strategic operation.

Finally, the *Traffic Alert and Collision Avoidance System (TCAS)*, is required by the Federal Aviation Administration mandate to address possible collisions less than 30 seconds in the future. TCAS software runs on each aircraft's on-board computer and detects possible collisions using a transponder installed in the aircraft. Thus, TCAS is totally independent from on-ground systems.

The *operational concept* of the AAC proposed in [13] specifies the authority and responsibilities of the above three systems, controllers and pilots. Figure 2 shows the responsibilities of the controller and TSAFE for each aircraft that is involved in a conflict. Since conflict detection is limited by the precision of route prediction, conflicts projected more than 20 minutes out are not considered. The AutoResolver detects long-term conflicts, up to 20 minutes in the future, corresponding to time slot (1), and also provides resolutions accordingly to the controller. If approved by the controller, the resolutions from the AutoResolver will be transmitted to the affected aircraft. TSAFE detects conflicts up to 3 minutes in the future. If the time to LOS is between 1 and 3 minutes, corresponding to time slot (2) in Figure 2, TSAFE will first alert the controller and wait for approval. In this circumstance, the controller has three choices: approve the resolution from TSAFE and give control responsibility for the involved aircraft to TSAFE, resolve the conflict manually, or wait without resolving the conflict. In the latter two cases, the controller maintains responsibility for controlling the aircraft involved in the conflict. If the controller transfers control to TSAFE, he should not give resolutions to the involved aircraft until the conflict has been resolved. However, if the time to LOS falls below the TSAFE *threshold* of 1 minute, as defined in [13], TSAFE will take control of the aircraft involved in the conflict from the controller, without having to wait for controller approval, and send resolutions to these aircraft automatically. This case corresponds to time slot (3) in Figure 2. After the conflict is resolved, TSAFE will return control of the aircraft involved to the controller, as shown in time slot (4) in Figure 2. Without the help of ground-based systems, TCAS is able to detect possible collisions that are projected in under 30 seconds and give resolutions, constituting the last layer of protection against collisions. TCAS does not wait for any approvals from the ground or notify other systems, but directly advises the pilot.

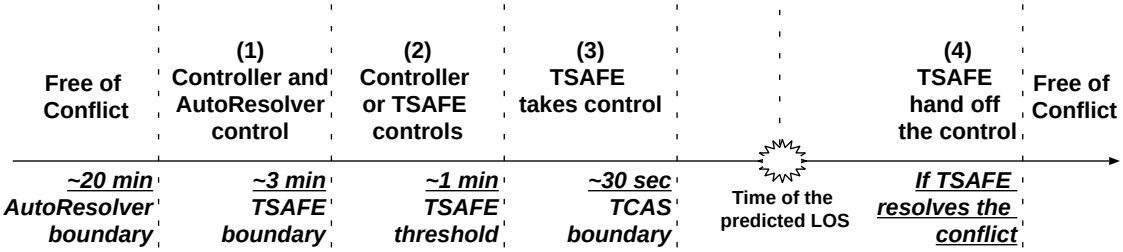

Figure 2: Operational concept for each aircraft

The operational concept defines a total order on resolution priority for pilots: TCAS resolution > TSAFE resolution > controller (with the AutoResolver) resolution. Pilots are required to execute the resolution maneuvers following this priority list if more than one resolution is received.

4 Formalization methodology

The system model and formal specification (for both validation and verification) are two sides of our formalization. To achieve efficient, design-time feedback using model checking, we also consider an abstract model, which conserves all possible execution paths in the original model. We provide specifications and report model checking results for both models.

4.1 Model of the AAC

Our model assumes the following: (1) All conflicts will eventually trigger AutoResolver, TSAFE, and/or TCAS alerts, i.e. while we do not assume completeness for any one component, we assume there is no conflict that will go undetected by all three components. (2) TCAS, TSAFE, and the AutoResolver have been independently verified and therefore always give correct resolutions ("correct resolution assumption"). As long as the resolution advisory is executed by the pilot, the detected conflict will be correctly resolved. (3) We assume humans are *competent operators*. A controller will never give an incorrect resolution or fail to give a resolution when one is expected. Pilots are aware of the AAC component systems and the priority of different resolutions received and are always able to respond correctly, in the required time frame. (4) There are no hardware failures or lost messages. We do not consider hardware failure probabilities and human error models in the system model. Since the AAC components have been individually verified and the humans-in-the-loop rigorously trained, we do not consider cases where automated algorithms and humans simultaneously behave incorrectly, but instead focus on verifying the protocol for transmission and execution defined in the operational concept.

As shown in Figure 3, the AAC is modularized into four components: *Aircraft*, *TSAFE*, *Controller*, and the *Environment*. Arrows show the important input/output variables interconnecting the modules. Note that there could be several instances of the aircraft module, $A1, A2, \cdots$, in our model; we picture a single instance to simplify the figure. The resolution transmissions between TSAFE, the controller, and aircraft are abstracted as the Boolean variables "TSAFE_command" and "CTR_command" with suffixes _1/_2/_3 to indicate the destination aircraft for these resolutions. We omit the suffixes when they are not necessary. For example, variable "TSAFE_command_1" is an output of the *TSAFE* module and an input to aircraft *A*1. If TSAFE transmits a resolution to aircraft *A*1, TSAFE_command_1 is set to **1**, and *Aircraft* 1 module will react accordingly.

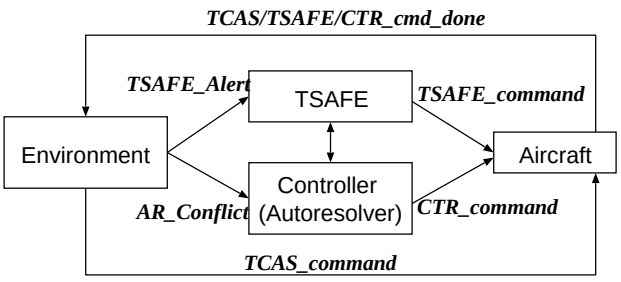

Figure 3: Interconnection between modules

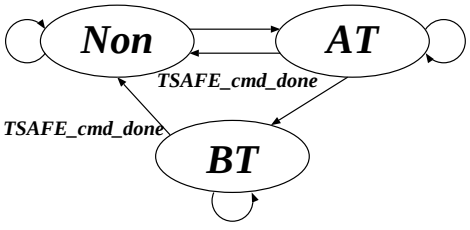

Figure 4: FSM for variable TSAFE_Alert

In the real system, the inputs to TSAFE and the AutoResolver are the aircraft trajectories and flight plans. We can abstract these inputs using the variables AR_Conflict and TSAFE_Alert to indicate "whether there is an AutoResolver/TSAFE alert." While AR_Conflict is a Boolean variable, there are three possible values for the variable TSAFE_Alert: Non, AT and BT, corresponding to no LOS detected, LOS detected with time to LOS above and below the threshold. Since TSAFE and the AutoResolver construct pairwise conflict lists, there is such a variable for each pair of aircraft with different suffixes. For example, "TSAFE_Alert_12=BT" means there is a LOS within the threshold time between aircraft $A1$ and $A2$. These variables are maintained in the *Environment* module, which is not a part of the AAC, and mimics all possible conditions of conflicts in the real system. After the *Aircraft* module sets "TCAS/TSAFE/CTR_cmd_done" flag, meaning a corresponding resolution has been executed by the pilot, the variables regarding the resolved conflict are reset.

Figure 4 is the finite state machine (FSM) for variable TSAFE_Alert, illustrating transitions between the three possible values. When TSAFE_cmd_done = **1**, meaning aircraft involved in the TSAFE alert have executed the resolution, TSAFE_Alert is reset to "Non." All the other non-self-loop transitions reflect possible effect of time elapsing, either a LOS occurs or time to LOS falls below the threshold. Similar FSMs are built for variables AR_Alert and TCAS_command to mimic the conflict detected by the AutoResolver and TCAS, and the only difference is that these variables only have true or false values. The cross-product of these FSMs constitutes the possible conflict conditions and transitions between them. In our model, we discretize the real-time execution of the system into sequences of transitions between possible configurations of the AAC system and conditions of the environment.

The *Aircraft* module models the TCAS system and the behaviors of the pilot. All aircraft in compliance with the AAC can be modeled as homogeneous aircraft modules. Each module maintains a priority list of received but not executed resolutions based on the priority TCAS resolution $>$ TSAFE resolution $>$ controller's resolution, and each pilot can execute only one resolution in each time frame. At one time, there can only be one TCAS/TSAFE/controller resolution active, so this list is always finite with a maximum length of three. Due to their resolution algorithms, the AutoResolver resolves the one most imminent pairwise conflict at a time, while TSAFE resolves all detected pairwise conflicts simultaneously. To simplify the logic, we assume the same pair of aircraft cannot appear on both the AutoResolver and TSAFE conflict lists at the same time. If this happens, we can safely remove the conflict from the AutoResolver conflict list. Based on our correct resolution assumption, a resolution cannot cause a conflict more imminent than the one it is resolving, when a resolution is executed, the sets of aircraft in conflict leave the conflict list. If the conflict detected by the AutoResolver and that detected by TSAFE involve two disjoint sets of aircraft, this is equivalent to resolving only the AutoResolver conflict and only the TSAFE conflict separately and independently. We must also account for the cases where the same aircraft is involved in both TSAFE and AutoResolver conflicts. To avoid the state space explosion problem, we need to build the AAC model containing the minimum sufficient number of aircraft instances. We agree with system designers that it is possible to reason about all meaningful corner cases of the real system utilizing *three* aircraft.

The *TSAFE* module models the behaviors of TSAFE with its dedicated data link. For a TSAFE conflict between aircraft $A1$ and $A2$ (TSAFE_Alert_12 = AT or BT), the TSAFE algorithm may generate a resolution for only $A1$ (TSAFE_command_1$=1$), only $A2$ (TSAFE_command_2$=1$),

or both (TSAFE_command_1= 1 ∧ TSAFE_command_2= 1); in our model one of these three cases is chosen nondeterministically. The AutoResolver does not generate cooperative resolutions, so we nondeterministically generate a resolution for one of the two aircraft involved in a conflict. Since AutoResolver resolutions must be sent by a controller, we integrate its functionality into the *Controller* module. When TSAFE or the AutoResolver alerts arise in the *Environment* module, these two modules react as defined in the operational concept and output resolutions for the involved aircraft. Our full-scale model contains 97 Boolean variables, including the variables indicating conditions of conflicts, TCAS/TSAFE/AutoResolver(controller's) resolutions, resolutions received by aircraft and hand-shaking signals between modules in the communication.

4.2 Specifications

We write two sets of specifications: model validation specifications to ensure the model follows the system design description, and verification specifications capture the emergent behaviors that should be avoided in the AAC system. Our specification sets are expressed using invariant declarations and LTL and CTL formulas. For the most part, we use LTL formulas, which more aptly describe the temporal behaviors for all paths; LTL most intuitively captures all of our system requirements and most of our checks of the system model design. We use CTL formulas to check that distinct behaviors occur on distinct paths at the same time, which is valuable for checking that we have modeled multiple disparate paths when validating our system model design. Table 1 lists the specifications we verified. Formulas are identified with IDs in the first column, where "LTL," "CTL," and "INV" distinguish LTL, CTL, and pure invariant formulas respectively. The specifications coalesce properties from two sources: the properties for model validation are from [13], and the properties for verification are generated from discussions with AAC designers. For model validation properties, which should correctly reflect the intentions of the designers, we quote the original sentences from [13] in the second column to demonstrate how we converted the natural language system design into formal temporal logic. Verification properties are primarily generated from the discussion with the designers, and we provide brief descriptions in the second column. Note that the actual formulas used in both model validation and verification (available online) are more complex than those listed in Table 1; for readability, we omit some details and only show the idea behind each formula.

4.3 Model abstraction

Our full-scale model takes more than 9 hours on an Intel Xeon 2.53GHz workstation with 36GB RAM to verify all properties in Table 1. Considering that we will need to modify the model and rerun the verification as the structure of the AAC or its operational concept evolve, this run time is not satisfactory. Faster response times are required to be useful in design time as system designers check proposed AAC modifications. Therefore, we employ a simple model abstraction [10] to speed up the verification process. Constructing both models required approximately three man months.

We abstract out the transition relation for variables related to the three aircraft and instead assign them randomly in each cycle. In NuSMV, this abstraction simply removes all statements under the keywords "TRANS" and "ASSIGN" in the *Aircraft* module. The set of possible executions in the abstract model is a strict super-set of the possible executions in the original model. Since the abstract model has the same set of variables in the concrete model, the map between

ID	Model Validation Specifications from [13]	Formula	Abs.	Cpl.
LTL-1	"With TSAFE alert, if the time to LOS in TSAFE alert exceeds a threshold, controllers retain responsibility for formulating resolutions and taking action to resolve conflicts. "	G((CTR_control & X TSAFE_Alert $\neq$ Non) $\rightarrow$ X CTR_control))	T	T
INV-1	"The next step is to send the TSAFE Alert information into a system for generating resolution manoeuvres and ... "	TSAFE_Alert $\rightarrow$ TSAFE_res	T	T
CTL-1	"As long as the time to loss of separation remains (above threshold), the controller can choose from the following three options: 1. inhibit TSAFE Resolution from issuing a resolution advisory for the current conflict;" "(The controller) takes responsibility for resolving the conflict manually without the help of TSAFE resolution." "2. command TSAFE Resolution to issue a resolution advisory to the aircraft;" "3. take no action at the current time."	AG (TSAFE_Alert = AT $\rightarrow$ E (!TSAFE_snt_cmd U controller_cmd) AG (TSAFE_Alert = AT $\rightarrow$ EX TSAFE_snt_cmd) AG (TSAFE_Alert = AT $\rightarrow$ EG !TSAFE_snt_cmd)	NA	T
LTL-2	"If the controller chooses the second option, TSAFE immediately sends a resolution advisory to the aircraft. "	G (TSAFE_snt_cmd $\rightarrow$ X TSAFE_cmd)	T	T
LTL-3	"... in the events that the time to LOS in TSAFE alert falls below threshold, without the controller having chosen the first or second option, responsibility for resolving the conflict defaults automatically to TSAFE Resolution."	G(TSAFE_res & TSAFE_Alert = BT $\rightarrow$ X TSAFE_cmd) G(TSAFE_Alert = BT & CTR_control) $\rightarrow$ X !CTR_control	T	T
LTL-4	"A message on the controllers monitor indicates when the aircraft has cleared the conflict and control of the aircraft is handed off to the controller. "	G (ac.TSAFE_cmd_done $\rightarrow$ X CTR_control)	T	T
LTL-5	"As in TCAS, pilot compliance within a specified time period will be required." "Pilots need to be trained to respond to TSAFE and to understand the difference in the conditions that trigger either a TSAFE or TCAS alert."	G(ac.TCAS_cmd & !ac.TSAFE_cmd_done $\rightarrow$ (!ac.TSAFE_cmd_done U ac.TCAS_cmd_done))	F	T

ID	Model Verification Specifications	Formula	Abs.	Cpl.
LTL-6	All TSAFE alert will be resolved finally.	G (TSAFE_alert$\neq$ Non $\rightarrow$ F (TSAFE_alert=Non))	F	T
LTL-7	If TSAFE takes control of aircraft, it will finally hand control off to the controller.	G (TSAFE_control $\rightarrow$ F !TSAFE_control)	F	T
LTL-8	If the controller hands the control of an aircraft to TSAFE, this aircraft will not executes the command from the controller.	G ((!controller.CTR_control_1 & !ac1.CTR_command_done) $\rightarrow$ ((!ac1.CTR_command_done U controller.CTR_control_1) ||F ac1.TSAFE_command_done))	F	**F**
INV-2	AutoResolver or TSAFE controls an aircraft exclusively.	(TSAFE_control $\rightarrow$!CTR_control) & (CTR_control $\rightarrow$!TSAFE_control)	T	T
LTL-9	The elder TSAFE resolutions will always be executed ahead of later ones.	G ((ac1.TSAFE_command & !ac2.TSAFE_command) $\rightarrow$ (!ac2.TSAFE_command_done U ac1.TSAFE_command_done))	F	**F**

Table 1: Specifications for model validation from the AAC operational concept and for model verification via model checking

the states in the abstract model and the concrete model is clear. As a special case of Corollary 5.7 in [10], the following theorem holds in our abstract and concrete models.

Theorem [10]: If an invariant or LTL property is true for the abstract model, it must also be true for the concrete model.

In other words, this abstraction is sound but not complete and only true results on the abstract model are conserved in the original model. Note that this theorem does not hold for CTL properties with path predicate E. The abstract model takes only about half an hour to verify all properties. The verification results for the abstract and concrete models are listed in Columns "Abs." and "Cpl." of Table 1 respectively; "T" and "F" designate true and false. We only need to use the concrete model to verify properties that are false for the abstract model, saving run time and providing a valuable tool for model and property debugging.

5 Specification debugging

When model checkers identify counterexamples where the model does not satisfy the model validation specification, it means the system model and the specification are inconsistent, and at least one of them violates designer's intention. However, it is unclear whether the inconsistency is due to an error either in the system model (if it is different from the real system), or in the formal specification. Debugging must be conducted on both the model and the specification of the operational concept. Writing correct formal specifications is difficult. and there is not a automated, systematic approach to specification debugging. We adapt two specification debugging techniques to increase our confidence in our formal specification. LTL satisfiability checking [30] finds erroneous specifications that will always be trivially satisfied or will never be satisfied, regardless of the details of the model. Counterexample-guided specification debugging [31] aids us in fixing imprecise specifications that generate false negative results.

5.1 LTL satisfiability checking

Some errors may not be detected when the model-checking result is positive: a guarantee that the model satisfies the specification does not necessarily answer the real question, namely, whether the system has the intended behavior. If a specification is *valid*, or true in *all* models, then model checking this specification always results in a positive answer and this is certainly due to an error in the specification itself. Similarly, if a specification is *unsatisfiable*, or true in *no* model, then this is also certainly due to an error. Even if each individual specification in our set is satisfiable, their conjunction may be unsatisfiable. Recall that a logical formula φ is valid iff its negation $\neg\varphi$ is not satisfiable [29].

We utilize the observation that LTL satisfiability checking can be reduced to model checking [29]. Consider a formula φ over a set $Prop$ of atomic propositions and a model M that is *universal*, meaning that M contains all possible traces over $Prop$. Then φ is satisfiable precisely when the model M does *not* satisfy $\neg\varphi$. For all LTL and invariant properties, we employ the PANDA tool [30] as a front-end to NuSMV to check the satisfiability of each property, the negation of each property, and the conjunction of all properties. Note that we cannot perform the same test for CTL properties since CTL satisfiability checking is significantly harder than model checking: with respect to formula size, model checking is NLOGSPACE-complete for CTL [18] while satisfiability is EXPTIME-complete for CTL [11].

When employing satisfiability checking, we enhance the method in [30] by further considering

the fairness constraints in the system model, defined by "FAIRNESS" or "JUSTICE" keywords[3] in NuSMV. Fairness, constraining the verification only to executions where some events happen infinitely often, is often used to guarantee the valid progress of the system. Under the (weak) fairness constraint ψ, each LTL formula φ would be treated as $GF\psi \rightarrow \varphi$. Even when PANDA identifies that φ is not valid, an overstrict fairness constraint ψ will cause $GF\psi \rightarrow \varphi$ to be valid, concealing the potential meaningful counterexamples to φ. In a model checker like NuSMV, fairness statements are defined implicitly outside the specifications; this construction has not drawn much attention in previous work on satisfiability checking. PANDA efficiently encodes the negation of each LTL specification as a NuSMV symbolic automaton. We can tackle fairness in LTL satisfiability checking by incorporating all of the fairness constraints in the AAC model into the PANDA output. We conduct satisfiability checking via model checking against a universal NuSMV model. If a formula proves to be satisfiable, we conclude that it is satisfiable under the fairness constraints.

In our experience, this enhancement helped us find an error while all LTL formulas proved satisfiable without the added fairness constraints. Since TSAFE alerts occur rarely in practice (once or twice in an average day), we intended to distinguish consecutive TSAFE alerts using the following fairness constraint:

FAIRNESS $(TSAFE\_Alert = Non)$;

By LTL satisfiability checking with added fairness, we discovered that LTL-6 becomes valid under this constraint. We found that this fairness constraint is too strict. It obscures meaningful counterexamples to LTL-6. We rewrote it as:

FAIRNESS $(TSAFE\_Alert != AT)$;

which also suffices to guarantee the progress of the model.

We debugged our specification set until all specifications (both individually and as a set), and their negations were satisfiable. The performance of our LTL satisfiability checks was consistent with the data in [30]; all of the checks required less than a minute to complete.

5.2 Counterexample-guided specification debugging

When a model checker returns a negative result, the reason may be that the specification is imprecise and thus fails to express the system execution we want to check. We can manually trace spurious counterexamples back to imprecision in the specifications. While this practice aids in eliminating spurious counterexamples, it also relies heavily on verifier's intelligence. We detail the evolution of the formula LTL-8 to show how we performed counterexample-guided specification debugging.

Formula LTL-8 states that "If the controller hands off the control of an aircraft to TSAFE, this aircraft will not execute commands from the controller," which we first wrote an invariant $!(!CTR\_control \& ac.CTR\_cmd\_done)$. NuSMV returns a counterexample, pictured in Figure 5. In this case, a resolution is sent to Aircraft 1, after which the controller gives the control of Aircraft 1 to TSAFE. However, this counterexample is spurious; [13] states that a "TSAFE advisory

[3] We do not address strong fairness (defined by "COMPASSION" in NuSMV) since it does not appear in our model, but our method also applies to strong fairness.

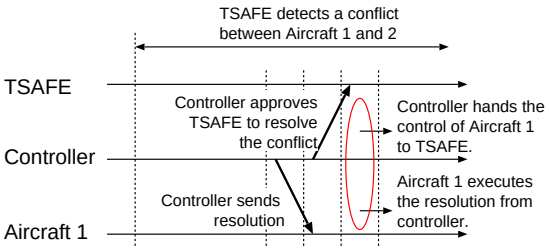

Figure 5: Counterexample to the original formula of LTL-8

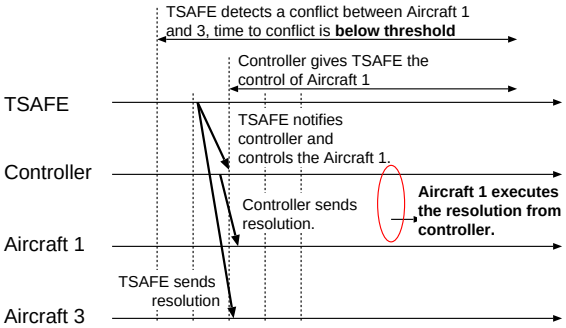

Figure 6: Counterexample to LTL-8

would supersede the most recently issued controller clearance." After TSAFE obtains the control of Aircraft 1, its resolution will supersede the controller's resolution. Thus, Figure 5 shows a valid execution in the AAC. We need to refine the specification to find any scenarios where after the controller gives control to TSAFE, the controller's resolution will still be executed, not superseded by the TSAFE resolution. Thus, we rewrite the original formula as LTL-8 in Table 1 and NuSMV returns a meaningful counterexample, described in the next section.

6 Counterexamples

We elaborate on the counterexamples returned for formulas LTL-8 and LTL-9 and discuss the corresponding fixes they inspired in the AAC system design.

Formula LTL-8 specifies that after a controller has transferred control of an aircraft, that aircraft should not execute a resolution from its previous controller. However, counterexamples are generated for some cases when a controller's resolution is sent out *just before* transfer of control. One such counterexample depicts the following sequence of events:

1. TSAFE detects a conflict between Aircraft 1 and Aircraft 3 with a time to LOS below the TSAFE threshold, indicating TSAFE should take control of both Aircraft 1 and 3 automatically in the next time step. TSAFE generates a maneuver for Aircraft 3 as the resolution.

2. TSAFE notifies the controller that it will take control of Aircraft 1 and 3 and also send a resolution to Aircraft 3, but just before TSAFE takes control of Aircraft 1, the controller sends a command to Aircraft 1.

3. Aircraft 1 receives the resolution from the controller; the pilot executes this resolution since no higher-priority resolution was received.

Aircraft 1 executes the controller's resolution despite the transfer of control to TSAFE. Since this resolution does not take the urgent TSAFE alert into consideration, it may potentially cause a conflict between Aircraft 1 and 3. This counterexample reflects two problems in the AAC protocol. First, aircraft control transfers are only defined between TSAFE and the controller. There is no aircraft notification of control transfers; the resolution received by Aircraft 1 appears valid despite the transfer of control to TSAFE. Second, per the TSAFE protocol, aircraft involved in a conflict do not receive notice of the conflict resolution if they are not required to change course to resolve the conflict. Since Aircraft 1's pilot is not aware of the transfer of control to

TSAFE and does not receive a higher-priority resolution from TSAFE, the controller's command effectively overrides TSAFE's control without violating the pilot's command priority ranking protocol. In response to this counterexample, AAC system designers added a requirement that a "hold current route" command be sent as a part of such a TSAFE resolution. Now, in the above scenario, Aircraft 1 would receive the controller's command followed by a "hold current route" resolution from TSAFE, superseding the controller's command and ensuring safe separation as Aircraft 3 executes the TSAFE maneuver.

A counterexample to LTL-9 depicts the following scenario:

1. TSAFE detects a conflict between Aircraft 1 and 3 with a time to LOS below the TSAFE threshold. TSAFE sends a maneuver for Aircraft 1 as a resolution.
2. Next, TCAS raises an alert in Aircraft 1. Aircraft 1 must immediately execute the TCAS resolution.
3. TSAFE detects a new conflict between Aircraft 1 and 2 and, seeing that Aircraft 1 has not executed the previous TSAFE resolution, attempts to resolve both the conflict between Aircraft 1 and 2 and that between Aircraft 1 and 3 with a new resolution involving all three aircraft. TSAFE then generates a cooperative maneuver for Aircraft 2 and 3, but not Aircraft 1.

In this circumstance, the first resolution sent to Aircraft 1 is expired and should now be ignored but, since the replacement resolution did not include a maneuver for Aircraft 1, it does not know to ignore the expired resolution. After completing the TCAS resolution, it executes the next highest-priority command, which happens to be the expired TSAFE resolution. This problem pinpoints a gap in the protocol definition for the synchronization of multiple resolutions. The current AAC protocol does not define when and how a new TSAFE resolution can override a previous resolution that was sent but did not resolve the targeted conflict, for example, due to a delay caused by a TCAS resolution. Since the new TSAFE resolution may be sent to a different aircraft from the previous one, system designers added a requirement for a notification from TSAFE to invalidate the expired TSAFE resolution. Through experimenting with modifications to our AAC model, we also found that we can eliminate this counterexample to LTL-9 by only allowing TSAFE to send a new resolution after a *sufficiently long* interval following the previous resolution, where sufficiently long is defined to equal the projected time required for the affected aircraft to execute the previous TSAFE resolution.

7 Conclusions and future work

We detail all of the facets of adapting classical model checking to a real aerospace system, including deriving the formal model and a set of specifications from natural language descriptions. To ensure the model checking results are meaningful, we have to ensure that both the model and specifications correctly reflect the intentions of the designers, thus we employ model validation and property debugging techniques. We demonstrate the utility of enhancing LTL satisfiability checking by taking the fairness constraints of the system model into consideration. We argue that specification debugging in real applications deserves more attention in future research efforts, and the utility of a *system formalization*, *model and specification debugging*, and *verification* trilogy for model checking real systems under development.

In this paper we assume there are no hardware failures or lost messages. As the AAC design develops and hardware details are decided by AAC designers, we plan to take the failure rates

of the chosen components into consideration, i.e. by extending our work to probabilistic model checking using PRISM [19]. Previous work has reported on analyzing the safety of air traffic control systems using simulation [3] or fault trees [1]. By extending the model we designed in this paper, we can carry out safety analysis using PRISM to capture the dynamic interactions in the AAC.

Bibliography

[1] J. Andrews, Welch J., and Erzberger H. Safety analysis for advanced separation concepts. *Air Traffic Control Quarterly*, 14(1), 2006.

[2] R Bloem, R Cavada, I Pill, M Roveri, and A Tchaltsev. RAT: A Tool for the Formal Analysis of Requirements. In *CAV*, volume 4590, pages 263–267. Springer, 2007.

[3] D. Blum, D. Thipphavong, T. Rentas, Y. He, X. Wang, and M.E. Patẽ-Cornell. Safety analysis of the advanced airspace concept using Monte Carlo simulation. *AIAA Meeting Papers on Disc*, 15(9), 2010.

[4] M. Bozzano, A. Cimatti, J. Katoen, V. Nguyen, T. Noll, and M. Roveri. The COMPASS approach: correctness, modelling and performability of aerospace systems. In *SAFECOMP, 5775, LNCS*, pages 173–186. Springer, 2009.

[5] D. Bushnell, D. Giannakopoulou, P. Mehlitz, R. Paielli, and C. Pasareanu. Verification and validation of air traffic systems: Tactical separation assurance. In *IEEE Aerospace Conf.*, pages 1 –10, 2009.

[6] Y. Choi. From NuSMV to SPIN: Experiences with model checking flight guidance systems. *FMSD*, 30(3):199–216, 2007.

[7] Y. Choi and M. Heimdahl. Model checking software requirement specifications using domain reduction abstraction. In *IEEE ASE*, pages 314–317, 2003.

[8] A. Cimatti, E.M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In *CAV*, LNCS 2404, pages 359–364. Springer, 2002.

[9] A. Cimatti, E.M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: A new symbolic model checker. *STTT Int'l J.*, 2(4):410–425, 2000.

[10] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5):1512–1542, 1994.

[11] E.A. Emerson and J.Y. Halpern. Decision procedures and expressiveness in the temporal logic of branching time. *J. of Computer and System Sci*, 30:1–24, 1985.

[12] H. Erzberger. Automated conflict resolution for air traffic control. In *International Congress of the Aeronautical Sciences*, 2004.

[13] H Erzberger and K Heere. Algorithm and operational concept for resolving short-range conflicts. *Proc. IMechE G J. Aerosp. Eng.*, 224(2):225–243, 2010.

[14] H. Erzberger, T. Lauderdale, and Y. Chu. Automated conflict resolution, arrival management and weather avoidance for ATM. In *International Congress of the Aeronautical Sciences*, 2010.

[15] D. Giannakopoulou, D. Bushnell, J. Schumann, H. Erzberger, and K. Heere. Formal testing for separation assurance. *AMAI*, 63:5–30, 2011.

[16] M. Gribaudo, A. Horvath, A. Bobbio, E. Tronci, E. Ciancamerla, and M. Minichino. Model-checking based on fluid Petri nets for the temperature control system of the ICARO co-generative plant. Technical report, SAFECOMP, 2434, LNCS, 2002.

[17] C Heitmeyer, R. Jeffords, and B. Labaw. Automated consistency checking of requirements specifications. *ACM Trans. Softw. Eng. Methodol.*, 5(3):231–261, July 1996.

[18] O. Kupferman, M.Y. Vardi, and P. Wolper. An automata-theoretic approach to branching-time model checking. *J. ACM*, 47(2):312–360, Mar 2000.

[19] M. Z. Kwiatkowska, G. Norman, and D. Parker. Probabilistic symbolic model checking with PRISM: a hybrid approach. *STTT*, 6(2):128–142, 2004.

[20] S. Miller. Will this be formal? In *TPHOLs 5170*, pages 6–11. Springer, 2008.

[21] S. Miller, A. Tribble, and M. Heimdahl. Proving the shalls. In *FME*, volume 2805 of *LNCS*, pages 75–93. Springer, 2003.

[22] S. P. Miller, A. C. Tribble, M. W. Whalen, M. Per, and E. Heimdahl. Proving the shalls. *STTT*, 8(4-5):303–319, 2006.

[23] C. Muñoz, G. Dowek, and V. Carreño. Modeling and verification of an air traffic concept of operations. In *ISSTA*, ISSTA '04, pages 175–182. ACM, 2004.

[24] R. Paielli, H. Erzberger, D. Chiu, and K. Heere. Tactical conflict alerting aid for air traffic controllers. *AIAA J. Guid. Control Dyn.*, 32(1):184–193, 2009.

[25] R. Paielli, H. Erzberger, D. Chiu, and K. Heere. Tactical conflict alerting aid for air traffic controllers. *J Guid Contr Dynam*, 32(1):184–193, 2009.

[26] I. Pill, S. Semprini, R. Cavada, M. Roveri, R. Bloem, and A. Cimatti. Formal analysis of hardware requirements. In *DAC '06*, pages 821–826, New York, NY, USA, 2006. ACM.

[27] A. Cimatti R. Cavada, C. Jochim, G. Keighren, E. Olivetti, M. Pistore, M. Roveri, and A. Tchaltsev. NuSMV2.4 user manual. Technical report, CMU/ITC-irst, 2005.

[28] F. Raimondi, A. Lomuscio, and M. J. Sergot. Towards model checking interpreted systems. In *FAABS02*, *LNAI 2699*, pages 115–125. Springer, 2002.

[29] K.Y. Rozier and M.Y. Vardi. LTL satisfiability checking. *STTT*, 12(2):123–137, 2010.

[30] K.Y. Rozier and M.Y. Vardi. A multi-encoding approach for LTL symbolic satisfiability checking. In *FM*, volume 6664 of *LNCS*, pages 417–431. Springer, 2011.

[31] R. Siminiceanu and G. Ciardo. Formal verification of the NASA Runway Safety Monitor. *STTT*, 9(1):63–76, 2007.

[32] A. Tribble and S. Miller. Software safety analysis of a flight management system vertical navigation function-a status report. In *DASC*, pages 1.B.1–1.1–9 v1, 2003.