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

Yang Zhao[a], Kristin Yvonne Rozier[b]

[a]University of California, Riverside, CA, USA. zhaoy@cs.ucr.edu
[b]NASA Ames Research Center, Moffett Field, CA, 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 both NuSMV and CadenceSMV, and perform *model validation* by checking against temporal logic specifications in LTL and CTL that 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: Temporal Logic, 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) [1] 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 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, particularly for analysis of coordination protocols. Model checkers automatically perform an exhaustive check that a system model satisfies a specification, given in a temporal logic such as Linear Temporal Logic (LTL) or Computation Tree Logic (CTL). If the system does not meet the given specification, the model checker returns a counterexample detailing a system trace that demonstrates the specification violation. *Symbolic* model checkers represent and analyze the system model and specifications symbolically, reasoning over sets of states and sets of transitions using binary decision diagrams (BDDs) [2] to increase scalability over *explicit-state* model checkers that explicitly enumerate the state space (see e.g. [3] for an experimental comparison). We use the symbolic model checkers CadenceSMV[1] [4] and NuSMV[2] [5, 6], which both evolved from the original Symbolic Model Verifier developed at CMU [7] and use nearly identical input languages. We choose CadenceSMV and NuSMV because they are well-documented [8, 9], freely available, and frequently used in industry [10, 11, 12, 13, 14, 15, 16, 17, 18] (see [19] for a survey).

[1]Release 10-11-02p46. `http://www.kenmcmil.com/smv.html`

[2]Version 2.5.4-zchaff, *open source*. `http://nusmv.irst.itc.it/`

We build a *system model* of the operational concept [20, 21, 1, 22] in formal semantics and also describe the behaviors required for safe system operation, the *model verification specifications*, using temporal logic. NuSMV and CadenceSMV are then able to prove that the system model conforms to the model verification specifications or return counterexamples corresponding to possible executions in the operational concept that violate these specifications. 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 system model of the AAC operational concept building upon existing literature and input from human experts via about 100 person-hours of interviews with system architects [23]. We extract *model validation specifications* from the operational concept and construct *model verification specifications* by codifying the expectations of system designers, extracted via conversation [23], in temporal logic; all models and specifications are available online.[3]

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 and the specifications correctly reflects the designers' intentions. Counterexamples returned from the model checker may not reveal problems in the operational concept itself, but may instead correspond to imprecisions in the system model or unrepresentative specifications. Our solution to this problem is to create both the system model and a set of temporal logic properties (our model validation specifications) from the system description in the operational concept as two different

[3]All models for both NuSMV and CadenceSMV, all specifications, and the commands we ran to check them are available at `http://ti.arc.nasa.gov/m/profile/kyrozier/AAC/AAC.html`.

ways of formalizing the same description of the designers' intentions. We validate our system model by model checking this set of specifications, which directly correspond to statements in the designers' 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. To help ensure that the verdict of the model checker (either positive or negative) is not due to mistakes in the specifications, we employ specification debugging for all specifications (model validation specifications and model verification specifications). We exemplify our process of specification debugging, including LTL satisfiability checking [3, 24] and counterexample-guided specification 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 designers' intentions.

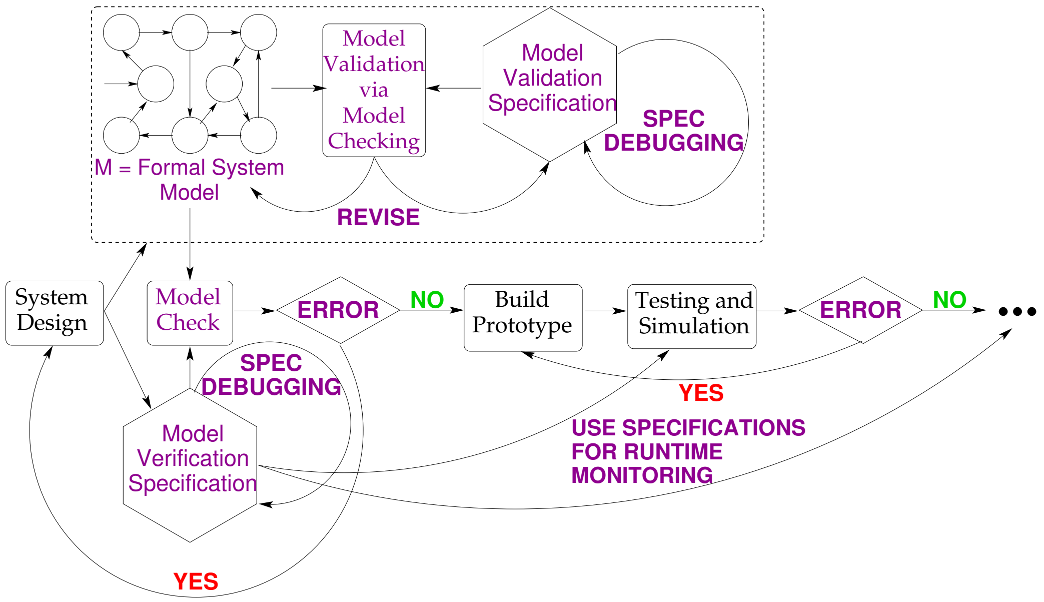
M = Formal System Model
Model Validation via Model Checking
Model Validation Specification
SPEC DEBUGGING
REVISE
System Design
Model Check
ERROR
NO
Build Prototype
Testing and Simulation
ERROR
NO
YES
SPEC DEBUGGING
Model Verification Specification
USE SPECIFICATIONS FOR RUNTIME MONITORING
YES

Figure 1: A better critical system design process, where the system design is verified previous to, and separately from, the system prototype and later implementation.

The third aspect, *model checking*, completes our system verification process by checking the system model against the model verification specifications. We argue for this critical system design process, pictured in Figure 1, wherein the system design and system requirements are formalized, validated, debugged, and verified via model checking before any system prototype is built. This process clarifies the design and requirements up front, helps to separate design and specification errors from prototyping or other errors, and saves costs as unexpected emergent behaviors in the system design are caught early when it is the easiest and least expensive to fix them. (Note also that products from model checking can be re-used later in the system development process, such as by runtime monitoring the model verification specifications during simulation or system runtime [25, 26, 27, 28].) We detail two emergent unexpected behaviors of the early AAC system design that were pinpointed by counterexamples and discuss possible modifications to prevent 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 modeling, 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. Our verification results are explored in 6. Section 6.1 presents important counterexamples describing unexpected emergent behaviors along with modifications to the system design to combat them. We include performance results and other practical concerns in Section 6.2. Section 7 concludes and points out future work.

2. Related Work

The three AAC software components that calculate resolution maneuvers have undergone verification through simulation as well as ongoing component-level verification utilizing formal methods [29, 30, 31]. 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 a formidable task, and too time-consuming to provide effective system logic evaluation, including capturing rare events. Efficiently evaluating different potential system configurations, as we do here, was not previously possible.

Although [32] 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 [33, 34, 35, 17]. 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 [34] and [17] generate the system models from the source code, and [33, 35] generate NuSMV models automatically from another formal language. In this way our study is particularly instructive: we found the creation and validation of the system model from an operational concept to be a major bottleneck in our verification process.

Specification debugging has drawn much attention [36, 37, 38, 39]. These methods, also called *sanity checks* [37] analyze the *consistency*, *completeness*, and *safety* of formal specifications. Sanity checks in industry include many types of simple, often ad hoc, tests such as checking for duplicate conflicting variable assignments or enabling conditions that are never enabled [40]. *Vacuity checking* can help detect errors in specifications by checking whether a subformula of a specification does not affect the satisfaction of the specification in the system model [41]. *Inherent vacuity checking* is a set of sanity checks that can be applied to a set of temporal properties, even before a model of the system has been developed, but many possible errors cannot be detected by inherent vacuity checking [42]. Though CadenceSMV and NuSMV do not include built-in options for vacuity checking, this capability is available in some proprietary industrial tools [43]. *Realizability checking* provides another, stronger sanity check for a set of temporal properties in LTL by testing whether there is an open system that satisfies all the properties in the set [44], but such a test is very computationally expensive and we do not know of any freely available implementations. However, [3] demonstrated that the sanity check of *satisfiability checking* of LTL specifications can be easily incorporated as a prelude to model checking and argued for checking each specification, its negation, and the conjunction of all LTL specifications for satisfiability. Therefore we use the efficient PANDA tool

[24] to perform this sanity check on all of our LTL specifications.

Previously, [38, 39] presented a platform for formal analysis of hardware requirements called “RAT,” which has been applied in the analysis of aerospace systems [45]. Using this framework, instead of creating a system model, we only need to generate the model validation specifications in Section 4.2 as the formal specifications of the operational concept; RAT is able to check this specification against our model verification specifications. However, the consistency between the formal specifications under verification and the designers’ 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 [31, 35, 32, 34] 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 verification 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 [33], 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 our NuSMV and CadenceSMV 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-term, and near-term conflicts, and collision avoidance. Figure 2 illustrates the infrastructure of the AAC.

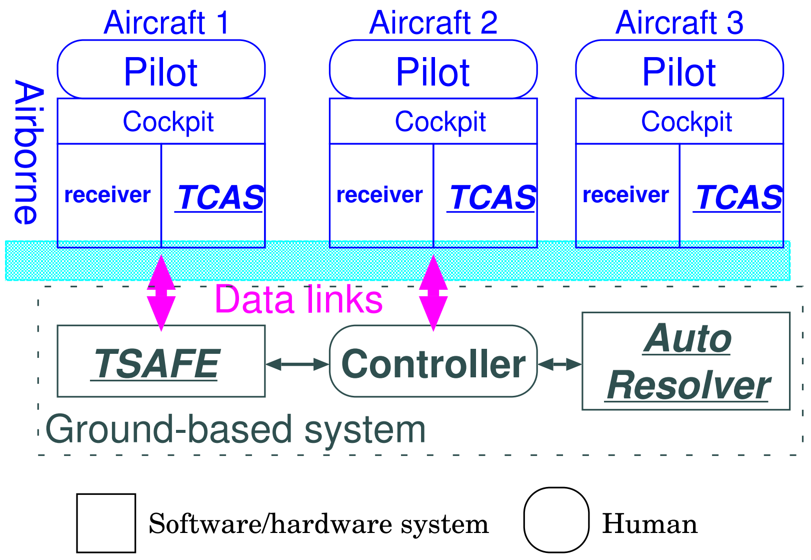

Figure 2: Automated Airspace Concept

The strategic separation layer, referred to as the *AutoResolver*, addresses short-term conflicts from 20 minutes in the future down to 3 minutes out. 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 sending the resolutions generated from the AutoResolver.

The tactical separation layer, known as the *Tactical Separation Assured Flight Environment* (*TSAFE*), addresses near-term 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 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 [20, 21, 1, 22] specifies the authority and responsibilities of the above three systems, controllers and pilots. Figure 3 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 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 3, 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 [1], 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 3. After the conflict is resolved, TSAFE will return control of the aircraft involved to the controller, as shown in time slot (4) in Figure 3. 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.

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.

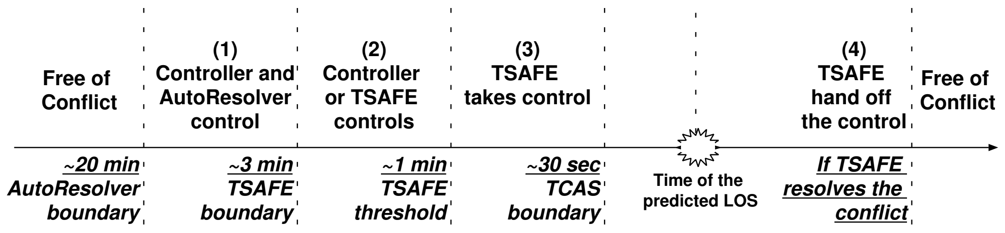
Free of Conflict
(1) Controller and AutoResolver control
(2) Controller or TSAFE controls
(3) TSAFE takes control
(4) TSAFE hand off the control
Free of Conflict
~20 min AutoResolver boundary
~3 min TSAFE boundary
~1 min TSAFE threshold
~30 sec TCAS boundary
Time of the predicted LOS
If TSAFE resolves the conflict

Figure 3: Operational concept for each aircraft

4. Formalization Methodology

The system model and formal specifications (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. Modeling the AAC

Our goal in this case study is to answer the *fundamental question* of early-stage system logic design: if all of the components work perfectly all of the time, then does the system as a whole behave the way we expect? If not, this needs to be fixed in the initial logic design; otherwise time and costs will be higher later in the testing process when methods like fault injection are employed and test engineers search for safe operation in the presence of faults. We must first show that the system operates correctly with no faults. Accordingly, we introduce assumptions that each of the system components operates correctly and then seek to prove there are no logical design flaws in the AAC coordination protocol.

Our model assumes the following: (1) All potential 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

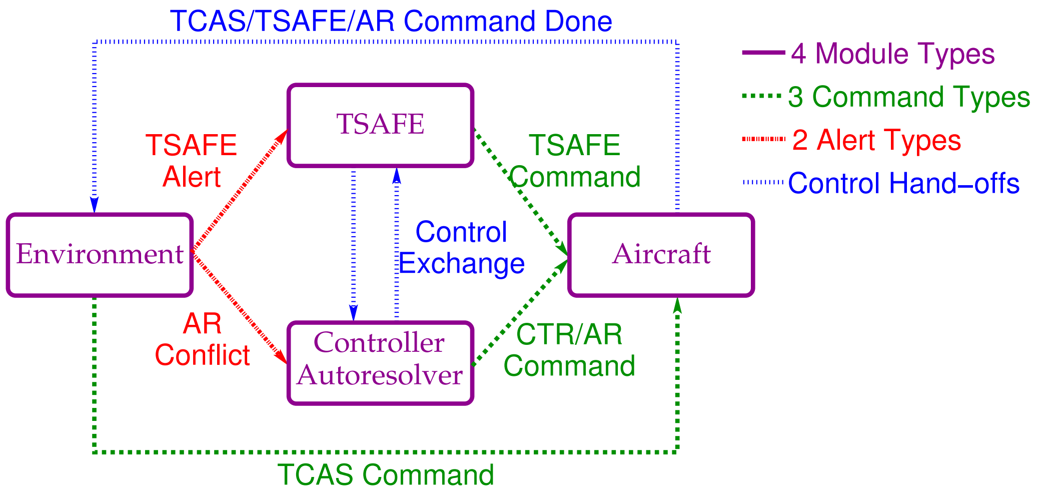

Figure 4: NuSMV/CadenceSMV module types and the interconnections between modules

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 4, the AAC is modularized into four components: *Aircraft*, *TSAFE*, *Controller*, and *Environment*. Arrows show the important input/output variables interconnecting the modules. Note that there can 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 1. If TSAFE transmits a resolution to Aircraft 1, TSAFE_command_1 is set to **1**, and the *Aircraft 1* module will react accordingly.

In the real system, the inputs to TSAFE and the AutoResolver are the

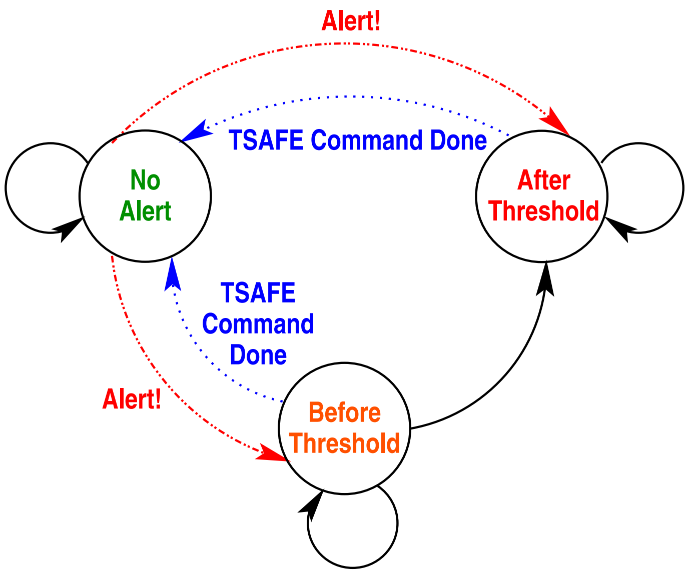

Figure 5: Finite State Machine for variable TSAFE_Alert

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, BT, and AT, corresponding to no LOS detected, or LOS detected with time to LOS before or after 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 = AT" means there is a LOS after the threshold time between Aircraft 1 and 2. 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 the "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 5 shows the finite state machine (FSM) for the 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 effects 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 conflicts 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 the 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.

Although not a part of the AAC concept, the *Environment* module plays an important role in our model: generating all possible conflict conditions, which are sequences of conflicts, corresponding to what can actually occur between aircraft in the airspace sector. If the *Environment* module fails to generate a certain conflict condition that may occur, the possible failure of the AAC concept in this case would not be revealed by the model checker since no feasible counterexample exists in the model due to the incomplete *Environment* module. On the other hand, the *Environment* tends to be aggressive in producing conflicts. For example, in practice, TSAFE alerts are usually not frequent (once or twice in a day), and consecutive TSAFE alerts are even rarer. However, this observation is not reflected in the *Environment* module, and the generated counterexamples in Section 6.1 consist of complex sequences of different alerts. Indeed, the purpose of model checking protocols that have previously been thoroughly examined by domain experts i.e. for rigorous peer-reviewed publication like the AAC operational concept is that model checking finds counterexamples too rare and complex to be discovered by human reasoning. Model checking considers all of the rare corner cases that are difficult or impossible to cover by testing with real flight data and guarantees when a property is proved that the proof is exhaustive. Consequently, one of the challenges we faced when modeling the *Environment* module is identifying and eliminating the false negative results triggered by inconsistencies between the *Environment* module and the AAC environment in practice; a property may be disproved due to the existence of a conflict condition that exists in the *Environment* module but is not possible in practice. We demonstrate revising a specific property to tackle this challenge in Section 5.2.

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. To avoid the state space explosion problem, we need to build the AAC model containing the minimum sufficient number of aircraft instances, which leads us to the following Lemma.

Lemma (See [23]): We can reason about all meaningful interleavings of the AutoResolver and TSAFE conflict lists utilizing *three* aircraft.
Proof sketch: 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. We can safely assume the same pair of aircraft cannot appear on both the AutoResolver and TSAFE conflict lists at the same time because if this happens in the real AAC, the conflict will be removed from the AutoResolver list automatically, either by a system check or by the issuance of the ensuing TSAFE resolution. Therefore in our model we safely remove any shared TSAFE/AutoResolver conflicts 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, therefore we know 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 can model such scenarios using two instances of the *Aircraft* module. However, we must also account for the cases where the same (single) aircraft is involved in both a TSAFE and an AutoResolver conflict; this requires using three instances of the *Aircraft* module. Therefore, due to the pairwise algorithms implemented by AutoResolver and TSAFE and our correct resolution assumption, we conclude, in agreement with system designers [23], that it is possible to reason about all meaningful corner cases of the real system utilizing *three* aircraft. $\Box$

The *TSAFE* module models the behaviors of TSAFE with its dedicated data link. For a TSAFE conflict between Aircraft 1 and 2 (TSAFE_Alert_12 = BT or AT), the TSAFE algorithm may generate a resolution for only Aircraft 1 (TSAFE_command_1= 1), only Aircraft 2 (TSAFE_command_2= 1), or both (TSAFE_command_1= 1 $\wedge$ 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 AutoResolver alerts arise in the *Environment* module, these two modules react as defined in the operational concept and output resolutions for the involved aircraft.

Both NuSMV and CadenceSMV derive their input languages from the original SMV [7] so they share similar syntax and semantics. We created our models first in NuSMV and then later translated them to CadenceSMV, an exercise that we found to be straightforward. CadenceSMV supports some new features in its modeling language, and we will mention one of them in Section 4.3. The full-scale model contains 97 Boolean variables, including the variables indicating conditions of conflicts, TCAS/TSAFE/controller's resolutions, resolutions received by aircraft and hand-shaking signals between modules in their communications.

4.2. Specifications

We write two sets of specifications: model validation specifications to ensure the model follows the system design description as shown in Table 1, and verification specifications to capture the emergent behaviors that should be avoided in the AAC system, as shown in Table 2. Our specification sets are expressed using 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. In Tables 1 and 2, formulas are identified with IDs in the first column, where "LTL-" and "CTL-" distinguish LTL and CTL formulas respectively. The specifications coalesce properties from two sources: the properties for model validation are from [1], and the properties for verification are generated from discussions with AAC designers. The correspondence between the operational concept statements and specifications is listed in these tables. For model validation specifications, which should correctly reflect the intentions of the designers, we quote the original sentences from [1] in the second column of Table 1 followed by a brief explanation after "$\Longrightarrow$" to demonstrate how we converted the natural language system design into formal temporal logic. Verification specifications are primarily generated from the discussions with the designers, and we provide brief descriptions in the second column of Table 2.

ID	Model Validation Specifications from [1]	Formula
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. "(Section 8, page 239, left column) $\Longrightarrow$ If the controller is in control of an Aircraft that is not currently involved in any TSAFE alert, as long as this Aircraft will not be involved in a TSAFE alert after the threshold (i.e. next), the controller will retain the control of this Aircraft after that (i.e. next next time).	G (controller.CTR_control_1 ∧ env.tsafe_alert_12 = Non ∧ env.tsafe_alert_13 = Non ∧ env.tsafe_alert_23 = Non ∧ (X env.tsafe_alert_12 != AT) ∧ (X env.tsafe_alert_13 != AT) ∧ (X env.tsafe_alert_23 != AT) → X (X controller.CTR_control_1))
CTL-1	"As long as the time to loss of separation remains (before 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." (Section 8, page 239, right column) $\Longrightarrow$ If there is a TSAFE alert between Aircraft 1 and 2 (before the threshold) and there are no more imminent TSAFE alerts, there is always a path where the controller does not permit the TSAFE resolution but resolves the conflict by controller resolution.	AG ((env.tsafe_alert_12 = BT ∧ env.tsafe_alert_13 != AT ∧ env.tsafe_alert_23 != AT ∧ controller.CTR_control ∧ !controller.tsafe_sndcmd ∧ !ac1.TSAFE_command_done ∧ !ac2.TSAFE_command_done) → (E [!controller.tsafe_sndcmd U controller.CTR_command1]))
CTL-2	"2. command TSAFE Resolution to issue a resolution advisory to the aircraft;" (Section 8, page 240, left column) $\Longrightarrow$ If there is a TSAFE alert between Aircraft 1 and 2 (before the threshold) and there are no more imminent TSAFE alerts, there is always a path where the controller permits the TSAFE resolution in the following time step.	AG ((env.tsafe_alert_12 = BT ∧ env.tsafe_alert_13 != AT ∧ env.tsafe_alert_23 != AT ∧ controller.CTR_control ∧ !controller.tsafe_sndcmd ∧ !ac1.TSAFE_command_done ∧ !ac2.TSAFE_command_done) → EX (controller.tsafe_sndcmd)
CTL-3	"3. take no action at the current time." (Section 8, page 240, left column) $\Longrightarrow$ If there is a TSAFE alert between Aircraft 1 and 2 (before the threshold) and there are no more imminent TSAFE alerts, there is always a path where the controller never permits the TSAFE resolution.	AG ((env.tsafe_alert_12 = BT ∧ env.tsafe_alert_13 != AT ∧ env.tsafe_alert_23 != AT ∧ controller.CTR_control ∧ !controller.tsafe_sndcmd ∧ !ac1.TSAFE_command_done ∧ !ac2.TSAFE_command_done) → EG (!controller.tsafe_sndcmd)
LTL-2	"If the controller chooses the second option, TSAFE immediately sends a resolution advisory to the aircraft. " (Section 8, page 240, left column) $\Longrightarrow$ In case two Aircraft are involved in a TSAFE alert before the threshold, and the controller decides to give the control of these two Aircraft to TSAFE, TSAFE will send a resolution to at least one of the Aircraft in the following time step.	G ((env.tsafe_alert_12 = BT ∧ env.tsafe_alert_13 != AT ∧ env.tsafe_alert_23 != AT ∧ controller.CTR_control ∧ controller.tsafe_sndcmd ∧ tsafe.TSAFE_window) → X (tsafe.TSAFE_command1 | tsafe.TSAFE_command2));
LTL-3	"... in the event that the time to loss of separation falls below 1 minute, without the controller having chosen the first or second option, responsibility for resolving the conflict defaults automatically to TSAFE Resolution." (Section 8, page 240, left column) $\Longrightarrow$ If there is a TSAFE alert after the threshold between Aircraft 1 and 2, then a TSAFE resolution for Aircraft 1 or 2 will be sent.	G ((env.tsafe_alert_12 = AT ∧ env.tsafe_alert_13 != AT ∧ env.tsafe_alert_23 != AT ∧ controller.CTR_control ∧ !ac1.TSAFE_command ∧ !ac2.TSAFE_command) → F (tsafe.TSAFE_command1 | tsafe.TSAFE_command2))

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. " (Section 8, page 240, right column) $\Longrightarrow$ When TSAFE controls Aircraft 1 and Aircraft 1 has completed the TSAFE maneuver, TSAFE will hand off the control of Aircraft 1 to controller.	G (ac1.TSAFE_command_done $\wedge$!ac1.TSAFE_command $\wedge$ tsafe.TSAFE_control1 $\rightarrow$ X !tsafe.TSAFE_control1)
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." (Section 8, page 241, left column) $\Longrightarrow$ Pilots will always first execute the resolutions from TCAS, then TSAFE, and then AutoResolver.	TCAS: G(ac.TCAS_cmd $\wedge$!ac.TSAFE_cmd_done $\rightarrow$ (!ac.TSAFE_ cmd_done U ac.TCAS_cmd_done)) TSAFE: G(ac1.TSAFE_command $\wedge$!ac1.CTR_command_done $\rightarrow$ (!ac1.CTR_command_done U ac1.TSAFE_command_done))

Table 1: Specifications for model validation from the AAC operational concept

ID	Model Verification Specifications from [23]	Formula
LTL-6	All TSAFE alerts will be resolved eventually.	G (TSAFE_alert_12 != Non $\rightarrow$ F (TSAFE_alert_12=Non))
LTL-7	If TSAFE takes control of Aircraft 1 at some point it will eventually hand the control off to the controller.	G(F (env.tsafe_alert_12 = Non $\wedge$ env.tsafe_alert_13 = Non)) $\rightarrow$ G(tsafe.TSAFE_control1 $\rightarrow$ F !tsafe.TSAFE_control1)
LTL-8	If the controller hands off the control of an aircraft to TSAFE, this aircraft will not execute commands 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))
LTL-9	The elder TSAFE resolutions will always be executed ahead of later ones. (Note that if a TSAFE_command variable is set to *true* it stays *true* until the corresponding TSAFE_command_done flag is *true*.)	G ((ac1.TSAFE_command & !ac2.TSAFE_command & !ac2.TSAFE_command_done) $\rightarrow$ ((!ac1.TSAFE_command_done & !ac2.TSAFE_command_done) U ac1.TSAFE_command_done))
LTL-10	If a TSAFE command is sent to an aircraft, controller/AutoResolver should then hand off the control of this aircraft.	G (tsafe.TSAFE_command1 $\wedge$ controller.CTR_control_1 $\rightarrow$ X !controller.CTR_control_1);

Table 2: Specifications for model verification of the AAC operational concept via model checking

4.3. Model Abstraction

Our full-scale model takes more than 10 hours using NuSMV and 1 hour using CadenceSMV, each with the default flags, to verify all properties in Tables 1 and 2. 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 [46] to speed up the verification process. Altogether, constructing the full-scale system model plus testing abstractions to derive this useful abstract version of the model required approximately three man months.

We adopt an abstraction without the assumption of "competent pilot" and assume pilots can choose arbitrary behaviors as the response to the received resolutions. In NuSMV, this abstraction simply removes statements defining the next state of the *TCAS/TSAFE/CTR_command_done* variables in the *Aircraft* module. CadenceSMV supports a new feature called layer declarations, which allows users to define specifications that should be followed for a module. We put the assignments to *TCAS/TSAFE/CTR_command_done* in the layer called *pilot*. In the complete model, we enforce that the three instantiations of the *Aircraft* module, *ac1*, *ac2*, *ac3*, fulfill the layer *pilot*, written as *ac1//pilot*, *ac2//pilot*, *ac3//pilot*, while in the abstract model, we consider module free of this layer by *ac1//free*, *ac2//free*, *ac3//free*.

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 the states in the abstract model and the concrete model is clear. As a special case of Corollary 5.7 in [46], the following theorem holds in our abstract and concrete models.

Theorem (See [46]): If an invariant or LTL property is true for the abstract model, it must also be true for the concrete model.
Proof sketch: Assume an invariant or LTL property φ does not hold for the concrete model, there exists a path π in the concrete model that represents a counterexample, where $\pi \models \neg\varphi$. Note that any feasible execution in the concrete model is also a feasible execution in the abstract model, there exists the same π in the abstract model that also violates φ, meaning that φ does not hold in the abstract model either.

ID	Abs.	Cpl.	ID	Abs.	Cpl.
LTL-1	T	T	LTL-2	T	T
LTL-3	F	T	LTL-4	F	T
LTL-5	F	T	LTL-6	F	T
LTL-7	T	T	LTL-8	F	**F**
LTL-9	F	**F**	LTL-10	T	T

Table 3: The model checking results of each LTL property on the abstract (Abs.) and complete (Cpl.) models. The two counterexamples we found (in bold-face) are detailed in Section 6.1.

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 a little more than a minute to verify all properties for NuSMV and a few seconds for CadenceSMV, again using the default flags for each. The verification results for the abstract and concrete models are listed in Columns "Abs." and "Cpl." of Table 3; "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 a specification, it means the system model and the specification are inconsistent, and at least one of them violates the designers' intentions. However, it is unclear whether the inconsistency is due to an error either in the system (or 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 specifications of the operational concept; we debug both model validation specifications and model verification specifications. Writing correct formal specifications is difficult and there is not an automated, systematic approach to specification debugging. We adapt two specification debugging techniques to increase our confidence in our formal specifications. LTL satisfiability checking [3, 24] 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 [34] 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 [3].

We utilize the observation that LTL satisfiability checking can be reduced to model checking [3]. 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 [24] as a front-end to CadenceSMV to check the satisfiability of each property, the negation of each property, and the conjunction of all properties. (NuSMV can also be used as a back-end to check some PANDA encodings.) Note that we cannot perform the same test for CTL properties since CTL satisfiability checking is significantly harder than CTL model checking: with respect to formula size, model checking is NLOGSPACE-complete for CTL [47] while satisfiability is EXPTIME-complete for CTL [48].

When employing satisfiability checking, we enhance the method in [24] by further considering the fairness constraints in the system model, defined by "FAIRNESS" or "JUSTICE" keywords[4] 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 $GF\psi \to \varphi$. Even when PANDA identifies that φ is not valid, an overstrict fairness constraint ψ will cause $GF\psi \to \varphi$ to be valid, concealing the potential meaningful counterexamples to φ. In a model checker like NuSMV,

[4]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.

fairness statements are defined 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 CadenceSMV or 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 input. We conduct satisfiability checking via model checking against a universal CadenceSMV/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:

$$\textbf{FAIRNESS}\ (\mathit{TSAFE\_Alert} = \mathit{Non});$$

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:

$$\textbf{FAIRNESS}\ (\mathit{TSAFE\_Alert}\ != \mathit{BT});$$

to avoid the case where a TSAFE alert lasts forever before threshold since the alert is either resolved or the time to LOS eventually falls after threshold.

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 [24]; 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. Through counterexample-guided specification debugging, we can manually trace spurious counterexamples back to imprecisions in the specifications. Given a counterexample, we edit the failed specification using the counterexample as a guide to systematically test small modifications to the specification. This allows us to gradually refine the specification until we are confident that it accurately captures the intended system requirement.

While this practice aids in eliminating spurious counterexamples, it also relies heavily on verifier's intelligence and experience.

First, we will investigate the false negative of LTL-7 due to the *Environment* module. LTL-7 specifies that although TSAFE is able to take control of Aircraft 1, at some point, the control of Aircraft 1 will eventually be returned to the controller from TSAFE. In a straightforward way, the property was initially written as:

$$\mathsf{G}(tsafe.TSAFE\_control1 \rightarrow \mathsf{F}!tsafe.TSAFE\_control1)$$

However, model checking returns counterexamples that Aircraft 1 is continuously involved in TSAFE alerts such that TSAFE always holds the control of Aircraft 1. Although these counterexamples are feasible paths in the model, they are beyond the scope of our considerations in the real air traffic control system, since in normal cases, an aircraft should eventually be free of conflict after conflict resolutions. In other words, our model considers conflict conditions that are too harsh to be true in practice. To fix the problem, we specify in the property that we only consider cases where $\mathsf{G}(\mathsf{F}(env.tsafe\_alert\_12 = Non \wedge env.tsafe\_alert\_13 = Non))$, meaning there always be some times when Aircraft 1 is not involved in a TSAFE alert, then the property can be verified. The complete property LTL-7 is listed in Table 1.

Next, 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 as the invariant $!(!CTR\_control \& ac.CTR\_cmd\_done)$. NuSMV returns a counterexample, pictured in Figure 6. 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; [1] states that a "TSAFE advisory 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 6 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 2 and NuSMV returns a meaningful counterexample, described in the next section.

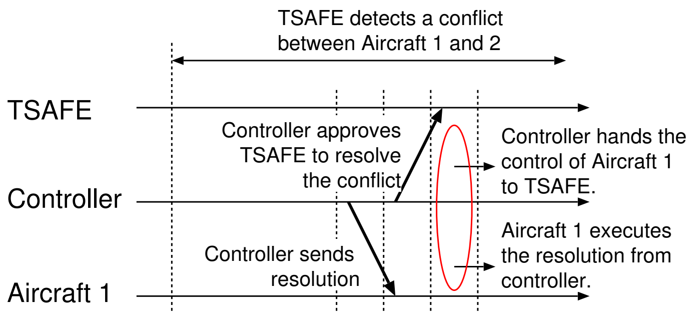
TSAFE detects a conflict between Aircraft 1 and 2
TSAFE
Controller
Aircraft 1
Controller approves TSAFE to resolve the conflict
Controller sends resolution
Controller hands the control of Aircraft 1 to TSAFE.
Aircraft 1 executes the resolution from controller.

Figure 6: Counterexample to the original (incorrect) formula for LTL-8

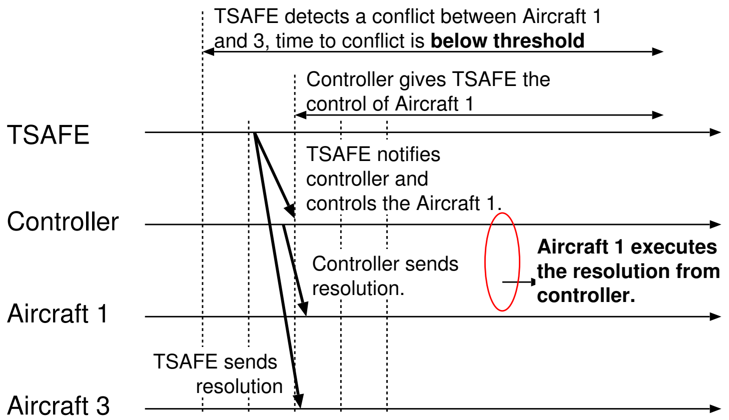
TSAFE detects a conflict between Aircraft 1 and 3, time to conflict is **below threshold**
Controller gives TSAFE the control of Aircraft 1
TSAFE
Controller
Aircraft 1
Aircraft 3
TSAFE notifies controller and controls the Aircraft 1.
Controller sends resolution.
Aircraft 1 executes the resolution from controller.
TSAFE sends resolution

Figure 7: Counterexample to LTL-8

6. Verification Results

In this section we report on two aspects of our verification results. In Section 6.1, we discuss the two most impactful counterexamples returned for our verification specifications, how they reveal unexpected emergent behaviors in the early AAC design, and also discuss the design fix that system designers incorporated in response to our study. In Section 6.2 we detail the computational resources required to obtain these results and compare the runtime and memory consumption of CadenceSMV and NuSMV for each of the specifications.

6.1. 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, pictured in Figure 7:

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 Aircraft 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 Aircraft 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 Aircraft 3. This counterexample reflects two problems in the AAC protocol preliminary design. 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.

6.2. Performance

We ran CadenceSMV and NuSMV on an Intel Xeon 2.53GHz workstation running 64-bit CentOS Linux (kernel version 2.6.18) with 36GB RAM. Since CadenceSMV is only available in a 32-bit implementation, we ran both it and NuSMV on this platform (with 32-bit compatible libraries) to ensure comparability of our performance results. Note that we ran CadenceSMV with the default settings whereas we always ran NuSMV with

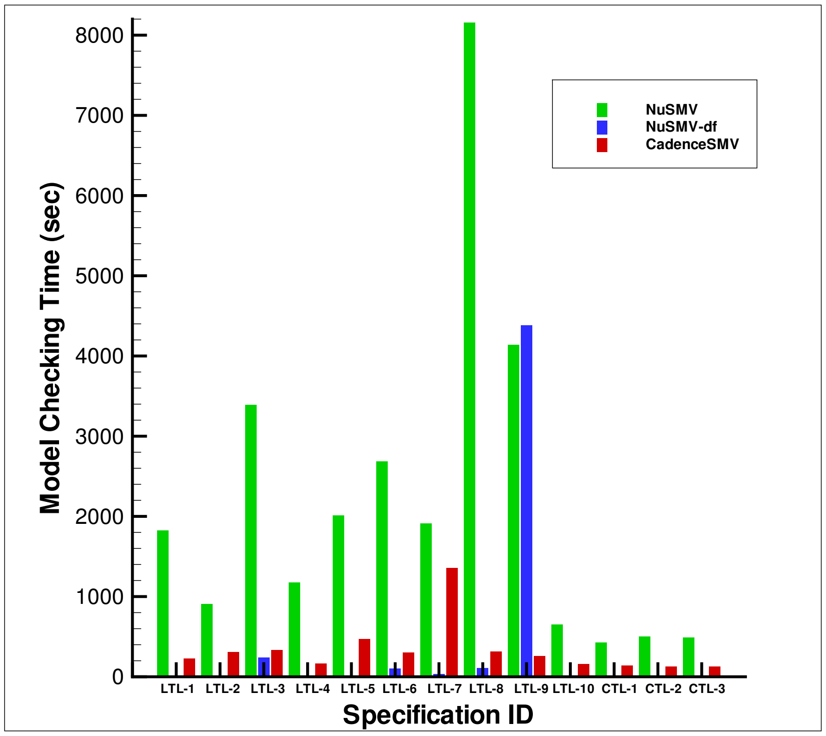

Figure 8: Runtimes, in seconds, for model checking each of the specifications against our original three-airplane model.

its `set dynamic_reorder` parameter enabled as earlier experiments demonstrated that this flag improved the performance of NuSMV on our models; `set dynamic_reorder` enables dynamic reordering of BDD variables in the CUDD package. On advice from the NuSMV team, we later ran NuSMV with the `-df` flag in order to improve its performance; this flag disables the computation of the set of reachable states, which are otherwise computed by default.

We present graphs of the performance results from model checking each of the 13 specifications in our test set against the original complete (not abstracted) system models (i.e. the models before we implemented the changes requested by system designers in response to our counterexamples). Thus in the performance results reported in this section, formulas LTL-8, and LTL-9 return counterexamples.

We created both complete and abstract models in both NuSMV and CadenceSMV for both three and four aircraft. We included four aircraft as a demonstration of scalability only, though there is always the potential that the system designers could change the pairwise resolution algorithms employed by the AutoResolver and TSAFE such that four-aircraft scenarios become relevant and in that case it would be helpful that we already have the capability to analyze scenarios of that size.

Figure 8 displays the total runtime, in seconds, as recorded by each model checker, to finish model checking the complete model on each specification. It is interesting that the latest version of NuSMV (released in 2011, compiled for 64-bit architecture) is much newer than CadenceSMV (released in 2002, compiled for 32-bit architecture) yet, surprisingly, CadenceSMV incurred faster model checking times for all 13 specifications in our test set. However, the use of the `-df` flag dramatically improved runtimes for NuSMV, as seen in Figure 8, taking NuSMV from being much slower than CadenceSMV for every specification without `-df` to being faster than CadenceSMV for all but one specification with the `-df` flag.

Figure 9 displays the maximum memory usage (in MB) recorded during model checking each of the 13 test specifications, measured with the Linux system command `ps`. We use this comparison rather than measuring the maximum number of BDD nodes allocated because the number of BDD nodes is dependent on the implementations of the two different BDD packages used; CadenceSMV has its own BDD package while NuSMV uses CUDD. The total number of BDD nodes allocated depends on how often garbage collection is done, variations due to variable ordering, quantification scheduling, and other factors that make comparison of this number between CadenceSMV and NuSMV an inaccurate representation of their relative memory efficiency [49].

It is interesting to note that, while CadenceSMV performs better in terms of time for all formulas than NuSMV without the `-df` flag, it requires more memory to check formula CTL-1 than NuSMV. With the `-df` flag, NuSMV uses less memory than CadenceSMV to check every specification except for LTL-9. The difference in performance (both time and memory) between CadenceSMV and NuSMV is notable since both CadenceSMV and NuSMV implement the same algorithm [2], just with differing optimizations i.e. for BDD variable ordering. Investigating this performance disparity further is an interesting topic for future research.

Neither CadenceSMV nor NuSMV without the `-df` flag was able to model check any of the 13 specifications for a system model with four planes within

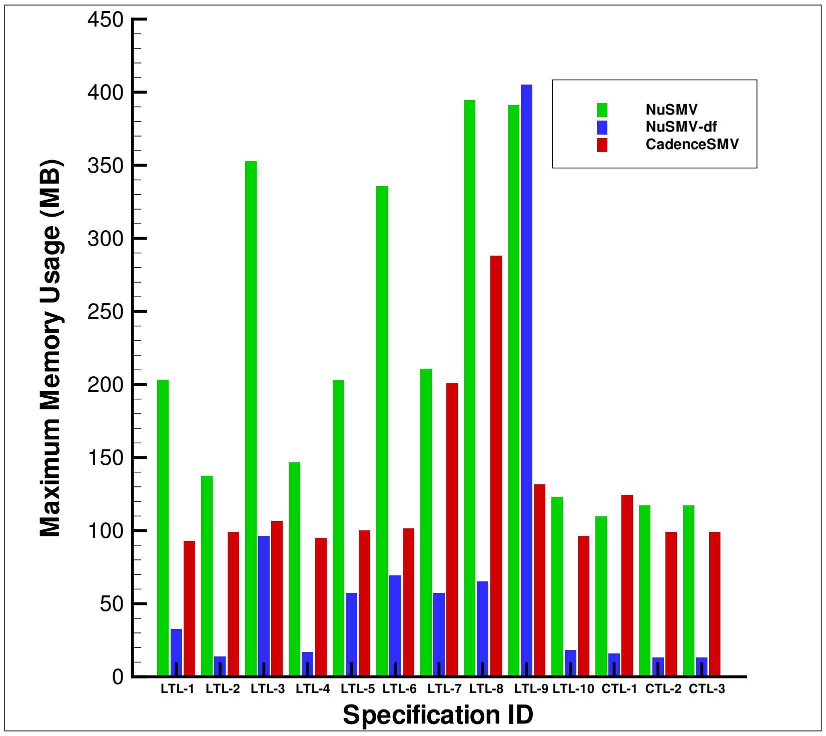

Figure 9: Maximum memory usage, in MB, for model checking each of the specifications against our original three-airplane model.

our machine time-out limit of 24 hours. However, running NuSMV with the `-df` flag enabled it to model check all but four of our specifications against our four-aircraft model, as seen in Figure 10. The accompanying memory usage for these runs is displayed in Figure 11. Note that two of the most difficult formulas to check were formulas LTL-8, and LTL-9, so alternative strategies to find their counterexamples faster are worthwhile to investigate. LTL-9 is of particular interest since it is the one formula for which adding the `-df` flag to NuSMV did not improve performance.

Further Optimizations. If we have a need in the future, there are several paths we could explore to enable efficiently model checking of a larger, e.g. four-aircraft, model. For models that may have counterexamples to the LTL specifications, bounded model checking (BMC) as a first step before full

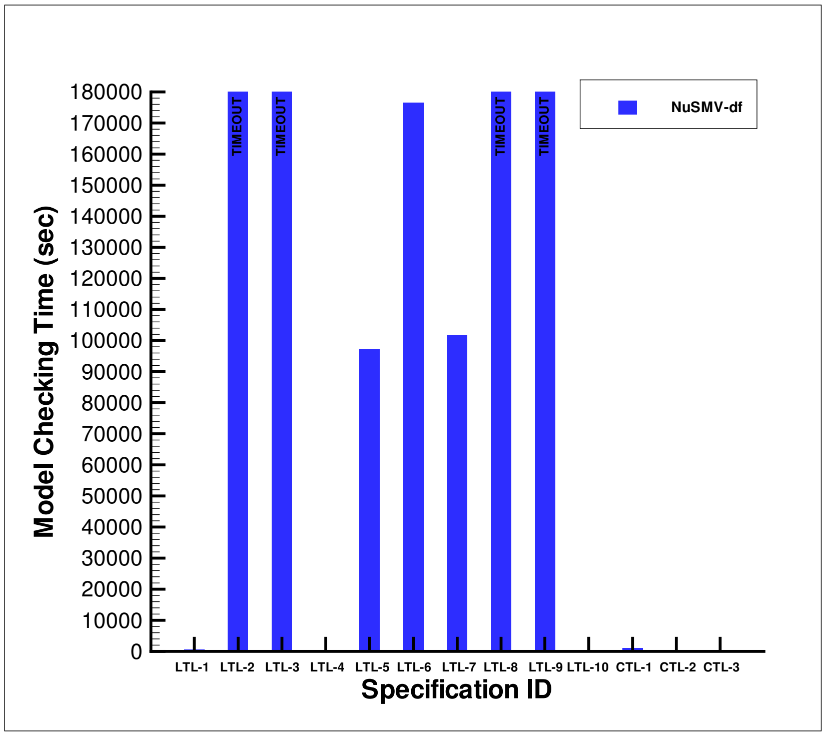

Figure 10: Runtimes, in seconds, for model checking each of the specifications against our four-airplane model using NuSMV 2.5.4 with the `set dynamic_reorder` and `-df` parameters.

model checking proved to be an efficient analysis strategy.[5] NuSMV has the ability to compute the reachable states; one can easily configure it with the command `./configure CFLAGS=-O3` and run it with the flags `-dynamic -df -int`. Then the `compute_reachable` command easily returns an FSM diameter of 16 for our three-aircraft model in 108.11 seconds and using approximately 79 MB of RAM. (Note that we used the SAT solver MiniSAT2-070721.) Using BMC with this FSM diameter for each of our LTL formulas (since it is not applicable for CTL) adds less than a second to the analysis time for each formula yet finds the two counterexamples in our three-aircraft model thus preventing us from having to run full model checking on these formulas. For example, the most time- and memory-consuming BMC run was

[5]Thanks to the NuSMV team for this suggestion.

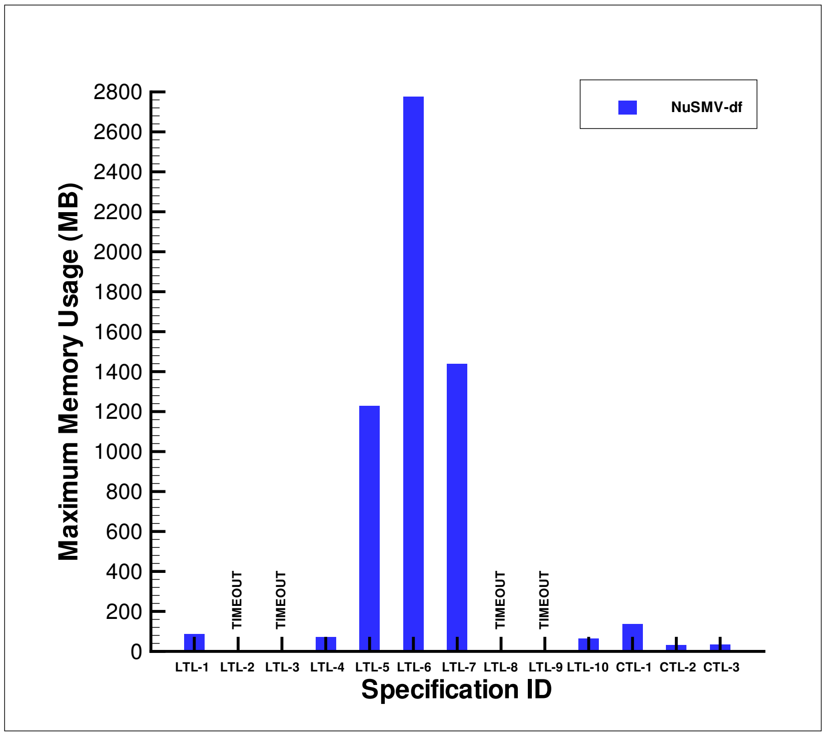

Figure 11: Maximum memory usage, in MB, for model checking each of the specifications against our four-airplane model using NuSMV 2.5.4 with the `set dynamic_reorder` and `-df` parameters.

for LTL-5, which took 0.89 seconds to complete and required about 32 MB of RAM. We had less success computing the FSM diameter for our four-aircraft model; this run timed out after 50 hours. However, we might proceed by making an educated guess for the k needed as input to BMC instead. Considering that two of the four formulas that experienced timeout, formulas LTL-8, and LTL-9, return counterexamples, finding these counterexamples first with BMC would have avoided these two timeouts. Other options for optimization include using alternative encodings for our properties, such as rewriting LTL to INVAR properties or using PANDA encodings, and experimenting with alternative advanced model checking techniques such as IC3 [50].

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 two sets 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.

Specification debugging has received little attention in verification in practice, and requires case-by-case treatment of each property, since no systematic approach is available. Yet it provided us with valuable insights in practice. 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, as does the utility of a *system formalization, model and specification debugging*, and *verification* trilogy for model checking real systems under development. We also compare the performance of NuSMV and CadenceSMV for completing our verification workload.

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 [51]. Previous work has reported on analyzing the safety of air traffic control systems using simulation [52] or fault trees [53]. 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.

Acknowledgments. Thanks to Heinz Erzberger, Todd Lauderdale, César Muñoz, and Russ Paielli for enlightening conversations on the Automated Airspace Concept and for insightful comments on earlier drafts of this paper.

Thank you to Kenneth L. McMillan for providing guidance on the CadenceSMV tool.

Thank you to the NuSMV team at Fondazione Bruno Kessler (FBK) for reproducing our results from the models and specifications on our website and sending us helpful advice on how to best utilize NuSMV 2.5.4 to efficiently complete our analysis. Specifically, thanks to the FBK team members Marco Bozzano, Alessandro Cimatti, Marco Gario, Cristian Mattarei, and Stefano Tonetta.

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.

References

[1] H. Erzberger, K. Heere, Algorithm and operational concept for resolving short-range conflicts, Proc. IMechE G J. Aerosp. Eng. 224 (2) (2010) 225–243. URL `http://pig.sagepub.com/content/224/2/225.abstract`

[2] J. Burch, E. Clarke, K. McMillan, D. Dill, L. Hwang, Symbolic model checking: 10^{20} states and beyond, Information and Computation 98 (2) (1992) 142–170.

[3] K. Rozier, M. Vardi, LTL satisfiability checking, STTT 12 (2) (2010) 123–137. URL `http://dx.doi.org/10.1007/s10009-010-0140-3`

[4] K. McMillan, The SMV language, Tech. rep., Cadence Berkeley Lab (1999).

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

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

[7] K. McMillan, Symbolic Model Checking, Kluwer Academic Publishers, 1993.

[8] A. C. R. Cavada, C. Jochim, G. Keighren, E. Olivetti, M. Pistore, M. Roveri, A. Tchaltsev, NuSMV 2.4 user manual, Tech. rep., CMU/ITC-irst (2005).

[9] K. L. McMillan, Getting started with SMV, Tech. rep., Cadence Berkeley Labs (1999).

[10] F. Raimondi, A. Lomuscio, M. J. Sergot, Towards model checking interpreted systems, in: FAABS02, LNAI 2699, Springer, 2002, pp. 115–125.

[11] M. Gribaudo, A. Horvath, A. Bobbio, E. Tronci, E. Ciancamerla, M. Minichino, Model-checking based on fluid Petri nets for the temperature control system of the ICARO co-generative plant, Tech. rep., SAFECOMP, 2434, LNCS (2002).

[12] A. Tribble, S. Miller, Software safety analysis of a flight management system vertical navigation function-a status report, in: DASC, 2003, pp. 1.B.1–1.1–9 v1.

[13] Y. Choi, M. Heimdahl, Model checking software requirement specifications using domain reduction abstraction, in: IEEE ASE, 2003, pp. 314–317.

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

[15] S. Miller, Will this be formal?, in: TPHOLs 5170, Springer, 2008, pp. 6–11.
URL `http://dx.doi.org/10.1007/978-3-540-71067-7_2`

[16] J. Yoo, E. Jee, S. Cha, Formal modeling and verification of safety-critical software, Software, IEEE 26 (3) (2009) 42–49.

[17] X. Gan, J. Dubrovin, K. Heljanko, A symbolic model checking approach to verifying satellite onboard software, Science of Computer Programming (2013).
URL `http://dx.doi.org/10.1016/j.scico.2013.03.005`

[18] J. Lahtinen, J. Valkonen, K. Björkman, J. Frits, I. Niemelä, K. Heljanko, Model checking of safety-critical software in the nuclear engineering domain, Reliability Engineering & System Safety 105 (0) (2012) 104 – 113.
URL `http://www.sciencedirect.com/science/article/pii/S0951832012000555`

[19] K. Rozier, Linear Temporal Logic Symbolic Model Checking, Computer Science Review Journal 5 (2) (2011) 163–203.
URL `http://dx.doi.org/10.1016/j.cosrev.2010.06.002`

[20] H. Erzberger, Automated conflict resolution for air traffic control, in: International Congress of the Aeronautical Sciences, 2004.

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

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

[23] T. L. Heinz Erzberger, Russ Paielli, Interviews with AAC System Designers, NASA Ames Research Center, Moffett Field, California (May-September 2011).

[24] K. Rozier, M. Vardi, A multi-encoding approach for LTL symbolic satisfiability checking, in: FM, Vol. 6664 of LNCS, Springer, 2011, pp. 417–431.

[25] K. Kähkönen, J. Lampinen, K. Heljanko, I. Niemelä, The LIME interface specification language and runtime monitoring tool., in: S. Bensalem, D. Peled (Eds.), RV, Vol. 5779 of Lecture Notes in Computer Science, Springer, 2009, pp. 93–100.

[26] D. Tabakov, K. Y. Rozier, M. Y. Vardi, Optimized temporal monitors for SystemC, Formal Methods in System Design 41 (3) (2012) 236–268.
URL http://www.springerlink.com/content/lq16400q55t72k16/

[27] J. Schumann, K. Y. Rozier, T. Reinbacher, O. J. Mengshoel, T. Mbaya, C. Ippolito, Towards real-time, on-board, hardware-supported sensor and software health management for unmanned aerial systems, in: Proceedings of the 2013 Annual Conference of the Prognostics and Health Management Society (PHM2013), 2013.

[28] T. Reinbacher, K. Y. Rozier, J. Schumann, Temporal-logic based runtime observer pairs for system health management of real-time systems, in: Proceedings of the 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Vol. 8413 of Lecture Notes in Computer Science (LNCS), Springer-Verlag, 2014, pp. 357–372.

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

[30] D. Giannakopoulou, D. Bushnell, J. Schumann, H. Erzberger, K. Heere, Formal testing for separation assurance, AMAI 63 (2011) 5–30.
URL http://dx.doi.org/10.1007/s10472-011-9224-3

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

[32] C. Muñoz, G. Dowek, V. Carreño, Modeling and verification of an air traffic concept of operations, in: ISSTA, ACM, 2004, pp. 175–182.
URL http://doi.acm.org/10.1145/1007512.1007536

[33] S. Miller, A. Tribble, M. Heimdahl, Proving the shalls, in: FME, Vol. 2805 of LNCS, Springer, 2003, pp. 75–93.
URL http://dx.doi.org/10.1007/978-3-540-45236-2_6

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

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

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

[37] O. Kupferman, Sanity checks in formal verification, in: CONCUR, Proc. 17th Int'l Conf., Vol. 4137 of Lecture Notes in Comp Sci, Springer, 2006, pp. 37–51.

[38] R. Bloem, R. Cavada, I. Pill, M. Roveri, A. Tchaltsev, RAT: A Tool for the Formal Analysis of Requirements, in: CAV, Vol. 4590, Springer, 2007, pp. 263–267.

[39] I. Pill, S. Semprini, R. Cavada, M. Roveri, R. Bloem, A. Cimatti, Formal analysis of hardware requirements, in: DAC '06, ACM, New York, NY, USA, 2006, pp. 821–826.

[40] R. Kurshan, FormalCheck User's Manual, Cadence Design, Inc., 1998.

[41] O. Kupferman, M. Vardi, Vacuity detection in temporal model checking, J. on Software Tools For Technology Transfer (STTT) 4 (2) (2003) 224–233.

[42] D. Fisman, O. Kupferman, S. Sheinvald-Faragy, M. Vardi, A framework for inherent vacuity, in: Haifa Verification Conference, LNCS 5394, Springer, 2008, pp. 7–22.

[43] I. Beer, S. Ben-David, C. Eisner, Y. Rodeh, Efficient detection of vacuity in ACTL formulas, Formal Methods in System Design 18 (2) (2001) 141–162.

[44] A. Pnueli, R. Rosner, On the synthesis of a reactive module, in: POPL, 1989, pp. 179–190.

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

[46] E. M. Clarke, O. Grumberg, D. E. Long, Model checking and abstraction, ACM Trans. Program. Lang. Syst. 16 (5) (1994) 1512–1542.

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

[48] E. Emerson, J. Halpern, Decision procedures and expressiveness in the temporal logic of branching time, J. of Computer and System Sci 30 (1985) 1–24.

[49] K. McMillan, Personal correspondence (March 2013).

[50] A. R. Bradley, SAT-based model checking without unrolling, in: VMCAI, 2011, pp. 70–87.

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

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

[53] J. Andrews, H. Erzberger, J. Welch, Safety analysis for advanced separation concepts, Air Traffic Control Quarterly 14 (1) (2006) pp. 5–24.