

Formal Verification of the NASA Runway Safety Monitor ^{*}

Radu I. Siminiceanu¹, Gianfranco Ciardo²

¹ National Institute of Aerospace, Hampton, VA 23666, e-mail: radu@nianet.org

² University of California, Riverside, CA 92521, e-mail: ciardo@cs.ucr.edu

The date of receipt and acceptance will be inserted by the editor

Abstract. The Runway Safety Monitor (RSM) designed by Lockheed Martin is part of NASA's effort to reduce aviation accidents. We developed a Petri net model of the RSM protocol and used the model checking functions of our tool SMART to investigate a number of safety properties for the RSM. To mitigate the impact of state-space explosion, we built a highly discretized model of the system, obtained by partitioning the monitored runway zone into a grid of smaller volumes and by considering scenarios involving only two aircraft. The model also assumes that there are no communication failures, such as bad input from radar or lack of incoming data, thus it relies on a consistent view of reality by all participants. In spite of these simplifications, we were able to expose potential problems in the conceptual design of RSM. Our findings were forwarded to the design engineers, who undertook corrective action. Additionally, the results stress the efficiency attained by the new model checking algorithms implemented in SMART, and demonstrate their applicability to real-world systems. Attempts to verify RSM with similar NuSMV and SPIN models have failed due to excessive memory consumption.

Key words: Formal verification, model checking, aviation safety, collision avoidance protocols

1 Introduction

As the avionics systems that are put into operation grow in complexity every year, an increasing share of the functionality in modern aircraft is shifted to computer-based,

automated devices. However, this rapid advance in sophistication makes the verification and certification of the deployed devices more difficult. This is due to the tremendous amounts of resources, measured in time, human expertise, and money, required for the analysis of complex systems.

The field of *formal methods* offers an alternative to traditional testing approaches that can explore only a limited number of scenarios. Formal verification uses rigorous mathematical techniques to *exhaustively* check that a model of the system satisfies a set of desired properties.

Model checking [13], which has gained increased popularity since the early 90s, is an automatic technique that relies on discovering the set of reachable states of the model and evaluating whether a given property, expressed in a *temporal logic*, is satisfied or not. The model is usually specified in a modeling language, such as automata, Petri nets, or pseudo-code, rather than using mathematical notation. If a temporal property holds, model checking attests it with 100% confidence. When a property does not hold, the model checking tool provides a counterexample, in the form of an execution path in the model, which can illustrate the source of the errors.

When using these computerized tools to verify modern protocols, the major obstacle is usually the *state-space explosion* phenomenon. As the size and complexity of a model increases, the size of the state-space (in number of reachable states) also increases, sometimes exponentially (in the number of components, variables, or parts of the system). Nevertheless, advances in model checking techniques, particularly in *symbolic* model checking [23], have made it possible to analyze systems with extremely large state spaces.

Model checking has been successful in the verification of complex, mostly synchronous, circuit designs, while verifying large asynchronous protocols and software has been generally viewed as more difficult. For the last several years, our research has targeted the class of *globally-*

^{*} Work supported in part by the National Aeronautics and Space Administration under grant NAG-1-02095 and by the National Science Foundation under grants CCR-0219745 and ACI-0203971.

asynchronous locally-synchronous systems [5], consisting of loosely coupled systems (homogeneous or heterogeneous) evolving somewhat independently of each other. Recently, NASA and Lockheed Martin have begun developing a protocol to detect runway incidents, called the Runway Safety Monitor (RSM) [16], which represents an excellent candidate for testing and evaluating our techniques. Although the verification of RSM was challenging and pushed our computational resources to the limit, we were able to discover several obscure scenarios that constitute potential hazards. Equally significant, however, is the fact that so few hazards were discovered overall, compared to the total number of reachable states, 6.7×10^{42} . This is strong evidence that RSM is robust and safe.

The rest of the paper is structured as follows. Sections 2 and 3 describe RSM and our tool SMART, which we used for this study. Section 4 gives the details of the RSM model we developed and Section 5 reports the results of our analysis. Finally, Section 6 summarizes our work and discusses ideas for future extensions.

2 The Runway Safety Monitor

The Runway Safety Monitor (RSM) is a component of NASA’s Runway Incursion Prevention System (RIPS) research [20]. Designed and implemented by Lockheed Martin engineers, RSM is intended to be incorporated in the Integrated Display System (IDS) [1], a suite of cockpit systems which NASA has been developing since 1993. IDS also includes other conflict detection and prevention algorithms, such as TCAS II [22]. The IDS design enables RSM to exploit existing data communications facilities, displays, Global Positioning System (GPS), ground surveillance system information, and data-links.

Collision avoidance protocols are already in operation. TCAS [22] has been in use since 1994 and is now required by the Federal Aviation Administration (FAA) on all commercial US aircraft. TCAS has a full formal specification, but it has been verified only partially, due to its complexity [4, 17].

The Small Aircraft Transportation System SATS [3], also under development at NASA Langley to help ensure safe landings of general aviation craft at towerless regional airports, has instead been formally verified [14].

Purpose of RSM. The goal of the Runway Safety Monitor is to detect *runway incursions*, defined by the FAA as “any occurrence at an airport involving an aircraft, vehicle, person, or object on the ground, that creates a collision hazard or results in the loss of separation with an aircraft taking off, intending to take off, landing, or intending to land.”

Since most air safety incidents occur on or near runways, the Runway Safety Monitor plays a key role in accident avoidance. RSM is not intended to *prevent* in-

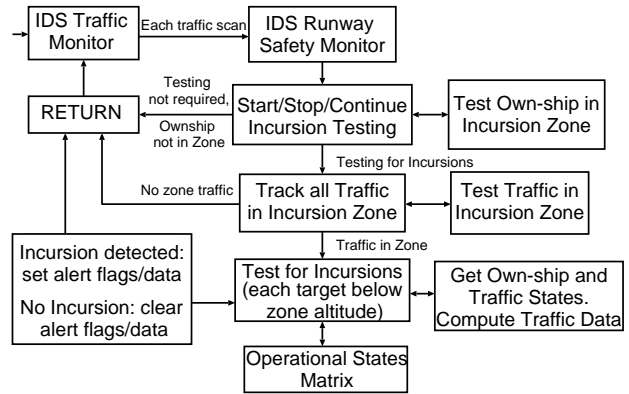


Fig. 1. RSM Algorithm top-level design.

cursions, but to *detect* them and *alert* the pilots. Prevention is provided by other components of RIPS in the form of a number of IDS capabilities such as heads-up display, electronic moving map, cockpit display of traffic information, and taxi routing. Experimental studies conducted by Lockheed Martin [16, 27] show that incursion situations are less likely to occur when IDS technology is employed on aircraft. RSM should greatly improve this positive effect.

RSM design. Figure 1 shows the high-level architecture of the RSM algorithm. RSM runs on a device installed in the cockpit and is activated prior to takeoff and landing procedures at airports. An independent copy of RSM runs on each aircraft and refers to the aircraft on which it is operating as *ownship* and to other aircraft, ground vehicles using the same runway, or even physical obstacles such as equipment, as *targets*.

RSM monitors traffic in a zone surrounding the runway where the takeoff or landing is to take place. The zone is a 3-D volume of space that runs up to 220 feet laterally from each edge of the runway, up to 400 feet of altitude above the runway, and 1.1 nautical miles from each runway end (the 400 feet altitude corresponds to a 3° glide slope for takeoff/landing trajectories).

The protocol, implemented as a C-language program, consists of a repeat-loop over three major phases. In the first phase, RSM gathers traffic information from radar updates received through a data-link. It identifies each target present in the monitored zone and stores its 3-D physical coordinates. The frequency of the updates may not be constant, updates can be lost, and data might even be faulty. The implications of data-link errors or omissions are not addressed in this study, but present a challenging task for future study. These errors have already been the subject of some experimental measurements [27], and their analysis calls for a *stochastic* flavor not captured in our present model, which is instead concerned only with *logical* errors.

In the second phase, the algorithm assigns a *status* to each target, from a predetermined set of values that includes taxi, pre-takeoff, takeoff, climb out, landing, roll

out, and fly-through modes. We discuss in detail the meaning of these states when we describe our model of the system.

The third phase is responsible for detecting incursions, and is performed for each target based on the spatial attributes (position, heading, acceleration) of ownship and target, plus some logical conditions. Table 2, discussed later, shows the *operational state matrix* of this phase. Our analysis focuses on verifying that this decision procedure is able to detect all possible incursion scenarios, or on finding possible incursion scenarios where RSM fails to raise an alarm.

3 Overview of the SMART tool

To model the Runway Safety Monitor, we employ our tool SMART (the Stochastic and Model checking Analyzer for Reliability and Timing) [7], which we developed for the logical and stochastic analysis of structured systems. Given a formal description of a system as a Petri net, SMART can generate the state-space, verify temporal-logic properties, and provide efficient numerical solutions for timing and stochastic analysis. SMART has several advantages over most other model checkers:

- Compact storage for states with Multiway Decision Diagrams (MDDs) [24], a generalized version of the Binary Decision Diagrams (BDDs [2]) for multi-valued variables.
- Extremely compact encoding of the transition relation between states with Kronecker matrices [9].
- Efficient symbolic state-space exploration algorithms based on *saturation* [8], a novel fixed-point iteration strategy.
- Fast generation of counterexamples, based on Edge-Valued MDDs (EVMDDs) [10].

The SMART input is a Petri net with Turing-equivalent extensions (immediate transitions and marking dependent arc cardinalities) [26], required to have a finite state-space. Each SMART input file defines one or more structured (i.e., partitioned into submodels) event-based models. A model can be parameterized and defines a set of *measures*, which, in our case, can be thought of as logical queries to be evaluated by systematic state exploration.

SMART implements a wide range of explicit as well as implicit exploration methods. Use of the most advanced techniques requires a *partition* of the model to exploit the system structure. A partition of the Petri net model into K submodels is equivalent to partitioning its places (representing the system variables) into subnets. Subsequently, a system state (a marking of the places with tokens) can be written as the concatenation of K local states (submarkings) and thus be encoded as an MDD. In particular, a partition is *Kronecker-consistent* if any global system behavior can be expressed as a functional product of local behaviors for each subsystem. From a

logical point of view, for example, an event in the model is globally enabled if it is locally enabled in each of the K submodels in isolation. Similar consistency requirements can be defined for transition guard expressions or, from a stochastic point of view, for transition firing rates (but only the logical interpretation of Kronecker consistency is needed in this study). A more detailed discussion of the implications of consistency requirements follows in Section 4.

A comprehensive SMART User Manual is available online [6].

There are several reasons for using a tool designed for the analysis of discrete-state systems to model and verify an embedded (hybrid) system. Even though there exist tools for the verification of hybrid systems, such as HyTech [18], their focus is on the integration of discrete and continuous aspects of the systems. The discrete aspects of a large application like RSM are well beyond the scope of the state-of-the-art hybrid model checkers. Moreover, it is clear that the actual RSM algorithm is implicitly using a discretized view of time: the radar updates are not fed continuously to the RSM device, but only at a given frequency (nominally, at every 0.5 seconds). This suggests that an abstraction scheme for the other continuous-type variables (location and speed of aircraft) is adequate in this case. Last but not least, the discretized RSM model belongs to the class of globally asynchronous/locally synchronous systems, for which the saturation-based algorithms for analysis implemented in SMART excel.

4 The SMART model of RSM

To model RSM, we first identify the variables representing the system state and the events describing the potential state-to-state transitions. Then, we translate this information into a Petri net for input into SMART. We partition the model into $n + 1$ submodels where n is the number of targets moving inside the zone. The variables of the first submodel (indexed 0) describe the state of ownship. The variables of the other n submodels describe the state of each target. For submodel i , $0 \leq i \leq n$, the relevant attributes are:

- Location:** a 3-D vector (x_i, y_i, z_i) , where the X -axis is across the width of the runway, the Y -axis is along the length, and the Z -axis is on the vertical.
- Speed and heading:** a second 3-D vector (vx_i, vy_i, vz_i) .
- Acceleration along the runway:** ay_i .
- Status:** an enumerated type variable, $status_i$.
- Alarm flag:** a boolean variable, $alarm_i$.
- Phase:** an integer variable, $phase_i$.

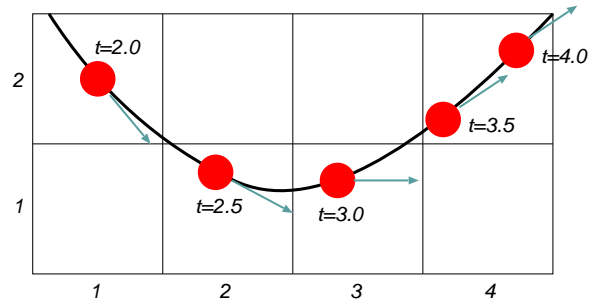
All other variables are deemed irrelevant to our study and can be abstracted away from the model to reduce the size of its state space.

As mentioned earlier, SMART requires a partitioning of the model variables in order to apply the most advanced symbolic model checking techniques. A natural choice is to group variables referring to the same target together. However, assigning *all* variables to the same partition leads to extremely large “local” state spaces for each submodel, which is unacceptable. A better choice is to further split the subnets into even smaller ones. We arranged the variables in $n + 1$ clusters, as follows:

$$\left\{ \begin{array}{l} \text{subnet } 5i + 1 : \textit{phase}_i \\ \text{subnet } 5i + 2 : \textit{status}_i, \textit{alarm}_i \\ \text{subnet } 5i + 3 : \textit{x}_i, \textit{vx}_i \\ \text{subnet } 5i + 4 : \textit{y}_i, \textit{vy}_i, \textit{ay}_i \\ \text{subnet } 5i + 5 : \textit{z}_i, \textit{vz}_i \end{array} \right.$$

Domain of the state variables. Since SMART operates on discrete-type systems, abstraction by discretization is necessary to cope with the continuous-type variables of the RSM algorithm. To come up with a good representation of the variable domains, we start with the roughest possible discretization that can be extracted from the protocol specifications, and then manually refine it further as needed. We had to take into consideration a balanced solution between a very rough discretization (which potentially hides too many meaningful behaviors by merging distinct states into a single representative), and a discretization that is too fine for an efficient state-space generation (which prevents analysis due to the state-space explosion problem). In the end, we chose the following domains (the subscript i is omitted for clarity):

- The coordinates x, y, z could be as simple as $x, y, z \in \{0, 1, 2\}$, where 0 means “out of the monitored zone”, 1 means “in the vicinity”, and 2 means “on the runway”. However, we chose a finer parametric representation: $x \in \{0, \dots, \textit{max}_x\}$, $y \in \{0, \dots, \textit{max}_y\}$, and $z \in \{0, \dots, \textit{max}_z\}$, where 0 means outside the zone, and the constants \textit{max}_x , \textit{max}_y , and \textit{max}_z can be adjusted to the modeler’s preference. In other words, location $(0, 0, 0)$ represents all positions outside the zone. A target that exits the zone, or has not yet entered it, is assigned this location. As an alternative, we could have used an “outer layer” of locations surrounding the monitored zone, but this would unnecessarily increase the state space with entries of the form $(0, y, z)$, $(x, 0, z)$, and $(x, y, 0)$, all representing the same circumstance: the target is not being monitored.
- The speed values $\textit{vx}, \textit{vy}, \textit{vz}$ could be assigned the domain $\{0, \pm 1, \pm 2\}$, where 0 means not moving, ± 1 means moving slowly (below the predetermined *taxi speed threshold* TS of 45 knots), and ± 2 means moving fast (above TS). Again, a better representation is $\textit{vx}, \textit{vy}, \textit{vz} \in \{-\textit{max}_{speed}, \dots, 0, \dots, \textit{max}_{speed}\}$, using another parameter \textit{max}_{speed} .



Real trajectory: (51.5, 161.3), (128.5, 93.6), (220.1, 80.3), (318.5, 111.2), ...
Discretized trajectory: (1,2), (2,1), (3,1), (4,2), ...
Discretized speed: (+1,-1), (+1,-1), (+1,0), (+1,+1), ...

Fig. 2. Example projection of a continuous trajectory in 2-D.

Since, in Petri nets, places cannot hold a negative number of tokens, we have to offset the values of the speed variables by $-\textit{max}_{speed}$.

- The acceleration a_y has only two relevant values: non-negative or strictly negative.
- The *status* is one of $\{\textit{out}, \textit{taxi}, \textit{takeoff}, \textit{climb}, \textit{land}, \textit{rollout}, \textit{flythru}\}$.
- The *phase* is one of $\{\textit{radar_update}, \textit{set_status}, \textit{detect}\}$.

The variable *phase* works like a program counter for the execution of the algorithm on each participant, which loops through three steps:

- A. Update location of targets ($\textit{phase} = \textit{radar_update}$).
- B. Update status of targets ($\textit{phase} = \textit{set_status}$).
- C. Set or reset alarm ($\textit{phase} = \textit{detect}$).

We next discuss in detail the modeling decisions that were taken for each of these three steps.

A. The 3-D motion of targets. Our discretization divides the monitored space into a number of volumes arranged in a 3-D grid. As a result, the possible positions of the aircraft are identified by a finite number of grid cells, from the discrete domain $(0, 0, 0) \cup \{1, \dots, \textit{max}_x\} \times \{1, \dots, \textit{max}_y\} \times \{1, \dots, \textit{max}_z\}$. Similarly, continuous trajectories have to be represented by abstract, discretized trajectories through the cells of the 3-D grid. This is a reasonable compromise when modeling continuous variables with discrete-state approaches. Regarding the possible trajectories allowed in the model, there are three alternatives to be considered.

First is the projection method, which assigns to every continuous trajectory its corresponding discrete path in the grid. An example of such a projection is given in Figure 2 (in a 2-D space, for the sake of readability). The grid cells in the figure have the size of 100 units (in feet), and the snapshots are taken after each 0.5 time units (in seconds). The speed units are measured in the number axis divisions traveled after each update. The major challenge in putting in practice this method is the difficulty in discerning between physical possibilities

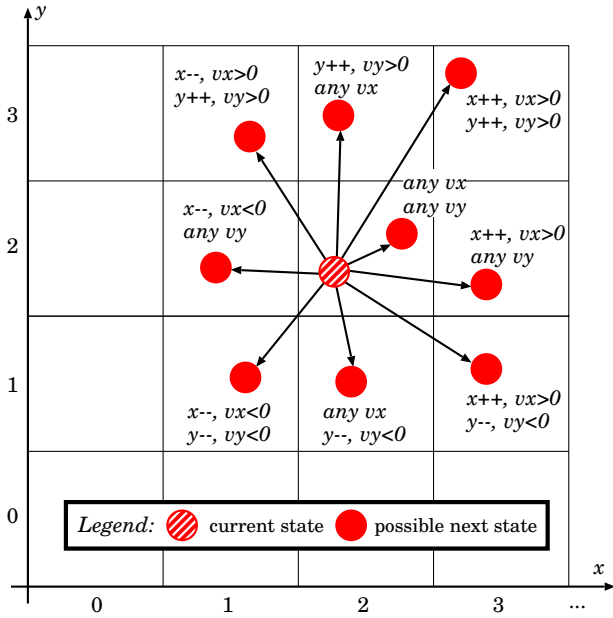


Fig. 3. Possible 2-D movements of a target (“free-motion” model).

and impossibilities. There is no efficient way of ruling out all anomalies. For example, a target could change its real location, while its discretized location might not. The dependency between the speed and the number of time units a target may remain in one grid cell is also very difficult to establish: it could be one move (at high speed), or more (at low speed), but no upper bound on the number of time units allowed within one grid cell can be computed in the discretized model.

Therefore, we have considered a different approach to modeling the motion of targets, that proved to be more practical. One alternative allows nearly free movement of a target, in the sense that a move to an adjacent cell is always allowed. In principle, a target is free to remain in the current cell or to move to any of the neighboring 26 cells, corresponding to a nondeterministic decrease, no change, or increase in the coordinates x , y , and z . However, the changes must be consistent with the heading. This allows for almost random movements. On the one hand, the restriction to allow transitions *only* between adjacent cells excludes a large number of trajectories, most of which are truly physically impossible. On the other hand, we have to argue that no realistic trajectory is excluded by the model. This is indeed true when the cell size is large (corresponding to a “rough” discretization of the space, into a small number of cells). In our simplest model, which captured all the interesting properties, the size of a grid cell is 900 feet. Given that the location updates arrive on the data-link every 0.5 seconds, a target can skip a grid cell and move to a cell two discrete positions away only if traveling at speeds exceeding $1800 \text{ ft/sec} \approx 1227 \text{ mph}$ (or $\approx 1975 \text{ km/h}$). This is over 1.6 times the speed of sound. Although it is not entirely safe to assume that these speeds are not encoun-

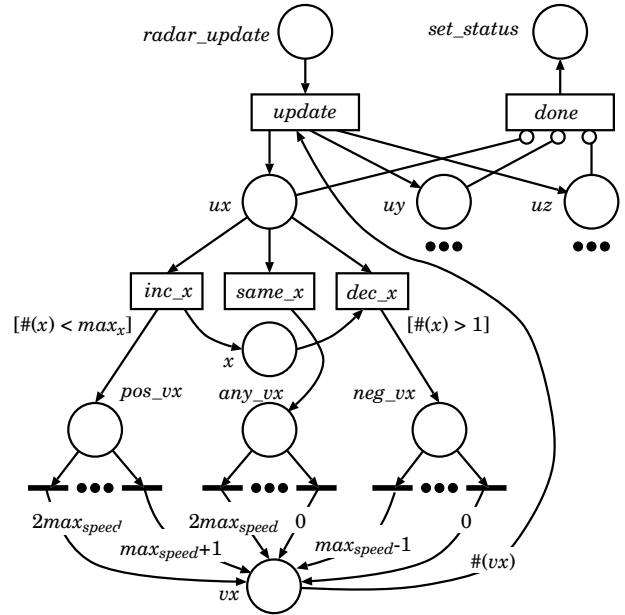


Fig. 4. Subnet for updating the variables x and vx in each sub-model

tered at civil airports, their exclusion from our model is reasonable and helps simplify the analysis. Moreover, a rough discretization also serves the purpose of mitigating the state-space explosion problem, as the number of possible states becomes manageable. Figure 3 shows the possible moves of a target in this second model (also in a 2-D space, for clarity).

To achieve the non-deterministic choice, we model the 3-D motion not via 27 concurrent and mutually exclusive events, but rather via the composition of a non-deterministic choice to decrease, not change, or increase each coordinate. More precisely, the next position is computed as the composite effect of firing three concurrent and independent events, non-deterministically chosen from among a set of three mutually exclusive options for each axis, for a total of just nine events. The update of the coordinate x_i and speed component vx_i for target i is modeled by the subnet shown in Figure 4 (updates for the y and z coordinates are analogous, they are triggered by the arrival tokens in places uy and uz , respectively). Petri net places are drawn as circles, transitions as rectangles, and immediate transitions as thick bars. Transitions inc_x and dec_x have associated guard expressions, and arc cardinalities (other than the default value 1) are shown on each arc. Note that $\#(a)$ indicates the current number of tokens in place a , thus the effect of the arc from place vx to transition $update$ is to reset the old value of vx in preparation for its new setting. The three arrays of immediate transitions are needed to assign a value to vx_i : random positive ($1 \leq vx_i \leq max_speed$, i.e., from $max_speed + 1$ to $2max_speed$ tokens), any value ($-max_speed \leq vx_i \leq max_speed$, i.e., from 0 to $2max_speed$

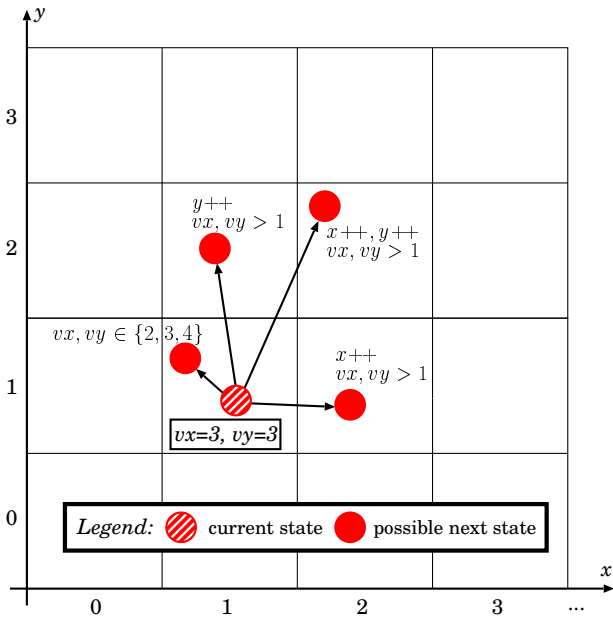


Fig. 5. Possible movements from a state satisfying $vx = 3, vy = 3$ (“restricted” model).

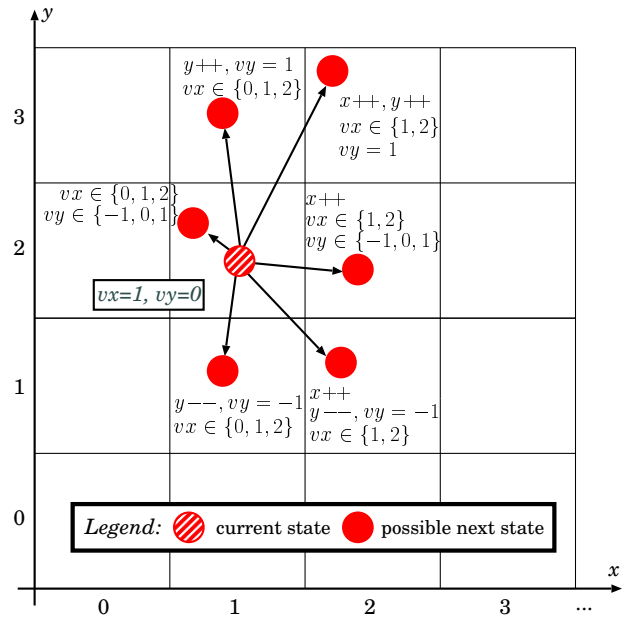


Fig. 6. Possible movements from a state satisfying $vx = 1, vy = 0$ (“restricted” model).

tokens), or random negative ($-max_{speed} \leq vx_i \leq -1$, i.e., from 0 to $max_{speed} - 1$ tokens).

Also, when a target enters the zone, its position is nondeterministically chosen on the frontier of the monitored volume, i.e., $x \in \{1, max_x\}$, $y \in \{1, max_y\}$, or $z = max_z$ (but not $z = 1$, since no entry is possible from below ground). The entry speed parameters are also chosen nondeterministically, but consistently with the direction of entry. For example, a target cannot enter from the left with a negative vx .

This second model might still include unrealistic trajectories. Examples of typically abnormal behaviors allowed in the model are: oscillating back and forth between two adjacent squares (when the corresponding speed components alternate from positive to negative and back) or staying forever in one square, even with a positive speed. This is still acceptable in the verification process as long as the model covers all realistic behaviors. If a property holds globally in the abstract model, then it will also hold in the realistic model. However, if a property does not hold globally, we must check the corresponding counterexample generated by SMART to determine whether it represents a realistic scenario.

If a more thorough elimination of unwanted trajectories is desired, a third alternative that forbids abrupt variations in speed can be considered. In other words, both the coordinates x, y, z and the speed components vx, vy, vz can change by at most one in absolute value. This further restriction can be achieved by allowing only the increase, decrease, and no change of speed at each timestep, together with a consistent update of the coordinates: for example, the variable x cannot be decreasing when the speed component vx is non-negative.

In comparison to the free-motion model, Figure 5 shows the possible next states (in 2-D space) for a target whose speed components are $vx = 3$ and $vy = 3$ in the current state. In this case, only four new locations are possible, corresponding to the no change or increase in x and, independently, y . The reduced number of choices is due to the strictly positive value of the speed, which does not allow any move in the negative axis direction. Only when one speed component is 0 in the current state, the target can move in both directions of the corresponding axis, as seen in Figure 6. In this model, at least two steps are required to go from positive to negative speed (and vice versa). This implies that “zigzagging” is not possible, a fact that could have a significant importance in the analysis, as seen in Section 5.

We implemented both versions, with free or restricted movement between adjacent cells, in SMART.

B. Status definitions. In the second phase of the execution loop, the status variable of each aircraft is deterministically updated using the other state information. In our model, the status values are:

- out*: not in the monitored zone
 $\equiv (x = 0) \wedge (y = 0) \wedge (z = 0)$
- taxi*: on the ground, either at low speed or not with a runway heading
 $\equiv (z = 1) \wedge ((|vx| \leq TS \wedge |vy| \leq TS) \vee (vx \neq 0))$
- takeoff*: on the ground, with a runway heading, accelerating
 $\equiv (z = 1) \wedge (|vy| > TS) \wedge (vx = 0) \wedge (a_y \geq 0)$
- rollout*: on the ground, with a runway heading, decelerating
 $\equiv (z = 1) \wedge (|vy| > TS) \wedge (vx = 0) \wedge (a_y < 0)$

↓ grid size <i>max_speed</i> ↓	Number of states in the model					SS gen. time (sec)				SS gen. memory (MBytes)			
	1 tar.	2 tar.	3 tar.	4 tar.	1 tar.	2 tar.	3 tar.	4 tar.	1 tar.	2 tar.	3 tar.	4 tar.	
3×5×3	2	1.0×10 ¹³	3.4×10 ¹⁹	1.1×10 ²⁶	3.5×10 ³²	2.01	2.93	3.93	4.91	2.00	3.00	4.00	4.99
5×10×5	2	4.1×10 ¹⁴	8.4×10 ²¹	1.7×10 ²⁹	3.5×10 ³⁶	5.52	8.27	11.19	13.91	7.21	10.81	14.41	18.01
10×10×10	2	7.6×10 ¹⁵	6.6×10 ²³	5.8×10 ³¹	5.0×10 ³⁹	13.62	20.58	27.50	34.42	20.86	31.29	41.72	52.15
3×5×3	5	2.7×10 ¹⁴	4.4×10 ²¹	7.2×10 ²⁸	1.2×10 ³⁶	4.41	6.51	8.77	10.98	4.22	6.33	8.44	10.55
5×10×5	5	8.3×10 ¹⁵	7.6×10 ²³	6.9×10 ³¹	6.3×10 ³⁹	12.91	19.07	25.42	32.05	15.48	23.21	30.95	38.69
10×10×10	5	1.4×10 ¹⁷	5.0×10 ²⁵	1.8×10 ³⁴	6.7×10 ⁴²	28.45	42.84	57.25	71.75	39.73	59.59	79.45	99.31

Table 1. State-space generation results for our model, for 1, 2, 3, or 4 targets and as a function of $max_x \times max_y \times max_z$ and max_speed .

climbout: airborne, with a runway heading, strictly positive vertical speed

$$\equiv (z > 1) \wedge (vx = 0) \wedge (vz > 0)$$

land: airborne, with a runway heading, negative vertical speed

$$\equiv (z > 1) \wedge (vx = 0) \wedge (vz \leq 0)$$

flythru: airborne, not in *climbout* or *land* mode

$$\equiv (z > 1) \wedge (vx \neq 0)$$

The predicates $z = 1$ and $z > 1$ used above also imply $x > 0$ and $y > 0$, by the way we designed the non-monitored zone to be represented by a single cell, not by a rim of states. Also, the acceleration a_y , needed to discern between *takeoff* and *rollout* status, does not need to be modeled directly, since its value is computed on the spot based on the variation of the variable vy .

The partial model constructed so far can be used as a building block for further analysis, since it captures the free movement of targets in 3-D space (phase one of RSM) and the target status assignments (phase two of RSM). This model exhibits strong event locality, i.e., each event depends and affects only a few levels; this is an essential property exploited by the saturation algorithm we employ. To evaluate its complexity, we collected measurements of the state spaces generated for different input parameters of this core model: number of targets, n , grid size max_x , max_y , and max_z , and speed thresholds, max_speed . The state-space size, runtime, and memory consumption are listed in Table 1. The results show that the state space can be generated for multiple targets, a fairly large size of grid, and multiple thresholds of speed, in a few minutes using under 100 MBytes.

C. Setting the alarm. The third and most important phase of the RSM algorithm is setting the alarm flag for every target. In pseudo-code, this corresponds to a single variable assignment statement: set the (boolean) value of each $alarm_i$ based on different combinations of the current values of the other variables, as listed in the operational state matrix of Table 2. We can either model the third phase directly, by adding transitions to the Petri net, or define queries that use a combination of status and position variables to determine whether the alarm would have been set correctly. We choose the former approach.

Target → Ownship ↓	<i>taxi</i>	<i>takeoff</i>	<i>climbout</i>	<i>land</i>	<i>rollout</i>	<i>flythru</i>
<i>taxi</i>	—	$a \wedge f$	$a \wedge f$	$a \wedge f$	$a \wedge c \wedge f$	—
<i>takeoff</i>	$a \wedge f$	$d \vee e$	$d \vee e$	$d \vee e$	$a \vee d$	$b \wedge c$
<i>climbout</i>	$a \wedge f$	$d \vee e$	$d \vee e$	$d \vee e$	$d \vee e$	$b \wedge c$
<i>land</i>	$a \wedge f$	$d \vee e$	$d \vee e$	$d \vee e$	$a \vee d$	$b \wedge c$
<i>rollout</i>	$a \wedge c \wedge f$	$a \vee d$	$a \vee d$	$a \vee d$	$d \vee e$	$b \wedge c$
<i>flythru</i>	—	$b \wedge c$	$b \wedge c$	$b \wedge c$	$b \wedge c$	—

a: Distance closing

b: In takeoff/landing path

c: Distance less than min. sep.

d: Takeoff/landing in same direction, less than min. sep.

e: Takeoff/landing in opposite direction, closing

f: Taxi/stationary on or near runway

Table 2. Operational state matrix for setting the alarm.

Modeling this rather complex assignment statement in a Petri net is difficult because of two factors. First, predicates such as “distance is closing” or “in the takeoff path” potentially involve geometry and linear equations and are difficult to express in a discretized model. However, certain factors help make our task easier: the designers kept the concepts simple and trigonometry can be circumvented on a case by case basis. For example, “distance to target i is closing” should normally be evaluated by comparing the value of the expression $\sqrt{(x_0 - x_i)^2 + (y_0 - y_i)^2 + (z_0 - z_i)^2}$ in the current and previous states. This could further imply that the previous location of each target should be stored in a set of auxiliary variables, say $oldx_i, oldy_i, oldz_i$, further increasing the state space. However, this can be avoided by exploiting the information derived from each aircraft’s status. For example, if ownship is taxiing and target i is taking off, we know that $z_0 = 1, vx_0, vy_0 \leq TS, z_i = 1, vx_i = 0, |vy_i| > TS$, and $vz_0 = vz_i = 0$, i.e., the target is on the ground, lined up with the runway and moving faster than the taxi speed limit. For the distance to be closing, it is enough for ownship to be in front of the target, depending on which direction this is moving. Hence, in this situation, the predicate can be expressed as

$$a \equiv (vy_i > 0 \wedge y_0 > y_i) \vee (vy_i < 0 \wedge y_0 < y_i).$$

$max_speed \rightarrow$	2	3	4	5
\downarrow grid size	State-space generation time (sec.)			
$3 \times 5 \times 4$	75.92	105.17	179.28	252.25
$3 \times 7 \times 4$	195.54	324.65	604.23	805.95
$3 \times 10 \times 5$	995.18	2212.24	4668.55	7348.27
$5 \times 10 \times 7$	48257.3	-	-	-
	Memory consumption (MB)			
$3 \times 5 \times 4$	11.19	21.20	32.58	49.39
$3 \times 7 \times 4$	18.27	36.02	56.91	87.25
$3 \times 10 \times 5$	42.59	83.53	138.56	218.85
$5 \times 10 \times 7$	246.22	-	-	-

Table 3. Results from state-space generation on the complete model.

We can similarly express the other predicates as follows:

$$\begin{aligned}
 b \wedge c &\equiv (vy_0 > 0 \wedge y_0 \leq y_i \leq y_0 + 1 \wedge |x_0 - x_i| \leq 1 \wedge z_i \leq 2) \\
 &\quad \vee (vy_0 < 0 \wedge y_0 - 1 \leq y_i \leq y_0 \wedge |x_0 - x_i| \leq 1 \wedge z_i \leq 2) \\
 d &\equiv vy_0 \cdot vy_i > 0 \wedge |x_0 - x_i| \leq 1 \wedge |y_0 - y_i| \leq 1 \\
 e &\equiv (vy_0 > 0 \wedge vy_i < 0 \wedge y_i \geq y_0) \vee \\
 &\quad (vy_0 < 0 \wedge vy_i > 0 \wedge y_i \leq y_0) \\
 f &\equiv 1 < x_i < max_x,
 \end{aligned}$$

where the above example formulae are derived for the following pairs of states, respectively: $b \wedge c$ for takeoff-flythru, d and e for takeoff-takeoff.

The roughness of the discretization can also help simplify the model. If $max_x = 3$ (which is a reasonable assumption given that there is usually no room for two aircraft on the runway side-by-side, anyway), then $x_i = 2$ for any aircraft taking off or landing. In this case, the predicate $|x_0 - x_i| \leq 1$ is equivalent to $1 \leq x_0 \leq 3$, which is always true when ownship is not out.

Kronecker consistency requirements. A second challenge in modeling the third phase is that the Kronecker consistency requirements force us to split events into multiple finer-grain events. For example, the predicate “target i is in takeoff/landing path of ownship” can be expressed as:

$$(x_i = x_0) \wedge (((vy_0 > 0) \wedge (y_i > y_0)) \vee ((vy_0 < 0) \wedge (y_i < y_0))).$$

However, since variables y_i and y_0 are described by different local states of the model, each term involving the two must be split to satisfy Kronecker consistency, by domain enumeration:

$$\bigvee_{1 \leq C_y \leq max_y} ((x_i = x_0) \wedge ((vy_0 > 0 \wedge y_i = C_y \wedge C_y > y_0) \vee (vy_0 < 0 \wedge y_i = C_y \wedge C_y < y_0)))$$

The same procedure must be applied to x_0 and x_i , by further splitting terms: $\bigvee_{1 \leq C_x \leq max_x} \bigvee_{1 \leq C_y \leq max_y}$

$$\begin{aligned}
 &((x_i = C_x) \wedge (x_0 = C_x) \wedge (vy_0 > 0) \wedge (y_i = C_y) \wedge (C_y > y_0)) \vee \\
 &((x_i = C_x) \wedge (x_0 = C_x) \wedge (vy_0 < 0) \wedge (y_i = C_y) \wedge (C_y < y_0))
 \end{aligned}$$

This generates $2 \cdot max_x \cdot max_y$ events from a single original event, one for each term of the disjunction. The split

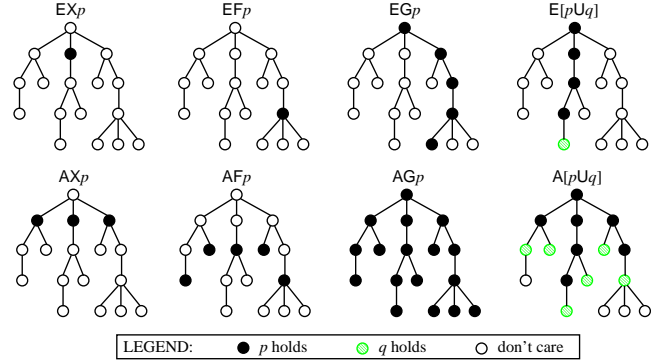


Fig. 7. The semantics of CTL operators.

events require over 2000 lines of additional SMART code, compared to just 500 lines needed to model the first two phases. At the end of this process, the most significant change in the model was a severe loss of event locality, leading to a slowdown in generation time and, most importantly, a much higher memory consumption. The peak MDD size increased to over 1000 times larger than the final, causing the SMART model checker to run out of memory for large parameters, including multiple targets. However, we were still able to build the state space for one target and a medium size of the grid, within 1GB of memory and in less than five minutes. This was enough to expose several potential problems with the decision procedure of the protocol. Table 3 shows the state-space measurements on this final SMART model with just one target. Missing entries in the table correspond to parameter choices that required excessive runtime or memory.

To compare our saturation technique with other existing methods, we tried using the symbolic model checker NuSMV [11] and the explicit model checker SPIN [19] on equivalent models of RSM. However, neither tool was able to construct the state-space successfully. NuSMV runs out of memory even before starting the generation, as the BDD encoding of the transition relation is too large, while SPIN explores a very small fraction of the state-space (less than $1/10^6$, even when using partial order reduction) to be able to expose any problem. We need to mention here that other techniques, such as compositional state-space minimization (CADP/CWB) [15] or probabilistic model checking (PRISM) [21], have also been successfully used to analyze large state-spaces, but time constraints prevented their analysis in this project.

5 Model checking RSM

Model checking is concerned with verifying temporal logic properties of discrete-state systems evolving in time.

SMART implements the branching time *Computation Tree Logic* (CTL) [12], widely used in practice due to its simple yet expressive syntax. In CTL, operators occur in pairs: the path quantifier, either A (on all future paths)

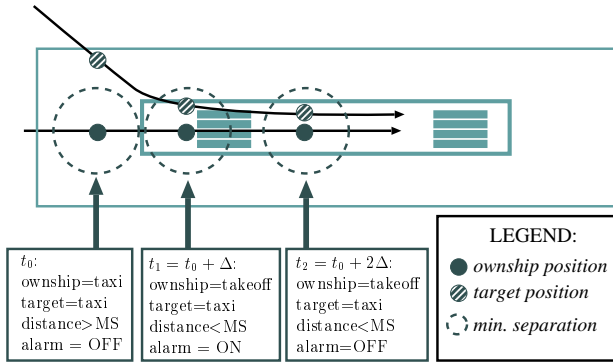


Fig. 8. Scenario for the memory-less property (ground level).

or E (there exists a path), is followed by the tense operator, one of X (next), F (future, finally), G (globally, generally), and U (until). Their semantics is informally shown in Fig. 7, where system states are depicted as the nodes of the trees and arcs represent transitions between states, so that a node precedes in temporal order the nodes it can reach. In each case, the root node is labeled with a CTL formula it satisfies.

Notation and formal definitions. The operational state matrix in Table 2 lists the alarm setting criteria, as given in the documentation of the RSM algorithm [16]. Our study aims at exhaustively checking whether this operational matrix is able to detect all incursion scenarios. A situation where two aircraft get too close to each other (within the minimum separation distance of 900 feet) without the alarm variable having been set is from now on called a *missed alarm* scenario. The following predicates are used to describe properties of interest (for sake of clarity, subscripts o and t refer to ownship and target, respectively):

$$\begin{aligned} detect &\equiv phase_o = detect \wedge phase_t = detect \\ sep &\equiv distance(o, t) > \text{min. sep.} \\ alarm &\equiv alarm_t = \text{true} \\ track &\equiv status_o \notin \{taxi, flythru\} \vee status_t \notin \{taxi, flythru\} \end{aligned}$$

We begin with asking the most simple safety property.

A safety property. “Is there a tracked state where minimum separation is lost and the alarm is off?”

$$- \text{CTL syntax: } EF(detect \wedge track \wedge \neg sep \wedge \neg alarm)$$

The omission of the predicate *detect* from the query can lead to false positives since, if a target’s coordinates are inspected in the middle of the radar updates or its status is queried before it is modified accordingly, the data can be inconsistent. Therefore, all queries should be asked only at the right moment: when ownship executes phase C of its algorithm.

A scenario that satisfies the query arises when the condition “distance is closing” is not satisfied in the current state. This is the case in the third snapshot of Figure 8. However, this might not correspond to an unwanted

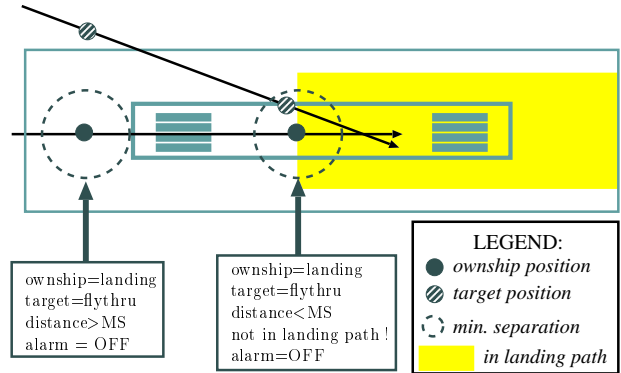


Fig. 9. Scenario 2, airborne: flythru target in conflict with landing aircraft.

behavior, since the alarm might have been set in a previous state, when the minimum separation was lost. The value of the alarm variable also depends on whether the alarm is “aged” or not for a few more cycles. Nevertheless, the situation is still of potential concern, even with aging of the alarm, since the target can maintain a constant distance (at less than minimum separation) for longer than the duration of the aging, eventually resulting in a “bad state” in the round after the alarm expires.

The “memory-less” nature of the query influences the result. We looked at the property in a particular snapshot of time, without considering the sequence of events leading to the current state. To get a better understanding of the system, we next investigate the states of the system immediately after the minimum separation distance between two aircraft is lost.

Analysis of the transition that causes loss of separation. “Is there a state where minimum separation is lost by transitioning to the current state while the alarm is off?”

$$- \text{CTL: } EF(detect \wedge track \wedge sep \wedge E[(\neg detect) U (detect \wedge track \wedge \neg sep \wedge \neg alarm)])$$

The nested EU operator in the query (instead of EX) is due to the fact that several transitions are needed to complete the update of the coordinates, 3, and of the status, 1, and to set the alarm again, 1. A witness for this query (see Figure 9) has ownship in a landing or climbout state, the target flying across the runway faster than ownship, moving within separation distance from the side at an angle. The condition for setting an alarm in this circumstance is “distance less than minimum separation **and** target in takeoff/landing path”. The second term is not satisfied, hence no alarm is raised. Aircraft can actually collide (trajectories intersect in Figure 9), while none of the participants are ever warned.

The above scenario is the only one satisfying this query, a fact attesting to the robustness of RSM. This situation can be corrected by adding “distance less than

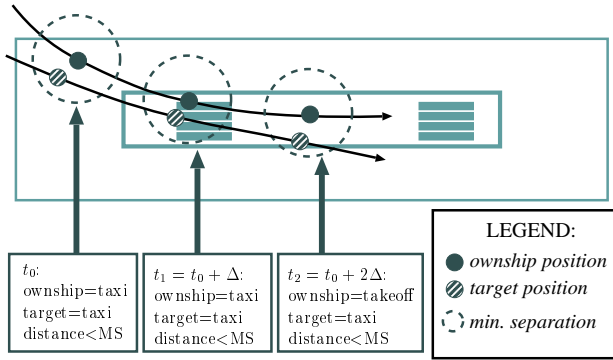


Fig. 10. Scenario 3 (ground level): taxiing target interferes with ownship taking off.

minimum separation” as part of the criterion for this combination of states.

We included the predicate *track* in both states (before and after the transition), as we are interested in scenarios involving only takeoff and/or landing trajectories. However, this additional constraint could mask some other undesired behaviors. Therefore, we next ask a more general question.

A stronger safety property. “Is there a tracked state where minimum separation is lost, reachable without ever previously setting the alarm?”

$$- \text{CTL: } E[(\neg \text{alarm}) \cup (\text{detect} \wedge \text{track} \wedge \neg \text{sep} \wedge \neg \text{alarm})];$$

Several scenarios satisfy this query.

Example 1. As shown in Figure 10, actors enter the monitored area taxiing fast, not aligned to the runway, and already at close distance to each other. Note that the RIPS specifications explicitly ignore this situation, as the algorithm is only active when ownship is taking off or landing. However, once on the runway, say ownship changes direction and aligns itself to the runway. Thereafter, it is categorized as takeoff (or climbout, if it becomes airborne). The other aircraft stays within minimum separation, but it does not close in: it can be either behind ownship or, more dangerously, in front of it. No alarm is raised because the criterion “distance is closing” is, again, not satisfied. If the distance between aircraft at entry is very small, there might not be enough time for an escape maneuver, even if, later on, the alarm is set by closing in.

Figure 10 shows an abstract trace that contradicts this safety property. The trajectories are shown for a horizontal section in the monitored zone at ground level. The third snapshot illustrates the “bad state” of the system: the two aircraft are within minimum separation distance, but no alarm has been issued either for the current state or any of the previous states in the scenario.

Example 2. An identical scenario exists for airborne states that are not tracked (status *flythru*).

Example 3. Additional scenarios do not satisfy this safety property, where events develop immediately after

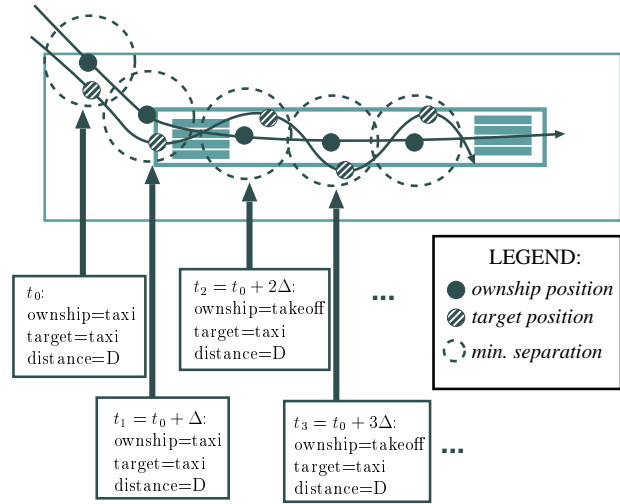


Fig. 11. Scenario 4 (ground level): target zigzags near the ownship taking off.

both planes enter the monitored zone. The bad behavior in these cases is caused by the fact that the previous position is unknown — coordinates (0, 0, 0) in our model — for both planes, hence the distance cannot be closing in the next state. If the airplanes enter the zone at positions very close to each other (e.g., both are trying to land), the alarm will not be raised. However, this behavior is exhibited only in our theoretical model, due to our choice of modeling conventions, and not in the actual implementation of the protocol on the RSM device.

Summarizing the common characteristics of the above scenarios, we observe that the key factor is that both aircraft are in the taxi or flythru status when minimum separation is lost. The situation is not tracked, hence a potentially bad occurrence is masked by a protocol specification. The predicate “distance closing” is not satisfied and no alarm is issued, although the distance is less than minimum separation.

To further extend our discussion, we look at possible continuations of the scenario after the bad state is reached. If the distance is closing in the next state, a warning will be issued and the “missed alarm” situation will cease to exist. The only way for a malicious agent to perpetuate the problem is shown in Figure 11, which is an extension of Scenario 3. The target can stay within minimum separation radius for a longer period of time if it “zig-zags” in front of ownship and at each radar update has the same distance to ownship. The target *has to* zig-zag to maintain the distance, since following a parallel path to ownship will cause RSM to consider it as taking off. The alarm criterion for the new combination of operational states is “taking off in the same direction and distance less than minimum separation”. Therefore, an alarm will be issued as soon as the target stops zig-zagging.

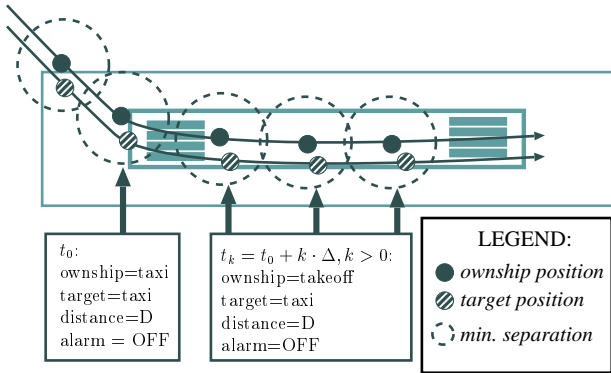


Fig. 12. Scenario 5: aircraft vs. ground vehicle.

The case when the target is not an aircraft (vehicle, service truck, etc.) adds an extra degree of freedom for a malicious behavior of the target (see Scenario 5, in Figure 12). Initially, ground vehicles were always considered in taxi mode by the protocol, regardless of their speed, heading, and physical coordinates. Therefore, as in Scenario 4, the target may follow ownership at close distance, and even continue chasing ownership after it is lined up for takeoff and accelerating. No flag will be raised for the same reasons as in Scenario 4.

For the most recent implementation of RSM, the designers took into account our findings and eliminated the special treatment given to ground vehicles. This addresses the situation in Scenario 5.

The situation in Scenario 4 is of less concern, since it is extremely difficult to realize in practice, even intentionally by a saboteur. At the same time, there is some benefit in exposing it: the designer is aware of this low-probability event. Also, by the fact that it is the only remaining unwanted behavior in the system, it serves as a validation for phase 3 of the RSM algorithm.

6 Conclusions and future work

Human effort vs machine effort. The modeling and analysis of RSM was conducted in a relatively short period of time: twelve months. Up to three people worked in various stages of the project, for an estimated total of nine man months of work. Of this, approximately six months were dedicated to modeling decisions, during which three regular meetings with the design engineers were scheduled. The final three man months were spent with the verification per se: formulating queries, analyzing, and synthesizing the answers. The understanding of RSM was facilitated by the the designers' strive to keep the specification and implementation as simple as possible.

Overall, while the human effort was predominant in all stages of the analysis, mostly due to the lack of automation in the modeling process, the impact of efficient model checking tools was crucial in completing the

study. At the same time, for a short-time collaboration like this, it was only natural that the experience of the modelers was often preferred to certain automated processes, like automatic model extraction and automated abstraction/refinement schemes.

Lessons learned. Several lessons were learned from our analysis, first and foremost that formal verification has an undeniable value. We presented the designers with a list of important findings which were not exposed during the testing activities involving real aircraft, already underway at different airport locations. The merit of our technique is that, besides being considerably less expensive than testing, it is *exhaustive*.

We were able to analyze all possible scenarios in our model and found situations of potential concern that happen with extremely low probability. These are almost impossible to expose during either testing procedures, which usually afford no more than a dozen test flights a day, or simulation sessions. When compared to the actual state space sizes of the order of $10^{13} - 10^{42}$ states, this shows the need for exhaustive analysis.

The second outcome of our experiments was that, after identifying the problems and suggesting modifications to the protocol to eliminate them, we have increased the level of assurance of the design in what concerns missed alarms. All the findings were related to situations when only one aircraft is landing or taking off and the other is not. The section of the decision table dealing with both aircraft landing or taking off was validated in the original form.

With respect to the dual analysis, of false alarms, this is still on the list of future plans. From a practical point of view, pilots are equally concerned with both types of situations. Individual reports indicate that frequent false alarms can become a distraction or, in the best case, a nuisance factor in operating an airplane. It is also the case that a system with too many false alarms will tend to be switched off or ignored, thus rendering it useless. Therefore the occurrence rate of false alarms has to be reduced, even though these are not as critical as missed alarms, which should be *completely* eliminated. The designers of the protocol had to come up with a balanced solution trading the simplicity of very “loose” requirements that raise too many alarms for the complexity of “stricter” conditions that decrease the number of false alarms, but make the analysis more difficult.

Another aspect not discussed here is fault tolerance. We assumed that all scenarios happen in the absence of communication faults, meaning that the radars and data-links provide accurate and timely updates to all participants. A natural extension of our analysis is to include faulty behaviors, of either benign nature (missed or late updates) or malicious/Byzantine (inconsistent data between participants). This type of analysis requires the inclusion of probabilistic aspects in the model, and will be the subject of further research. While our work verifies the correct operation of RSM under no-fault assump-

tions, the presence of faults on the data-link may significantly impact the correct operation of the algorithm. On the one hand, if all data is faulty, RSM will be of no help whatsoever in avoiding incursions. On the other hand, if no data is faulty, we have already demonstrated the correctness of the algorithm. The task of realistically modeling faulty data that falls in between these extremes is a major challenge.

Finally, this case study has inspired ideas of theoretical nature that can result in improvements and extensions to our technique. The observation that a single temporal logic query can have more than one counterexample (thus correcting it alone will not entirely rid the system of the error) suggests that generating and storing all counterexamples is beneficial. This is not commonly done by model checkers. Also, even though the Kronecker matrix encoding of the transition relation has been found to be more efficient than the traditional BDD encoding in all our previous studies, this particular application has revealed a situation when the need to satisfy the consistency requirements could be detrimental, by producing an excessive number of split events. A saturation approach that does not require the model to be Kronecker consistent has been proposed in [25], but it can suffer from poor performance due to excessively large local state spaces; we are currently working on extending it so that it is completely general, yet with an efficiency approaching that of the Kronecker-consistent case.

References

1. Sharon O. Beskenis, David F. Green, Paul V. Hyer, and Edward J. Johnson, Jr. Integrated Display System for Low Visibility Landing and Surface Operations. NASA Langley Contractor Report 208446, July 1998.
2. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comp.*, 35(8):677–691, Aug. 1986.
3. Victor Carreño, Hanne Gottlieb, Rick Butler, and Saraswati Kalvala. Formal Modeling and Analysis of a Preliminary SATS Concept. NASA Langley Technical Report 12999, March 2004.
4. William Chan, Richard J. Anderson, Paul Beame, Steve Burns, Francesmary Modugno, David Notkin, and Jon D. Reese. Model checking large software specifications. *IEEE Transactions on Software Engineering*, 24(7):498–520, July 1998.
5. Daniel M. Chapiro. Globally-Asynchronous Locally-Synchronous Systems. Ph.D. Thesis, Stanford University, 1984.
6. Gianfranco Ciardo, Robert L. Jones, Andrew S. Miner, and Radu I. Siminiceanu. SMART: Stochastic Model checking Analyzer for Reliability and Timing, User Manual. <http://cs.ucr.edu/~ciardo/SMART/>.
7. Gianfranco Ciardo, Robert L. Jones, Andrew S. Miner, and Radu I. Siminiceanu. Logical and stochastic modeling with SMART. In *Modeling Techniques and Tools for Computer Performance Evaluation*, LNCS 2794, pp. 78–97, September 2003.
8. Gianfranco Ciardo, Gerald Lüttgen, and Radu Siminiceanu. Saturation: An efficient iteration strategy for symbolic state space generation. In *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2001)*, LNCS 2031, pp. 328–342, Genova, Italy, April 2001.
9. Gianfranco Ciardo and Andrew S. Miner. A data structure for the efficient Kronecker solution of GSPNs. In Peter Buchholz, editor, *Proc. 8th Int. Workshop on Petri Nets and Performance Models (PNPM'99)*, pp. 22–31, Zaragoza, Spain, September 1999. IEEE Computer Society Press.
10. Gianfranco Ciardo and Radu Siminiceanu. Using edge-valued decision diagrams for symbolic generation of shortest paths. *Proc. Fourth International Conference on Formal Methods in Computer-Aided Design (FMCAD '02)*, LNCS 2517, pp. 256–273, Portland, November 2002. Springer-Verlag.
11. Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. NuSMV: a new Symbolic Model Verifier. *Proc. Computer-Aided Verification (CAV'99)*, N. Halbwachs and D. Peled, editors, LNCS 1633, pp. 495–499, Trento, Italy, July 1999. Springer-Verlag.
12. Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. *IBM Workshop on Logics of Programs*, LNCS 131, pp. 52–71. Springer-Verlag, 1981.
13. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 1999.
14. Gilles Dowek, Cesar Muñoz, and Victor Carreño. Abstract Model of the SATS Concept of Operations: Initial Results and Recommendations. NASA Langley Technical Report 213006, March 2004.
15. J.-C. Fernandez, H. Garavel, A. Kerbrat, and L. Mounier. CADP: a protocol validation and verification toolbox. LNCS 1102, pp. 437–446, Springer-Verlag, 1996.
16. David F. Green, Jr. Runway safety monitor algorithm for runway incursion detection and alerting. NASA Langley Contractor Report 211416, Jan 2002.
17. Mats P.E. Heimdahl. Experiences and lessons from the analysis of TCAS II. *Proc. ACM SIGSOFT Int. Symposium on Software Testing and Analysis (ISSTA '96)*, pp. 79–83, 1996.
18. Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. HYTECH: a model checker for hybrid systems. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1–2):110–122, December 1997.
19. Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.
20. Denise R. Jones. Runway Incursion Prevention System Fact Sheet. October 2000.
21. Marta Kwiatkowska, Gethin Norman, and David Parke. PRISM: Probabilistic Symbolic Model Checker. *Proc. Computer Performance Evaluation / TOOLS*, pp. 200–204, Apr 2003.
22. Carolos Livadas, John Lygeros and Nancy A. Lynch. High-Level Modeling and Analysis of TCAS. *RTSS '99*, IEEE Press, 1999.
23. Kenneth L. McMillan. *Symbolic Model Checking: An Approach to the State-explosion Problem*. PhD thesis, Carnegie-Mellon Univ., 1992.

24. Andrew S. Miner and Gianfranco Ciardo. Efficient reachability set generation and storage using decision diagrams. Proc. 21th Int. Conf. on Applications and Theory of Petri Nets (*ICATPN '99*), LNCS 1639, pp. 6–25, June 1999.
25. Andrew S. Miner. Saturation for a general class of models. In G. Franceschinis, J.-P. Katoen, and M. Woodside, editors, *Proc. QEST*, pages 282–291, Enschede, The Netherlands, Sept. 2004.
26. Tadao Murata. Petri Nets: properties, analysis and applications. Proc. IEEE 77(4):541–579, April 1989.
27. J. Timmerman. Runway Incursion Prevention System, ADS-B and DGPS data link analysis, Dallas – Ft. Worth International Airport. NASA Contractor Report 211242, NASA Langley, Hampton, VA, November 2001.