CHAPTER 5

SYMBOLIC MODEL CHECKING FOR AVIONICS

RADU I. SIMINICEANU
National Institute of Aerospace, Hampton, VA

GIANFRANCO CIARDO
Department of Computer Science and Engineering, University of California, Riverside, CA

5.1 INTRODUCTION

Since the 1970s, both on-board flight management systems and air traffic management operations on the ground have become significantly more automated. The cockpits of commercial and military airplanes are now heavily computerized. The general term for this class of equipment is avionics. This includes digital electronic devices, components, and entire subsystems used for

- navigation, such as the global positioning system (GPS);
- control, for example, autopilot systems;
- communications, for example, Automatic Dependent Surveillance-Broadcast (ADS-B), which in turn is based on GPS;
- conflict detection and resolution (CD&R) protocols, for example, the Traffic Collision Avoidance System (TCAS) and the Runway Incursion Prevention System (RIPS);
- flight data recorders;
- integrated display systems;
- entire aircraft management systems, for example, the Integrated Modular Avionics (IMA); and
- unmanned vehicle systems (UASs).
For safety-critical avionics, regulatory authorities (the Federal Aviation Administration [FAA], the Civil Aeronautics Authority [CAA] or the Department of Defense [DoD]) require software development standards, such as MIL-STD-2167 for military systems and RTCA DO-178B for civil aircraft. This is both a challenge and an opportunity for the formal methods community, whose technology has not been initially included in the regulations, but has since made tremendous progress and is now strongly recommended in the process of building dependable systems [39].

The challenges in verifying avionics systems can be attributed to two main factors:

- The *hybrid* nature of the embedded systems: Both discrete and continuous variables are needed to faithfully model concepts such as (continuous) aircraft trajectories, the physics behind the operating environment (e.g., temperature, pressure, speed, and acceleration), and most notably “real time.”
- The degree of *complexity* of the systems: The number of subcomponents, their size, and the interactions between these components can be quite large.

Hybrid systems analysis has been around for almost two decades. But, as pointed out in Reference 43, the research community has from the beginning adopted a methodology based on accepting the foundations of the theory of computation as “transformation of data and not about physical dynamics.” Thus, real time has been modeled by retrofitting basic abstract models, such as automata, with temporal properties. This is the way timed automata [2] and other incarnations (timed I/O automata, timeout automata, calendar automata) were conceived. Despite massive research efforts, especially in establishing the formal mathematical foundations, progress in terms of practical applications has virtually stalled and has not moved beyond systems with more than a dozen continuous variables. For avionics systems, a basic 3-D collision avoidance protocol, for example, requires at least six real variables to represent the coordinates of an aircraft and an intruder, and six more to represent the 3-D speed vectors, thus is already beyond the means of today’s capabilities.

Satisfiability Modulo Theories (SMT) solvers for linear arithmetic and uninterpreted functions have been used to efficiently verify real-time designs [33]. However, the approach is still limited to models with a relatively small number of discrete states and timing constraints that can be expressed as linear inequalities.

Until the field of hybrid system verification reaches a higher level of maturity, a verification practitioner must resort to more traditional, yet relatively scalable, techniques, such as discrete-state *model checking*. Therefore, the first and foremost task is finding appropriate *abstraction techniques* that allow the analysis of infinite-state spaces in the form of behaviorally equivalent finite models.
Once a suitable abstraction is found, the second challenge is addressing the issue of *state-space explosion*. One possibility is to develop reduction techniques (such as partial order reduction, symmetry reduction, or compositional approaches) that reduce the state space to manageable sizes. Another is to improve the model checking algorithms themselves, as is the case for the saturation algorithm [18] presented in Section 5.2.6. This approach to model checking, initially proposed as a very efficient iteration strategy for generating the state spaces of globally asynchronous, locally synchronous systems, has been gradually improved so that it is now able to tackle a much larger class of problems. In the most recent version, it supports on-the-fly variable domain computation (when the lower and upper bounds for variable values are not known a priori), and it is now also able to analyze models that do not satisfy "Kronecker consistency," as previously required.

In the following, we present our experience in applying the methodology outlined above (i.e., discrete-state model checking of abstracted models) to an industrial size application: the NASA Runway Safety Monitor (RSM) software [35]. This classical approach seems to have maintained a good track record in the industry, as also illustrated by recent reports from ONERA and Airbus [5, 51].

We do not pretend or intend to do a comparison of this approach with other formal verification tools and techniques, since this is not the custom practice in the industry, where the toolsets and skillsets of the personnel determine quite strictly the verification technology used in each project. We do try, however, to carefully justify our modeling and analysis decisions and share our perspectives and lessons learned from what turned out to be a rather successful project.

### 5.2 APPLICATION: THE RUNWAY SAFETY MONITOR

The RSM is part of the RIPS [40]. Designed and implemented by Lockheed Martin engineers, RSM is intended to be incorporated in the Integrated Display System (IDS) [4], a suite of cockpit applications that NASA has been developing since 1993. IDS also includes other conflict detection and prevention algorithms, such as TCAS II [44]. The IDS design enables RSM to exploit existing data communications facilities, displays, GPS, ground surveillance system information, and data links.

Collision avoidance protocols are already in operation. TCAS [44] has been in use since 1994 and is now required by the FAA on all commercial U.S. aircraft. TCAS has a full formal specification, but it has been verified only partially, due to its complexity [11, 37]. Other protocols, such as Small Aircraft Transportation System (SATS) [1], developed at NASA Langley to help ensure safe landings of general aviation craft at towerless regional airports, and Airborne Information for Lateral Spacing (AILS) [10], an alerting algorithm for parallel landing scenarios, have been instead formally verified [32].
5.2.1 Purpose of RSM

The goal of the RSM 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 RSM plays a key role in accident avoidance. RSM is not intended to prevent incursions, 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 [35, 53] show that incursion situations are less likely to occur when IDS technology is employed on aircraft. RSM should greatly improve this positive effect.

5.2.2 RSM Design

Figure 5.1 shows the high-level architecture of the RSM algorithm. RSM runs on a cockpit-installed device 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 can occur. This is a 3-D volume running up to 220 ft laterally from each edge of the runway, up to 400 ft of altitude above the runway, and 1.1 nautical miles from each runway end (this corresponds to a 3° glide slope for takeoff and landing trajectories).

![FIGURE 5.1 RSM algorithm top-level design.](image-url)
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 studies. These errors have already been the subject of some experimental measurements [53], and their analysis calls for a stochastic flavor not captured in our model, which is instead concerned only with logical errors.

In the second phase, the algorithm determines the status of each target, from a predetermined set of values: taxi, pre-takeoff, takeoff, climb out, landing, roll out, and fly-through modes. We discuss in detail the meaning of these states in Section 5.3.

The third phase is responsible for detecting incursions and is performed for each target based on the spatial attributes (position, heading, and acceleration) of ownship and target, plus some logical conditions. Table 5.1 in Section 5.3.2, shows the operational state matrix of this phase. The safety analysis focused on verifying that the decision criteria listed in the table are able to detect all possible incursion scenarios.

5.2.3 Formal Verification of RSM

We adopted a model-based approach to verification, rather than attempting to verify the actual C code itself, for two reasons. First, expressing the main safety property (“no missed alarms”) requires representing the environment (e.g., the movement of targets, which is only tracked by the software, while the laws governing the dynamics of the targets are not represented in the code). Second, temporal logic model checking is not known to be able to handle code of this size (in contrast to other static analysis techniques that have been applied to avionics software [28]).

We argue that a discrete-state model is adequate for this application (and, for similar air traffic monitoring applications, be they en route or airportal) because, in any behavior observable by the software, the state of the system is already discretized, as aircraft positions are taken at regular time intervals and not continuously.

Before proceeding with the description of our extracted model of RSM, we review the main features of the symbolic verification technique we have developed in the course of the last years.

5.2.4 Structural Symbolic Model Checking

When applying symbolic model checking to large and complex systems, the memory and time requirements can quickly become a formidable obstacle. To
mitigate this problem, one can try to exploit the special characteristics of the system under study. Our technique, which we call structural symbolic model checking [14, 23], is particularly successful in this regard when applied to globally asynchronous, locally synchronous systems [12]. Its main idea is to recognize locality (i.e., the fact that most events affect only a few of the many state variables) and use it to perform a saturation-style fixed-point iteration instead of a more traditional breadth-first-style iteration (i.e., events are explored in a particular locality-guided order that can greatly reduce the peak size of the decision diagrams).

We represent discrete-state models as a triple \((S_{pot}, S_{init}, \mathcal{N})\), where \(S_{pot}\) is the set of potential states of the model, \(S_{init} \subseteq S_{pot}\) is the set of initial states, and \(\mathcal{N} : S_{pot} \rightarrow 2^{S_{pot}}\) is the next-state function specifying the states reachable from each state in a single step. In practice, the model is described compactly in some high-level formalism (e.g., Petri nets or process algebras), but the size of \(S_{pot}\) is usually very large. A fundamental question is then to determine the set of states that are actually reachable from the initial states through repeated applications of the next-state function. Formally, the (reachable) state space \(S_{rch} \subseteq S_{pot}\) is the smallest superset of \(S_{init}\) closed with respect to \(\mathcal{N}\):

\[
S_{rch} = S_{init} \cup \mathcal{N}(S_{init}) \cup \mathcal{N}(\mathcal{N}(S_{init})) \cup \cdots = \mathcal{N}^*(S_{init}),
\]

where \(\mathcal{N}^*\) denotes reflexive and transitive closure and \(\mathcal{N}(X) = \bigcup_{i \in X} \mathcal{N}(i)\).

The first step in structuring the model is to decompose the model into \(K\) submodels, or, in other words, write a (global) state \(i\) as a \(K\)-tuple \((i_K, \ldots, i_1)\), where \(i_k\) is the local state of submodel \(k\), for \(K \geq k \geq 1\). Thus, the potential state space is given by the cross-product of \(K\) local state spaces, \(S_{pot} = S_K \times \cdots \times S_1\).

Assuming for now that each \(S_k\) is known a priori, we can map each of its local states \(i_k\) to an index \(i_k\) in the range \([0, 1, \ldots, n_k - 1]\), where \(n_k = |S_k|\). Then, we can identify \(S_k\) with \([0, 1, \ldots, n_k - 1]\) and encode any set \(X \subseteq S_{pot}\) in a quasi-reduced ordered multiway decision diagram [41, 42, 48] (MDD) over \(S_{pot}\).

Formally, an MDD is a directed acyclic edge-labeled multigraph where

- each node \(p\) belongs to a level in \([K, \ldots, 1, 0]\). If the level is \(k > 0\), we also say that the node refers to variable \(x_k\);
- level 0 can only contain the two terminal nodes Zero and One;
- there is a unique root node \(r\) with no incoming arcs, and it is either at level \(K\) or it is the terminal node Zero;
- a node \(p\) at level \(k > 0\) has \(n_k\) outgoing edges, labeled from 0 to \(n_k - 1\). The edge labeled by \(i_k\) points to a node \(q\), which is either the terminal Zero or a node at level \(k - 1\); we write \(p[i_k] = q\);
- there are no duplicate nodes: If \(p\) and \(q\) are at level \(k\) and \(p[i_k] = q[i_k]\) for all \(i_k \in S_k\), then \(p = q\).
Note that MDDs are a generalization of the very successful BDDs [6], which are restricted to binary choices at each node, thus require multiple Boolean variables to encode a single index $i_k$ whenever $n_k > 1$.

The MDD encodes a set of states $B(r)$, defined by the recursive formula

$$B(p) = \begin{cases} 0 & \text{if } p = \text{Zero} \\ \{i_1 : p[i_1] = \text{One} \} & \text{if } p \text{ is at level 1} \\ \bigcup_{i_k \in S_k} \{i_k \times B(p[i_k]) \} & \text{if } p \text{ is at level } k > 1 \end{cases}$$

In symbolic algorithms, several sets of states $\mathcal{X}^{(1)}, \ldots, \mathcal{X}^{(m)}$, all subsets of $S_{\text{pot}}$, may need to be stored at the same time. This is achieved through an MDD forest that stores the roots $r^{(1)}, \ldots, r^{(m)}$ of these MDDs and forces them to share nodes, so that duplicates are avoided even across different MDDs. A fundamental property of the MDDs just defined is that they are canonical: For a given variable order $x_K, \ldots, x_1$, two nodes representing the same set are isomorphic and, when using an MDD forest, this situation is easily recognized in $O(1)$ time using hash table techniques.

In addition to the compact representation of sets of states provided by MDDs, symbolic methods also require a compact representation of the next-state function. Since $\mathcal{N}$ can be seen as the set of pairs $(i, j)$ such that $j \in \mathcal{N}(i)$, we can store this set using a $2K$-level MDD over the “from” variables $x_K, \ldots, x_1$ and the “to” variables $y_K, \ldots, y_1$. However, this monolithic MDD encoding often requires excessive memory. The second structuring step is then to decompose $\mathcal{N}$ into a disjunction of next-state functions [9]: $\mathcal{N}(i) = \bigcup_{\alpha \in \mathcal{E}} \mathcal{N}_\alpha(i)$, where $\mathcal{E}$ is a finite set of events and $\mathcal{N}_\alpha$ is the next-state function associated with event $\alpha$. Then, we say that $\mathcal{N}_\alpha(i)$ is the set of states the system can enter when $\alpha$ occurs, or fires, in state $i$, and that $\alpha$ is disabled in $i$ if $\mathcal{N}_\alpha(i) = \emptyset$, and enabled otherwise. Usually, storing $|\mathcal{E}|$ MDDs in an MDD forest with the interleaved variable ordering $(x_K, y_K, \ldots, x_1, y_1)$ greatly improves the memory requirements for storing $\mathcal{N}$ [20].

The structural approach goes a step further and partitions each $\mathcal{N}_\alpha$ according to the $K$ state variables. An efficient approach [17, 48] adopts a Kronecker representation [31] inspired by the work on Markov chains [7], assuming that the model is Kronecker consistent, that is, that $\mathcal{N}_\alpha$ can be conjunctively decomposed into $K$ local next-state functions $\mathcal{N}_{k,\alpha}: S_k \rightarrow 2^{S_k}$, for $K \geq k \geq 1$, satisfying

$$\forall (i_K, \ldots, i_1) \in S_{\text{pot}}, \mathcal{N}_\alpha(i_K, \ldots, i_1) = \mathcal{N}_{k,\alpha}(i_k) \times \cdots \times \mathcal{N}_{1,\alpha}(i_1).$$

Defining $K \times |\mathcal{E}|$ matrices $N_{k,\alpha} \in \{0, 1\}^{n_k \times n_k}$, with $N_{k,\alpha}[i_k, j_k] = 1 \iff j_k \in \mathcal{N}_{k,\alpha}(i_k)$, we encode $\mathcal{N}_\alpha$ as a Kronecker product: $j \in \mathcal{N}_{\alpha}(i) \iff \otimes_{K \geq k \geq 1} N_{k,\alpha}[i_K, j_k] = 1$, where a state $i$ is interpreted as a mixed-base index in $S_{\text{pot}}$ and $\otimes$ indicates the (Boolean) Kronecker product of matrices.
The Kronecker encoding of $\mathcal{N}$ is quite memory efficient also because it can be used to recognize and exploit event locality. We say that event $\alpha$ is independent of level $k$ if $\mathbf{N}_{k,\alpha} = \mathbf{I}$, the identity matrix. For globally asynchronous, locally synchronous models, most events affect only a few state variables and are independent of all others. Only the matrices $\mathbf{N}_{k,\alpha} \neq \mathbf{I}$ require actual storage, and their number in practice is often $O(|\mathcal{E}|)$ instead of $O(K|\mathcal{E}|)$. Furthermore, these matrices are extremely sparse in practice and often require just $O(n_k)$ memory instead of $O(n_k^2)$.

5.2.5 The Saturation Algorithm for Symbolic State-Space Generation

Recognizing event locality is essential not only for a Kronecker representation of $\mathcal{N}$, but also for the saturation algorithm [18]. Let $\text{Top}(\alpha)$ and $\text{Bot}(\alpha)$ denote the highest and lowest levels for which $\mathbf{N}_{k,\alpha} \neq \mathbf{I}$. We say that a node $p$ at level $k$ is saturated if it is a fixed point with respect to all events $\alpha$ such that $\text{Top}(\alpha) \leq k$; that is,

$$\forall i_1, \ldots, i_{k+1} \in S_k \times \cdots \times S_{k+1}, \text{Top}(\alpha) \leq k \Rightarrow,$$

$$\{(i_1, \ldots, i_{k+1})\} \times \mathcal{B}(p) \supseteq \mathcal{N}_a (\{(i_1, \ldots, i_{k+1})\} \times \mathcal{B}(p)).$$

This is succinctly written as $\mathcal{B}(p) \supseteq \mathcal{N}_{a,k} (\mathcal{B}(p))$ by defining $\mathcal{N}_{a,k} = \cup_{\alpha: \text{Top}(\alpha) \leq k} \mathcal{N}_a$ and allowing the application of $\mathcal{N}_a$ to a set $\mathcal{X} \subseteq S_l \times \cdots \times S_1$ with $l \geq \text{Top}(\alpha)$, since $\mathcal{N}_a$ leaves all local states above $\text{Top}(\alpha)$ unchanged.

The saturation algorithm starts with the MDD encoding the initial states $S_{\text{init}}$ and saturates its nodes bottom-up. A simplified high-level pseudocode for the algorithm is shown in Figure 5.2, where $\mathcal{E}_k = \{ \alpha: \text{Top}(\alpha) = k \}$.

Saturation employs multiple nested lightweight, fixed-point iterations, unlike the traditional breadth-first approach, which applies every $\mathcal{N}_a$ to the entire set of known states, or to the entire set of newly discovered states, at every iteration. Whenever saturate operates on a node $p$ at level $k$, its descendants have been already saturated. To saturate $p$, the algorithm repeatedly fires on $p$ all events $\alpha$ with $\text{Top}(\alpha) = k$ until reaching a fixed point, with the proviso that, if a firing creates new nodes at levels below $k$, they are saturated immediately, prior to completing the saturation of $p$.

Results in Reference 18 show that saturation greatly outperforms breadth-first symbolic exploration by several orders of magnitude in both memory and time, making it arguably the most efficient symbolic fixed-point iteration strategy for state-space generation of globally asynchronous, locally synchronous discrete event systems.

While the algorithm we presented assumes that the local state space $S_k$ is known prior to calling StateSpaceGeneration, this is not a requirement. The saturation algorithm has been extended so that new local states can be discovered on-the-fly using explicit local state-space explorations interleaved with the global symbolic state-space generation just presented [19]. While the algorithmic implementation becomes substantially more difficult in this case, the
theoretical complexity remains the same, and the ease of modeling is substantially enhanced in practice, as the modeler need not worry about the bounds of the local state variables.

5.2.6 Saturation-Based Model Checking

The advantages of using a Kronecker representation for the next-state function apply also to a breadth-first iteration for state-space generation, and to the more general iterations required for symbolic Computation Tree Logic (CTL) model checking [25, 46], even though the latter is much more challenging.
It is well known that a CTL formula can be expressed using the three operators $\text{EX}$, $\text{EU}$, and $\text{EG}$, plus the Boolean operators $\neg$, $\land$, and $\lor$. Clearly, saturation is not applicable to the computation of the $\text{EX}$ operator, since this involves the application of exactly one step of the next-state function $N$, while any fixed-point algorithm, including saturation, is concerned with arbitrarily long sequences of steps in the evolution of the model. Analogously, the $\text{EG}$ operator starts with a set of states $P$ satisfying a particular condition $p$, and iteratively removes from $P$ any state $i$ that does not have a transition to a state $j \in P$. In other words, unlike state-space generation, where a state $j$ can be added to the set of states $S_{\text{rch}}$ if there is any event $\alpha$ such that $j \in N_\alpha(i)$ for some state $i \in S_{\text{rch}}$, the condition for removing a state in the $\text{EG}$ iterations requires to examine all events. Consequently, saturation does not appear to be directly applicable to $\text{EG}$ either.

For the $\text{EU}$ operator, however, the situation is much better [23]. First, consider the $\text{EF}$ operator, which can be seen as a special case of $\text{EU}$, since $\text{EF}q$ is equivalent to $\text{E}[\text{true} \cup q]$. To symbolically compute the set of states satisfying $\text{EF}q$, we can start from the set $Q$ of states satisfying condition $q$, and “walk the model backwards,” adding to $Q$ all the states that can transition to a state in $Q$. This is exactly the state-space generation problem, where the initial set of states is $Q$ and the next-state function is the inverse of the original next-state function of the model, that is, $i \in N^{-1}(j)$ iff $j \in N(i)$; thus, saturation naturally applies, with all its benefits. It is worth observing that, with our Kronecker encoding of $N$, it is sufficient to use the transpose matrices $N_k^T$ to describe the backward evolution of the model.

When tackling the general problem of computing the state satisfying condition $\text{E}[p \cup q]$, however, the original saturation algorithm cannot be directly applied, since we must ensure that, as we walk backward from the states in $Q$, we do not stray from the states in $P$ (satisfying $p$). The solution proposed in Reference 23 partitions the events $\mathcal{E}$ into safe events $\mathcal{E}_S$ and unsafe events $\mathcal{E}_U$. The distinguishing characteristic of the former set is that, for any safe event $\alpha$ and any state $j \in P \cup Q$, $i \in N_{\alpha}^{-1}(j)$ implies that $i \in P \cup Q$ as well; that is, it is not possible to move from a state not satisfying $p$ or $q$ to a state satisfying $p$ or $q$ by firing $\alpha$. The cost to classify all events is essentially just that of one breadth-first iteration (each event must be fired backward once). Then, we can compute the set of states satisfying $\text{E}[p \cup q]$ by alternating a backward saturation step on the safe events (i.e., $\text{EF}$ with the set of events $\mathcal{E}_S$) with a backward breadth-first step (i.e., $\text{EX}$ with the set of events $\mathcal{E}_U$), until no more states can be added. The efficiency of this saturation-based $\text{EU}$ algorithm depends on the model, and, in particular, on how many events can be classified as safe. In the best case, all events are safe (e.g., in the $\text{EF}$ case), while, in the worst case, no event is safe (the complexity is then that of the traditional symbolic breadth-first algorithm, although still with the advantages of a Kronecker representation for $N$).

An alternative approach to employ saturation for the $\text{EU}$ computation was recently proposed [55], where each next-state function $N_\alpha$ is modified by being...
constrained to stay in \( P \) when going backward, through an intersection with \( P \times S_{pot} \), performed either before beginning the computation, or on-the-fly during the fixed-point iteration. This new approach is even more efficient than the one in Reference 23, but it had not been discovered at the time we conducted the study described in this chapter.

5.2.7 The Stochastic and Model-Checking Analyzer for Reliability and Timing (SMAR\textsc{T}) Tool

The symbolic techniques we discussed are implemented in the tool SMAR\textsc{T} [16]. Given a description of a system as an extended Petri net [49], SMAR\textsc{T} can generate the state space, verify temporal logic properties, and compute numerical solutions for timing and stochastic analysis. SMAR\textsc{T} implements a wide range of explicit as well as symbolic methods. In addition to using MDDs to encode sets of states and either Kronecker operators, matrix diagrams [21, 47], or MDDs to encode the next-state function, SMAR\textsc{T} also employs a special form of edge-valued MDDs (EV$^+$MDDs) [22] for the generation of minimal-length counterexamples and witnesses in CTL model checking.

The SMAR\textsc{T} input is an extended Petri net with Turing-equivalent extensions (immediate transitions, marking dependent arc cardinalities, and transition guards) [49], which must have a finite-state space. Each SMAR\textsc{T} input file defines one or more structured (i.e., partitioned into submodels) 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.

The SMAR\textsc{T} User Manual is available online [15].

5.3 A DISCRETE MODEL OF RSM

5.3.1 Abstraction on Integer and Real Variables

Automated abstraction techniques are desirable and have been successfully used in many applications. The best known techniques include abstract interpretation [29], domain abstraction [30], data abstraction [26, 27, 38], predicate abstraction [34, 45], automated abstraction refinement (CEGAR) [8, 24], approximation [3, 50, 52], data equivalence [13], slicing [36], and verification of regular infinite-state spaces [54].

Manual abstraction has been seen as more problematic, as it usually requires a good understanding of the system and how this evolves. In many cases, however, a manually built model can be tailored to the strengths of the tool that is used. Therefore, manual abstraction can often yield better results, because the modeler has more control over the process. While automation is highly desirable, the very generality of the automated approaches can lead to their demise. For example, generated abstractions can be inefficient, inaccurate, or suffer from the Zeno effect in the case of CEGAR.
Moreover, in our particular case, we exploit the fact that saturation targets the analysis of globally asynchronous, locally synchronous systems. By modeling the continuous environment and the discrete system events such that there is a high degree of event locality in the model, saturation-based model checking succeeds in building the state space of the abstract model. No further reduction techniques (partial order reduction, symmetry, predicate abstraction) are needed, but some of them could be completely orthogonal to the discretization method, and thus could be applied on top of the reduced model. Also, the discretization not being so drastic (as it could be in the case of applying predicate abstraction), the system behaviors still closely resemble actual aircraft trajectories.

5.3.2 The SmART Model of RSM

Unlike using a predicate abstraction technique that would ultimately divide the entire 3-D space into (possibly irregular, but logically equivalent) regions, we decided to establish a fixed number of variables (the coordinates on the three axes), then divide their domain into segments. In taking this decision, we took into account two facts. First, there was no actual formal specification of the desired set of properties of the protocol that would drive the predicate abstraction. Second, our type of discretization leads to modeled executions that are closer to the geometry of airplane motion in space, which is thus better understood by the design engineers, who have no training in, and possibly no clear understanding of, abstraction techniques.

In the end, we chose the system variables by partitioning the model into \( n + 1 \) submodels where \( n \) is the number of targets moving inside the zone. The variables of submodel 0 describe the state of ownship, while those of the other \( n \) submodels describe the state of each target. For submodel \( i \), \( 0 \leq i \leq n \), the relevant attributes are the following:

- **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 \((v_{x_i}, v_{y_i}, v_{z_i})\).
- **Acceleration along the Runway.** \( a_{y_i} \).
- **Status.** An enumerated-type variable, \( status_i \).
- **Alarm Flag.** A Boolean variable, \( alarm_i \).
- **Phase.** An enumerated-type variable, \( phase_i \).

The variable domains are the following (the subscript \( i \) is omitted for readability):

- 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 representation: \( x \in \{0, \ldots, \text{max}_x\} \), \( y \in \{0, \ldots, \text{max}_y\} \), and \( z \in \{0, \ldots, \text{max}_z\} \), where 0 means outside the zone, and the constants \( \text{max}_x, \text{max}_y, \) and \( \text{max}_z \) 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, has this location.

- The speed values \( v_x, v_y, v_z \) could be assigned the domain \( \{0, \pm 1, \pm 2\} \), where 0 means not moving, \( \pm 1 \) means moving slowly (below the predetermined taxi speed threshold \( TS \) of 45 knots), and \( \pm 2 \) means moving fast (above \( TS \)). Again, a better representation is \( v_x, v_y, v_z \in \{-\text{max}_{\text{speed}}, \ldots, 0, \ldots, \text{max}_{\text{speed}}\} \) using another parameter \( \text{max}_{\text{speed}} \).
- The acceleration \( a_z \) has only two relevant values: nonnegative or strictly negative.
- The status is one of \( \{\text{out, taxi, takeoff, climb, land, rollout, flythru}\} \).
- The phase is one of \( \{\text{radar_update, set_status, detect}\} \).

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

\[
\text{phase} = \text{radar_update}: \text{Update location of targets.}
\]
\[
\text{phase} = \text{set_status}: \text{Update status of targets.}
\]
\[
\text{phase} = \text{detect}: \text{Set or reset alarm.}
\]

We next discuss the modeling decisions taken for each of these three steps.

### 5.3.2.1 3-D Motion of Targets

Our discretization method divides the monitored space into a 3-D grid. The possible positions of the aircraft are a finite number of grid cells, from the discrete domain \( \{(0, 0, 0)\} \cup \{1, \ldots, \text{max}_x\} \times \{1, \ldots, \text{max}_y\} \times \{1, \ldots, \text{max}_z\} \). Similarly, continuous trajectories have to be represented by discretized trajectories through the cells of the 3-D grid. Three alternatives were considered.

First is the projection method, which assigns to every possible continuous trajectory its corresponding discrete path in the grid. An example of such a projection is given in Figure 5.3 (in 2-D space, for the sake of readability). The grid cells in the figure have the size of 100 units (feet), and the snapshots are taken after each 0.5 time units (seconds). The speed units are measured in axis divisions traversed after each update. The problem with this method is the difficulty in discerning between physical possibilities 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 is 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. 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 sufficiently large. In our simplest model, which captured all the interesting properties, the size of a grid cell is 900 ft. 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 it travels at speeds exceeding $1800 \text{ ft/s} = 1227 \text{ mph}$ (or $1975 \text{ km/h}$). This is over 1.6 times the speed of sound. Although it is not entirely safe to assume that these speeds are not encountered on civil airport runways, their exclusion from our model is reasonable and helps simplify the analysis. Moreover, a rough discretization also mitigates state-space explosion, as the number of possible states becomes manageable. Figure 5.4 shows the possible moves of a target in this second model (also in 2-D space, for clarity).

This second model might still include unrealistic trajectories, such as oscillating back and forth between two adjacent cells (when the corresponding speed components alternate from positive to negative and back) or staying forever in one cell, even with a positive speed.

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 $v_x$, $v_y$, $v_z$ 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 $v_x$ is nonnegative.

In comparison to the free motion model, Figure 5.5 shows the possible next states (in 2-D space) for a target whose speed components are $v_x = 3$ and $v_y = 3$ 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. The speed is derived from two subsequent position observations, thus a target needs to be able to move from one cell to an adjacent one (e.g., from $x = 3$ to $x = 4$) even when the corresponding speed in that direction is 0 ($v_x = 0$); at the next step, the speed will reflect this change ($v_x = 1$), as seen in Figure 5.6. 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 5.3.3.

### 5.3.2.2 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 as follows:
FIGURE 5.5 Possible movements from a state satisfying $v_x = 3$, $v_y = 3$ ("restricted" model).

FIGURE 5.6 Possible movements from a state satisfying $v_x = 1$, $v_y = 0$ ("restricted" model).
out: not in the monitored zone

\[ \equiv (x = 0) \land (y = 0) \land (z = 0) \]

taxi: on the ground, either at low speed or not with a runway heading

\[ \equiv (z = 1) \land ((v_x \leq TS \land |v_y| \leq TS) \lor (v_x \neq 0)) \]

takeoff: on the ground, with a runway heading, accelerating

\[ \equiv (z = 1) \land (|v_y| > TS) \land (v_x = 0) \land (a_y \geq 0) \]

rollout: on the ground, with a runway heading, decelerating

\[ \equiv (z = 1) \land (|v_y| > TS) \land (v_x = 0) \land (a_y < 0) \]

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

\[ \equiv (z > 1) \land (v_x = 0) \land (v_z > 0) \]

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

\[ \equiv (z > 1) \land (v_x = 0) \land (v_z \leq 0) \]

flythru: airborne, not in climbout or land mode

\[ \equiv (z > 1) \land (v_x \neq 0) \]

5.3.2.3 Setting the Alarm  The third and most important phase of the RSM algorithm is setting the alarm flag for every target. In pseudocode, this corresponds to a single variable assignment statement: Set the (Boolean) value of each alarm, based on different combinations of the current values of the other variables, as listed in the operational state matrix of Table 5.1.

### TABLE 5.1 Operational State Matrix for Setting the RSM Alarm

<table>
<thead>
<tr>
<th>Target \rightarrow Taxi</th>
<th>takeoff</th>
<th>climbout</th>
<th>land</th>
<th>rollout</th>
<th>flythru</th>
</tr>
</thead>
<tbody>
<tr>
<td>taxi</td>
<td>–</td>
<td>a \land f</td>
<td>a \land f</td>
<td>a \land f</td>
<td>a \land c \land f</td>
</tr>
<tr>
<td>takeoff</td>
<td>a \land f</td>
<td>d \lor e</td>
<td>d \lor e</td>
<td>d \lor e</td>
<td>a \lor d</td>
</tr>
<tr>
<td>climbout</td>
<td>a \land f</td>
<td>d \lor e</td>
<td>d \lor e</td>
<td>d \lor e</td>
<td>d \lor e</td>
</tr>
<tr>
<td>land</td>
<td>a \land f</td>
<td>d \lor e</td>
<td>d \lor e</td>
<td>d \lor e</td>
<td>a \lor d</td>
</tr>
<tr>
<td>rollout</td>
<td>a \land c \land f</td>
<td>a \lor d</td>
<td>a \lor d</td>
<td>a \lor d</td>
<td>d \lor e</td>
</tr>
<tr>
<td>flythru</td>
<td>–</td>
<td>b \land c</td>
<td>b \land c</td>
<td>b \land c</td>
<td>b \land c</td>
</tr>
</tbody>
</table>

a, distance closing; b, in takeoff or landing path; c, distance less than minimum separation; d, takeoff or landing in the same direction, less than minimum separation; e, takeoff or landing in the opposite direction, closing; f, taxi or stationary on or near runway.
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 analytic geometry can be avoided 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 $z_0 = 1$, $vx_0$, $vy_0 \leq TS$, $z_i = 1$, $vx_i = 0$, $|vy| > TS$, and $vz_0 = vz_i = 0$; that is, 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 that ownship is in front of the target, depending on which direction this is moving. Hence, in this situation, the predicate can be expressed as

$$a \equiv (vy_i > 0 \land y_0 > y_i) \lor (vy_i < 0 \land y_0 < y_i).$$

We can similarly express the other predicates as follows:

$$b \land c \equiv (vy_0 > 0 \land y_0 \leq y_i \leq y_0 + 1 \land |x_0 - x_i| \leq 1 \land z_i \leq 2)$$

$$(vy_0 < 0 \land y_0 - 1 \leq y_i \leq y_0 \land |x_0 - x_i| \leq 1 \land z_i \leq 2),$$

$$d \equiv vy_0 \cdot vy_i > 0 \land |x_0 - x_i| \leq 1 \land |y_0 - y_i| \leq 1,$$

$$e \equiv (vy_0 > 0 \land vy_i < 0 \land y_i \geq y_0) \lor (vy_0 < 0 \land vy_i > 0 \land y_i \leq y_0),$$

$$f \equiv 1 < x_i < max_x,$$

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

Table 5.2 shows the state-space measurements on the $SMAR\tilde{T}$ model with one target. Missing entries in the table correspond to parameter choices that required excessive runtime or memory.

Our attempts to use other tools have failed: The symbolic model checker NuSMV runs out of memory even before starting the generation, as the binary decision diagram (BDD) encoding of the transition relation is too large, while the explicit model checker 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.
5.3.3 Model Checking RSM

The verification effort was concentrated on determining whether the operational matrix in Table 5.1 ensures the absence of missed alarm scenarios. Since a formal description of this requirement was not included in the protocol specification, we had to explore different ways to express this property.

The following predicates are used to define the notions of interest (subscripts $o$ and $t$ refer to ownship and target, respectively):

$$\begin{align*}
\text{detect} &\equiv \text{phase}_o = \text{detect} \wedge \text{phase}_t = \text{detect} \\
\text{sep} &\equiv \text{distance}(o, t) > \text{minimum separation} \\
\text{alarm} &\equiv \text{alarm}_t = \text{true} \\
\text{track} &\equiv \text{status}_o \notin \{\text{taxi, flythruf}\} \lor \text{status}_t \notin \{\text{taxi, flythruf}\}
\end{align*}$$

### 5.3.3.1 Safety Property

Is there a tracked state where minimum separation is not satisfied and the alarm is off?

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

The model checker returns a witness to this condition where the predicate “distance is closing” is not satisfied in the current state. This is the case of the third snapshot of Figure 5.7. However, this might not correspond to an unwanted behavior, since the alarm might have been set in a previous state, when the minimum separation was first lost. The value of the alarm variable also depends on whether the alarm is “aged” or not for a few more cycles.

We observe that the “memoryless” nature of the query influences the result, as we look 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 violated.

5.3.3.2 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?

\[
\text{EF} (\text{detect} \land \text{track} \land \text{sep} \land \text{E}[(\neg \text{detect}) \lor (\text{detect} \land \neg \text{track} \land \neg \text{sep} \land \neg \text{alarm})])
\]

A witness for this query (see Fig. 5.8) 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 or landing path.” The second term is not satisfied; hence, no alarm is raised. Aircraft can actually collide (trajectories intersect in Fig. 5.8), while none of the participants gets a warning.

This case was corrected by the RSM developers by adding “distance less than minimum separation” as part of the criterion for this combination of states.

Note that we include the predicate \textit{track} in both states (before and after the transition), as incursions are defined for at least one aircraft taking off or landing. However, this additional constraint could mask some other undesired behaviors. Therefore, we next investigate a more general property.

5.3.3.3 A Stronger Safety Property

Is there a tracked state where minimum separation is lost, reachable without ever previously setting the alarm?
The model checker finds several scenarios that satisfy this query. As shown in Figure 5.9, the aircraft may enter the monitored area taxiing (not aligned to the runway) and already at close distance to each other. Note that this combination of states is explicitly ignored by the algorithm, since it does not fit the definition of an incursion. However, once on the runway, ownship can change direction and align itself to the runway. Thereafter, it is categorized as a takeoff (or climbout, if it becomes airborne). The other aircraft can stay within minimum separation, but not closing 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 two aircraft
at entry is very small, there might not be enough time for an escape maneuver, even if, later on, the alarm is indeed triggered by closing in.

Figure 5.9 shows a counterexample to this last safety property. An identical scenario exists for the airborne pair of states that is not tracked (the flythru status).

In order to determine whether this situation is of real concern or not, we have to look at possible continuations of the scenario after the potentially 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. One way for a malicious agent to perpetuate the problem is shown in Figure 5.10. The target can stay within minimum separation radius for a longer period of time if it “zigzags” and at each radar update has the same discretized distance to ownship. The target must zigzag to maintain the distance, since following a parallel path to ownship will cause RSM to consider it as taking off. The alarm criteria 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 zigzagging.

The case when the target is not an aircraft but a vehicle, such as a service truck, adds a degree of freedom for malicious behavior by the target (see scenario 5, Fig. 5.11). 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 ownship at close distance, and even continue chasing ownship after it is lined up for takeoff and
accelerating. No flag will be raised for the same reasons as in scenario 4. The RSM developers took into account our findings and eliminated the special treatment given to ground vehicles. This fully addresses the situation in scenario 5.

The situation in scenario 4 was not addressed. It was deemed of far less concern, as it is extremely difficult to realize in practice, even intentionally by a very skillful 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 is the only remaining unwanted behavior in the system, it serves as a validation for phase 3 of the RSM algorithm.

5.4 DISCUSSION

5.4.1 Lessons Learned

With all its inevitable shortcomings, the verification of RSM has had an undeniable value. The designers of the RSM algorithm were presented with a list of findings, which were not exposed during the testing activities, involving aircraft and ground vehicles, already under way at Dallas/Fort Worth International Airport.

Testing is still essential to certification, regardless of the costs, because it is performed on the product itself, not on abstract models. In some instances, however, the costs of testing can be considerable. Each of the two testing sessions (prior to our formal analysis team joining the effort) required the cooperation of several parties (airport officials, air traffic managers, airlines) to secure the resources: two airport runways reserved for an entire day, volunteer pilots, test evaluation engineers. Additionally, a custom-built van was needed to simulate an aircraft on the ground [53]. In light of these facts, it is arguably
more productive to include the formal verification task as early as possible in the design process.

The merit of the model-based technique is that, besides being considerably less expensive, it is more comprehensive. We were able to analyze all possible scenarios in our model and found scenarios of potential concern that occur with extremely low probability or under very peculiar conditions. 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 with the actual state-space sizes (of the order of $10^{13} - 10^{42}$ states, depending on the choice of parameters), this shows the need for exhaustive analysis. Another outcome of the formal analysis was that identifying the problems in the abstract models and suggesting modifications to the protocol to eliminate them has significantly increased the level of confidence in the correctness of the design.

The success was facilitated by many factors: There were clear specifications and a good path of communication between design engineers and the formal methods team, while the application itself had the “right” size (amenable for full state-space construction) and the “right” blend of continuous and discrete-state components.

5.4.2 Level of Effort

The verification of RSM was a medium-scale project with a duration of 12 months. A team of three people worked in various stages of the analysis, for a total of 9 man-months of work. Of this, approximately 6 man-months was dedicated to modeling decisions, during which three regular meetings with the design engineers were scheduled. The final 3 man-months was spent with the verification per se: formulating queries, analyzing, and synthesizing the answers. The understanding of RSM was greatly helped by the designers’ effort to keep the specification and implementation as clear 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 and to the novelty of the problem area for the modelers, the impact of efficient model-checking tools was crucial in completing the study.

5.4.3 Fault Tolerance

An aspect not covered in the above RSM analysis is fault tolerance. While our work verifies the correct operation of RSM under no-fault assumptions, the presence of faults on the data link may significantly impact the correct operation of the algorithm. This type of analysis requires the inclusion of probabilistic aspects in the model. A natural extension of this study is to include faulty behaviors, of either benign nature (missed or late updates) or malicious/Byzantine (inconsistent data between participants).
5.4.4 Challenges

Besides the habitual challenges of making model checking practical (mitigating huge state spaces, arguing for the soundness of the abstraction), the RSM experience also revealed some novel facts. First, a good discretization method and powerful tools could be quintessential in tackling the complexity of this type of applications. Some embedded applications immediately suggest a discretization rule. In this case, the algorithm uses snapshots of aircraft positions taken at regular time intervals, so that in fact the state of the system is already discretized. On the negative side, verifying the actual C code was still out of reach.

In what regard the efficiency of evaluating temporal logic formulae, not all types of properties could be efficiently verified. In general, for safety and reachability properties, good strategies exist, while liveness and fairness properties are more costly to evaluate.

Overall, we can conclude that model checking is a viable alternative for validating the correctness of avionics protocols, given its ability to find non-trivial errors that are usually not exposed during simulation or testing.

REFERENCES


