Invariant Synthesis for Verification of Parameterized Cyber-Physical Systems with Applications to Aerospace Systems

Taylor T. Johnson*

University of Texas at Arlington, Arlington, TX 76010, USA

Sayan Mitra†

University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA

In this paper, we describe a method for synthesizing inductive invariants for cyber-physical aerospace systems that are parameterized on the number of participants, such as the number of aircraft involved in a coordinated maneuver. The methodology is useful for automating the traditionally manual process of deductive verification of safety properties, such as collision avoidance, and establishes such properties regardless of the number of participants involved in a protocol. We illustrate the methodology using a simplified model of the landing protocol of the Small Aircraft Transportation System (SATS) as a case study. Each participant (aircraft) in the protocol is modeled as a hybrid automaton with both discrete and continuous states and potentially nondeterministic evolution thereof. Discrete states change instantaneously according to transitions and continuous states evolve according to rectangular differential inclusions. The invariant synthesis method enables a fully automatic verification of the main safety property of SATS, namely, safe separation of aircraft on approach to the runway. The method is implemented in a prototype verification tool called *Passel*. We present promising experimental results using the methodology, which has enabled a fully automatic proof of safe separation for the model of SATS.

Nomenclature

$\mathcal{N}$	Arbitrary number of participants
N	Fixed number of participants
$[\mathcal{N}]$	Set of arbitrary participant identifiers
$[\mathsf{N}]$	Set of fixed, finite number of participant identifiers
$\mathcal{A}(\mathcal{N}, i)$	Hybrid automaton template specifying participant $i \in [\mathcal{N}]$
$\mathcal{A}(i)$	Shorthand for $\mathcal{A}(\mathcal{N}, i)$ when $\mathcal{N}$ is clear from context
$\mathcal{A}^{\mathcal{N}}$	Parameterized network of hybrid automata
$\zeta(\mathcal{N})$	Safety property
$\Gamma(\mathcal{N})$	Candidate inductive invariant
P	Number of participants used in projection step of invariant synthesis

*Assistant Professor, Department of Computer Science and Engineering, 500 UTA Blvd., Arlington, TX 76010.

†Assistant Professor, Department of Electrical and Computer Engineering, 1308 W. Main St., Urbana, IL 61801, AIAA Member.

I. Introduction

Since aerospace software systems require a high degree of assurance that they meet their specifications, formal methods have been extensively applied during the development of such systems.[41] When successful, formal methods may prove that a model of a system meets its specification. This is unlike simulation or testing-based approaches that can provide counterexamples if a system is not correct, but cannot generally establish correctness if the system has a large (or infinite) state-space, like in virtually any cyber-physical aerospace system. A variety of techniques for verifying aerospace systems have been applied, such as partially manual, deductive verification approaches with automated theorem proving[30,35,38,26] to automated methods like reachability computations for model checking.[43,42,8,25,23]

In this paper, we present preliminary results on combining deductive verification with model checking to enable automatic verification of properties for parameterized cyber-physical aerospace systems. The class of systems we consider are parameterized on the number of interacting systems, for instance, the number of aircraft approaching a runway according to a landing protocol, the number of satellites in a constellation, or the number of unmanned aerial vehicles in a flock. We model each participant as hybrid automaton, and consider the composition of $\mathcal{N}$ such interacting automata, for an arbitrary choice of $\mathcal{N} \in \mathbb{N}$. Hybrid systems modeling frameworks[3,20,32,34,16,37,17] specify state machines with combinations of discrete and continuous states and their evolution. State machines model discrete variables and discrete transitions, while real-valued continuous variables evolve according to ordinary differential equations (ODEs), differential algebraic equations (DAEs), or inclusions.[14] We aim to verify that any number of interacting systems satisfies some property, e.g., that no collisions occur for any number of aircraft attempting to land.

In networks of hybrid automata, automata communicate by reading one another's state and through globally shared variables. A hybrid automaton $\mathcal{A}(i)$ may read the variables of another hybrid automaton $\mathcal{A}(j)$ by maintaining a *pointer* to $\mathcal{A}(j)$. A pointer is a variable that takes values in the set of automaton identifiers or names. Pointer variables allow for modeling systems with dynamic communication topologies. Many distributed protocols utilize this type of communication, such as traffic control protocols where vehicles keep track of adjacent vehicles, swarm robotics protocols where robots keep track of neighbors, or routers that keep track of successors.

One such aerospace system that is naturally modeled in this framework is the Small Aircraft Transportation System (SATS).[2,1,35,45,25] We present the simplified model SATS analyzed in this paper in Figures 1 and 2. SATS incorporates a landing protocol that has had several properties formally verified in the PVS theorem prover,[35,44,45] as well as using model checking.[25] In SATS, aircraft communicate by reading the valuations of discrete variables and continuous positions using pointers. Before attempting the landing procedure, each aircraft checks if any other aircraft already attempting to land are sufficiently far away—L_G distance—from the geographic start of the approach to the runway, measured along one-dimension. If so, an aircraft may begin an approach to the runway. The aircraft travel along the approach to the runway with velocities $\dot{x}[i] \in [v_L, v_U]$ for $0 < v_L \leq v_U$. After traversing the length L_B of the distance to the runway, the aircraft may either land, or return to a holding pattern. The main safety property in SATS safe separation, which specifies that if any aircraft i is ahead of any other aircraft j, then the positions of the aircraft are actually separated by at least L_S distance:

$$\forall i, j \in [\mathcal{N}]. \left(i \neq j \wedge q[i] = \mathsf{base} \wedge q[j] = \mathsf{base} \wedge x[i] > x[j]\right) \Rightarrow \left(x[i] - x[j] \geq L_S\right). \quad (1)$$

Here, $[\mathcal{N}]$ is the set $\{1, 2, \ldots, \mathcal{N}\}$, $x[i]$ is the real position of some aircraft i, $q[i]$ refers to the discrete location of some aircraft i, and L_S is a positive real constant. We aim to automatically prove safe separation regardless of the number of aircraft, $\mathcal{N}$. To prove safety properties, we follow the deductive verification approach of finding and establishing inductive invariants.

To synthesize inductive invariants, we follow the process of invisible invariants,[39,5] which was originally developed for discrete systems, but that we have extended for hybrid automata. We first compute the set of reachable states for a finite instantiation of the network, i.e., for $\mathsf{N} = 3$, although we reiterate that our aim is to prove properties for any choice of $\mathcal{N}$. This yields a set of constraints $\psi(1, 2, 3)$ describing the possible values for the variables of automata 1, 2, and 3. Next, we discard the constraints on the variables of automaton 3 by projecting to the variables of automaton 3, which yields a new constraint $\psi(1, 2)$ over only the variables of automata 1 and 2. Then, we generalize this constraint $\psi(1, 2)$ by syntactically replacing 1 with a symbol i and 2 with a symbol j. Finally, we create a new quantified constraint $\forall i, j \in [\mathcal{N}]. i \neq j \Rightarrow \psi(i, j)$. While this is a heuristic method, it allows us to prove collision avoidance (Equation 1) for any choice of $\mathcal{N}$ automatically.

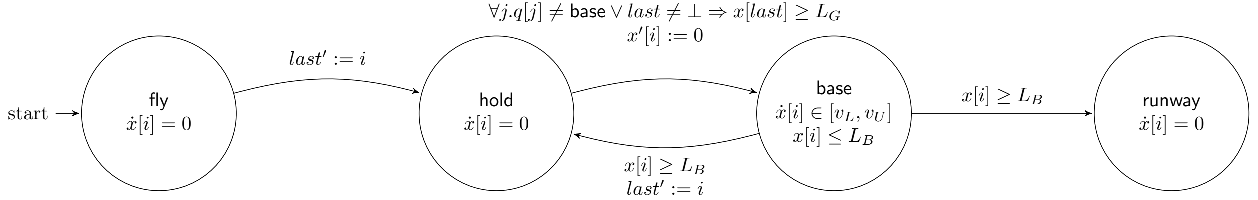

Figure 1. Simplified SATS protocol for aircraft $i \in [\mathcal{N}]$.

Outline. The formal framework for modeling and verifying cyber-physical aerospace systems as parameterized networks of hybrid automata is presented in Section II. The SATS case study is used as an example throughout Section II for illustrating the syntax and semantics of the modeling framework. In Section III, we present the inductive invariant synthesis method. In Section IV, we present a brief summary of experimental results for SATS using the *Passel* verification tool. Section V concludes the paper and presents directions for future work.

II. Formal Modeling Framework

In this section, we present the formal modeling framework for specifying and verifying parameterized networks of hybrid automata. The framework[22] generalizes and unifies the modeling frameworks developed in our prior work.[27,25,26] First, we present the syntax for specifying a hybrid automaton template. Next, we define the semantics of networks composed of (a potentially unbounded number of) instances of the template. Then, we formally define the uniform verification problem, which aims to establish properties for any number of participants in the network. The template is used as input for our *Passel* verification tool.

Preliminaries. We use two symbols for referring to the number of automata in a network. Where we use N, we mean a constant, numerical natural number, that is, a fixed natural number (e.g., $\mathsf{N} = 3$). Where we use $\mathcal{N}$, we mean a symbolic natural number, that is, $\mathcal{N}$ is some arbitrary natural number. For a natural number n, we define the set $[n] \triangleq \{1, \ldots, n\}$, and we use the sets $[\mathsf{N}]$ and $[\mathcal{N}]$ for indexing automata. For a set S, we define $S_\perp \triangleq S \cup \{\perp\}$.

For any natural number $\mathcal{N}$ and $i \in [\mathcal{N}]$, an individual hybrid automaton $\mathcal{A}(\mathcal{N}, i)$ is a (possibly nondeterministic) state machine with finitely many discrete locations and variables of various types like reals and indices.The state of $\mathcal{A}(\mathcal{N}, i)$ can change instantaneously through discrete transitions and its real-valued variables can evolve continuously over time according to *trajectories* specified by ordinary differential equations (ODEs) or inclusions.

A network of hybrid automata $\mathcal{A}^{\mathcal{N}}$ is a collection of $\mathcal{N}$ interacting instances of a template automaton $\mathcal{A}(\mathcal{N}, i)$, in which the transitions of each hybrid automaton can depend on the the state of certain other hybrid automata. We aim to establish properties that hold for the network $\mathcal{A}^{\mathcal{N}}$ for any choice of the natural number $\mathcal{N}$. We drop the argument $\mathcal{N}$ from $\mathcal{A}(\mathcal{N}, i)$ and write $\mathcal{A}(i)$ when $\mathcal{N}$ is clear from context. In a network $\mathcal{A}^{\mathcal{N}}$, the constituent automata may communicate over discrete transitions, but not through trajectories. That is, a transition taken by $\mathcal{A}(\mathcal{N}, i)$ can depend on and influence the the state of another automaton $\mathcal{A}(\mathcal{N}, j)$, but a trajectory of $\mathcal{A}(\mathcal{N}, i)$ depends on and influences only the state of $\mathcal{A}(\mathcal{N}, i)$.

The variables of the $\mathcal{N}$ automata in the network $\mathcal{A}^{\mathcal{N}}$ are described as arrays of length $\mathcal{N}$ of appropriate types. Real-typed variable may be updated continuously and/or discretely, while variables of other types are only updated discretely. Next, we introduce the syntax for specifying networks of hybrid automata by specifying one template hybrid automaton $\mathcal{A}(\mathcal{N}, i)$, and then introduce the semantics of the language to show how networks $\mathcal{A}^{\mathcal{N}}$ composed of $\mathcal{N}$ interacting instances of the template are modeled.

A. Syntax for Hybrid Automaton Template $\mathcal{A}(\mathcal{N}, i)$

In this section, we define the syntax for specifying a hybrid automaton template $\mathcal{A}(\mathcal{N}, i)$ used to construct parameterized networks of hybrid automata. We begin with some preliminary definitions.

VARIABLES. A *variable* is a name used for referring to state. A variable v is associated with a *type*—denoted $type(v)$—that defines a set of values the variable may take. The type of a variable may be:

(a) L: a finite set called the set of locations (defined below).

(b) $[\mathcal{N}]_\bot$: the set of automaton indices (identifiers) with the special element $\bot$ that is not equal to any index. A variable of this type is called a pointer variable or a pointer in short.

(c) $\mathbb{R}$: the set of real numbers.

A variable may be *local* with a name of the form $variable_name[i]$, or *global*, in which case the name does not have the index $[i]$. For example, $q[i] : \mathsf{L}$, $p[i] : [\mathcal{N}]_\bot$, and $x[i] : \mathbb{R}$ respectively define location, pointer, real, and typed local variables, while $g : [\mathcal{N}]_\bot$ is a global variable of index type. For a local variable $q[i]$, the array of variables $\langle q[1], q[2], \ldots, q[\mathcal{N}]\rangle$ is denoted by $\bar{q}^{\mathcal{N}}$. We write $\bar{q}$ when $\mathcal{N}$ is clear from context.

TERMS, FORMULAS, AND *Passel* ASSERTIONS. This section presents the syntax for formulas we use to specify the various syntactic components of a hybrid automaton template $\mathcal{A}(\mathcal{N}, i)$. Formulas are built-up from constants, variables, and terms of several types. Formulas are used for specifying the initial states and the state evolution of the network. The grammar for different types of terms is as follows:

$$\begin{aligned}\mathsf{ITerm} &::= \bot \mid 1 \mid \mathcal{N} \mid i \mid p[i]\\ \mathsf{DTerm} &::= l_c \mid q \mid q[\mathsf{ITerm}]\\ \mathsf{RTerm} &::= 0 \mid 1 \mid r_c \mid x \mid x[\mathsf{ITerm}]\end{aligned}$$

An index term (ITerm) is either (a) one of the constants $\bot$, 1, $\mathcal{N}$, an index variable i, or (b) a local pointer variable p referenced at an index variable i. The grammar does not allow arbitrary productions of recursive ITerms by restricting ITerms from being produced by $p[\mathsf{ITerm}]$—for example, $p[p[i]]$ is not an allowed term.[a]

Discrete terms (DTerm) and real terms (RTerm) are defined as specified in the grammar. For discrete terms, l_c is constant from L, q is a discrete variable, and $q[\mathsf{ITerm}]$ is a discrete array referenced at an index specified by an ITerm. For real terms, r_c is a real-valued numerical constant, x is a real variable, and $x[\mathsf{ITerm}]$ is a real array referenced at an index specified by an ITerm.

Real polynomials and constraints are built using the following grammar.

$$\begin{aligned}\mathsf{RPoly} &::= \mathsf{RTerm} \mid \mathsf{RPoly}_1 + \mathsf{RPoly}_2 \mid \mathsf{RPoly}_1 - \mathsf{RPoly}_2 \mid (\mathsf{RPoly}_1 * \mathsf{RPoly}_2)\\ \mathsf{RAtom} &::= \mathsf{RPoly} < 0\end{aligned}$$

RPoly_1 and RPoly_2 are shorter real polynomials joined by arithmetic operators—addition $+$, subtraction $-$, or multiplication $*$—to obtain longer polynomials. RAtom are used for specifying real constraints. Other comparison operators—like less than or equal ($\leq$), greater than or equal ($\geq$), greater than ($>$), and equality ($=$)—will be expressed using negation ($\neg$) and conjunction ($\wedge$) in the formulas we define next.

For a polynomial p generated by RPoly over n real variables $x_1, \ldots, x_n$ with k additive terms, $p = a_1 x_1^{e_{1,1}} * \ldots * x_n^{e_{n,1}} + \ldots + a_k x_1^{e_{1,k}} * \ldots * x_n^{e_{n,k}}$, with real coefficients $a_1, \ldots, a_k$ and natural number exponents $e_{1,1}, \ldots, e_{n,k}$, the degree of p is $deg(p) \triangleq \max_{1\leq q\leq k}(\sum_{r=1}^{n} e_{r,q})$. If $deg(p) > 1$, then p is *nonlinear*. If $deg(p) \leq 1$, then p is *linear*. The linear fragment is the subset of formulas where all polynomials have degree at most one. We assume standard precedence of operators (e.g., $*$ before $+$, etc.).

Using these terms and constraints, formulas are defined next:

$$\begin{aligned}\mathsf{Atom} &::= \mathsf{ITerm}_1 < \mathsf{ITerm}_2 \mid \mathsf{DTerm}_1 = \mathsf{DTerm}_2 \mid \mathsf{RAtom}\\ \mathsf{Formula} &::= \mathsf{Atom} \mid \neg\mathsf{Formula} \mid \mathsf{Formula}_1 \wedge \mathsf{Formula}_2 \mid \exists x\ \mathsf{Formula}\end{aligned}$$

Here, x is called a *bound variable*, and is a variable of one of the types. $\mathsf{Formula}_1$ and $\mathsf{Formula}_2$ are shorter formulas that are joined by Boolean operators to obtain a longer formula. By combining the Boolean operators $\wedge$ and $\neg$ with the $<$ operator, other comparison operators, such as $=$, $\neq$, $\leq$, $>$, and $\geq$, can be expressed in formulas for indices and reals. For example, $p_1[i] = p_2[j]$ can be written as $\neg(p_1[i] < p_2[j]) \wedge \neg(p_2[j] < p_1[i])$. Universally quantified variables can be expressed by $\neg\exists x : \mathsf{Formula} \equiv \forall x : \neg\mathsf{Formula}$. Thus, we assume the language contains the standard quantifiers and Boolean operators, even if not explicitly

[a] This restriction ensures that the theory is stratified.[5]

specified by the grammar (e.g., universal quantification $\forall$, implication $\Rightarrow$, disjunction $\vee$, less-than-or-equal $\leq$, non-equality $\neq$, etc.).

If a variable in a formula is not bound, then it is called a *free variable*. If a formula does not contain any quantifiers, then it is *quantifier-free*, but otherwise it is *quantified*. If a formula has no free variables, then it is a *sentence*. If a formula is quantified and all the bound variables appearing in it are of index type, then it is *index-quantified*. An *index sentence* is a formula with no free variables of index type.

Arithmetic operations on index terms (ITerm) are not allowed, and the allowed comparisons mean only total orders may be specified. Only equality (or non-equality) comparisons are allowed for discrete terms. If a formula is only composed of RTerms , then it is in the real polynomial subclass. A formula ϕ is in disjunctive normal form (DNF) if and only if it is a disjunction of conjunctive clauses, where a conjunctive clause is one or more conjunctions of one or more atoms. A formula ϕ is in conjunctive normal form (CNF) if and only if it is a conjunction of disjunctive clauses, where a disjunctive clause is one or more disjunctions of one or more atoms.

The *Passel assertion language* is the set of index-quantified formulas generated by the grammar just defined. For a formula ϕ, let $vars(\phi)$ be the set of variables appearing in ϕ. For a formula ϕ, let $ivars(\phi)$ be the set of distinct index variables appearing in ϕ. For a formula ϕ, let $free(\phi)$ be the set of free variables appearing in ϕ. For a formula ϕ, let $bound(\phi)$ be the set of bound variables appearing in ϕ. For a quantified formula ϕ, let $body(\phi)$ be the body of the quantifier, with all bound variables $bound(\phi)$ replaced with universally instantiated variables with the same names. For a set of variable names V and a formula ϕ, if $free(\phi) \subseteq \mathsf{V}$, then ϕ is *over* V. For a set of variable names V and a formula ϕ, if $free(\phi) = \mathsf{V}$, then ϕ is *over all* V. For a *Passel* assertion over a set of variables V, we always assume that a countable set of symbolic automaton indices are included in V for referencing different variables. *Passel* assertions over particular sets of variables—along with further restrictions, such as being quantifier-free—will be used for specifying various syntactic components of the hybrid automaton template $\mathcal{A}(\mathcal{N}, i)$.

Hybrid Automaton Template. We next define a syntactic structure called a hybrid automaton template, which we use to specify the behavior of a participant in a parameterized network.

Definition 1. *For symbolic constants $\mathcal{N} \in \mathbb{N}$ and $i \in [\mathcal{N}]$, a* hybrid automaton template $\mathcal{A}(\mathcal{N}, i)$ *is specified by the following syntactic components:*

(a) V_i*: a finite set of variable names,*

(b) L*: a finite set of location names,*

(c) Init_i*: an initial condition, which is a Passel assertion over* V_i*,*

(d) Trans_i*: a finite set of discrete transition statements, each of which is composed of a from-to pair of locations, along with a guard, a universal guard, and an effect, which are quantifier-free Passel assertions over* $\mathsf{V}_i \cup \mathsf{V}'_i$*, where* V'_i *is the set of primed variable names corresponding to* V_i*, and*

(e) Flow_i*: a finite set of trajectory statements, one for each element in* L*, each of which is composed of an invariant, a stopping condition, and a flowrate, each of which are quantifier-free Passel assertions over* $\mathsf{V}_i \cup \mathsf{V}_i_dot$*, where* V_i_dot *is the set of dotted variable names corresponding to the real-valued variables in* V_i*.*

The subscript i emphasizes that components may use the automaton's index.

A hybrid automaton template $\mathcal{A}(\mathcal{N}, i)$ is written $\mathcal{A}(i)$ when $\mathcal{N}$ is clear from context. Throughout this section, we use an example specification of a simplified version of the Small Aircraft Transportation System (SATS) to illustrate the language constructs available for specifying a hybrid automaton template $\mathcal{A}(i)$. The specification of the protocol in this language is shown in Figure 2, and an equivalent graphical representation appears in Figure 1.[b]

Specifying Locations. The set of location names L is specified by a list of location names. A location name follows the keyword `location name`. In SSATS, the set of locations L is $\{\mathsf{fly}, \mathsf{hold}, \mathsf{base}, \mathsf{runway}\}$ (lines 12, 14, 16, and 20). Locations are depicted graphically as the circles in Figure 1. Each automaton $\mathcal{A}(i)$ has a single local variable $q[i]$ that takes values in L. A trajectory statement may follow each location name, defined in detail in below.

[b]Figure 2 is marked-up for readability, but is essentially in *Passel*'s input language.

```
1   parameter name='L_B' type='real' value = 28.0    // base zone length
    parameter name='L_S' type='real' value = 7.0     // separation spacing
3   parameter name='v_L' type='real' value = 90.0    // minimum velocity
    parameter name='v_U' type='real' value = 120.0   // maximum velocity
5   parameter name='L_G' type='real' value = L_S + (v_U - v_L) * ((L_B - L_S) / v_L) // guard spacing

7   automaton name='SSATS'
      variable name='q[i]' type='L'        // location local variable
9     variable name='x[i]' type='real'     // continuous local variable
      variable name='last' type='index'    // global lock variable
11
      location name='fly'
13      flowrate: x[i]_dot = 0.0
      location name='hold'
15      flowrate: x[i]_dot = 0.0
      location name='base'
17      inv: x[i] <= L_B
        stop: x[i] = L_B
19      flowrate: x[i]_dot >= v_L and x[i]_dot <= v_U
      location name='runway'
21      flowrate: x[i]_dot = 0

23    transition from='fly' to='hold'
        eff: x[i]' = 0.0 and last' = i
25
      transition from='hold' to='base'
27      grd: last = i
        ugrd: q[j] != base
29      eff: x[i]' = 0.0

31    transition from='hold' to='base'
        grd: last != ⊥ and last != i and x[last] >= L_G
33      eff: x[i]' = 0.0

35    transition from='base' to='hold'
        grd: x[i] >= L_B
37      eff: x[i]' = 0 and last' = i

39    transition from='base' to='runway'
        grd: x[i] >= L_B and last = i
41      eff: x[i]' = 0 and last' = ⊥

43    transition from='base' to='runway'
        grd: x[i] >= L_B and last != i
45      eff: x[i]' = 0

47  initially: forall i (q[i] = fly and next[i] = ⊥ and last = ⊥)
    property: forall i, j ( i != j and q[i] = base and q[j] = base and x[i] > x[j] )
49            implies (x[i] >= x[j] + L_S)
```

Figure 2. ***Passel*** **input file specifying hybrid automaton $\mathcal{A}(i)$ for SSATS, a simplified SATS protocol.**

Specifying Variables, Parameters, Initial Conditions, and Invariant Properties. The set of variables V_i is specified by the list of variable names and types following the keywords `variable name` and `type`. For a SATS template automaton with index i, the set of variables is specified by the list of variables on lines lines 8 through 10. It has two local variables, $q[i]$ and $x[i]$, with types L and $\mathbb{R}$, and a single global variable *last* of type $[\mathcal{N}]_\perp$.

The specification of $\mathcal{A}(i)$ may use a set of symbolic or numerical parameters (constants). Each parameter is specified by its name, type, and, optionally, a quantifier-free *Passel* assertion that specifies constraints that the parameters must satisfy. For SATS, there are five real-valued parameters, L_B, L_S, v_L, v_U, and L_G (lines 1, 2, 3, 4, and 5).

We denote the set of local variables by $\mathsf{V}_L[i]$, the set of global variables by $\mathsf{V}_G[i]$, and the set of parameters by $\mathsf{V}_P[i]$. In the SATS example, $\mathsf{V}_L[i] = \{q[i], x[i]\}$, $\mathsf{V}_G[i] = \{last\}$, and $\mathsf{V}_P[i] = \{L_S, L_B, v_L, v_U, L_G\}$. When clear from context, we drop the index i and write V_L, V_G, and V_P for $\mathsf{V}_L[i]$, $\mathsf{V}_G[i]$, and $\mathsf{V}_P[i]$, respectively.

Initial Conditions. The initial condition assertion Init_i is a universally index-quantified *Passel* assertion following the keyword `initially`. In SATS, the initial condition assertion is (line 47):

$$\forall i : \ (q[i] = \mathsf{fly} \ \wedge \ x[i] = 0 \ \wedge \ last \ = \perp),$$

where i is implicitly quantified over $[\mathcal{N}]$. The initial condition assertion for SATS asserts that, for each index $i \in [\mathcal{N}]$, the variables of $\mathcal{A}(i)$ have the constraints $q[i] = \mathsf{fly}$ and $x[i] = 0$, and that the global variable $last = \bot$. If a variable $v \in \mathsf{V}_i$ is not specified in the initial condition, it is assumed that v is initially an arbitrary value in its type $type(v)$. Note that the Init_i assertion may specify constraints over all automata in the network using universally index-quantified *Passel* assertions.

Candidate Invariant Properties. Candidate invariant properties are specified as *Passel* assertions following the keyword `property`. For example, the safe separation invariant property can be specified as in Equation 1 (line 49).

Specifying Discrete Transitions. For any $\mathcal{N} \in \mathbb{N}$ and any $i \in [\mathcal{N}]$, the set of discrete transitions Trans_i is specified by the list of transition statements following the keyword `transition`. Each transition statement specifies a from-to pair of locations following the keywords `from` and `to`. If it exists, we denote a transition from location src to location dest by $\mathsf{t}(\mathsf{src}, \mathsf{dest}) \in \mathsf{Trans}_i$, which is written as t when the from-to locations are clear from context.

Each transition $\mathsf{t} \in \mathsf{Trans}_i$ *may* specify a *guard* following the keyword **grd**, a *universal guard* following the keyword **ugrd**, and an *effect* following the keyword **eff**. The guard, universal guard, and effect are quantifier-free *Passel* assertions, and they are denoted by $\mathbf{grd}(\mathsf{t}, i)$, $\mathbf{ugrd}(\mathsf{t}, i)$, and $\mathbf{eff}(\mathsf{t}, i)$ for $\mathcal{A}(i)$, respectively. If i is clear from context, we drop it and write $\mathbf{grd}(\mathsf{t})$, $\mathbf{ugrd}(\mathsf{t})$, or $\mathbf{eff}(\mathsf{t})$. The universal guard is a quantifier-free *Passel* assertion involving the variables V_j, for $j \neq i$, and we recall i is the index of the template $\mathcal{A}(\mathcal{N}, i)$. The universal guard specifies an assertion over the variables of other automata and global variables. Such assertions over the variables of all the other automata in the network are useful for modeling broadcast-like communications. The effect models the update of state, and is a quantifier-free *Passel* assertion over the variables $\mathsf{V}_i \cup \mathsf{V}'_i$, where V'_i concatenates a prime ($'$) to each variable name v in V_i. The effect specifies a relation between the variables before (unprimed) and after (primed) the transition. The effect is not required to specify variables that are not modified by the transition.

One of the transitions from hold to base for SATS with index i (line 26), where

$$\begin{aligned} \mathbf{grd}: &\ last = i, \\ \mathbf{eff}: &\ x[i]' = 0.0 \end{aligned}$$

specifies that automaton i *may* nondeterministically transition from a state where $q[i] = \mathsf{hold}$ to a state where $q[i]' = \mathsf{base}$, only if the global pointer variable $last$ is equal to i. Further, if the automaton *does* make this transition, then the effect specifies that $x[i]$ is to be reset to 0. If the guard condition is omitted, then it is assumed to be just the control location condition. For example, in SATS, the transition from fly to hold is enabled when $q[i] = \mathsf{fly}$.

Specifying Continuous Dynamics. The elements of the set of trajectory statements Flow_i are listed following the corresponding location names. Each location $\mathsf{m} \in \mathsf{L}$ has a trajectory statement in Flow_i. The trajectory statement consists of an invariant condition following the keyword **inv**, a stopping condition following the keyword **stop**, and a sequence of flow rates following the keyword **flowrate**. The invariant condition of a location $\mathsf{m} \in \mathsf{L}$ for automaton $\mathcal{A}(i)$ is denoted by $\mathbf{inv}(\mathsf{m}, i)$, the stopping condition is denoted by $\mathbf{stop}(\mathsf{m}, i)$, and the flow rate for some real-valued variable $x[i] \in \mathsf{V}_i$, is denoted by $\mathbf{flowrate}(\mathsf{m}, x[i])$.

The invariant and stopping conditions are *Passel* assertions involving only the real variables $\mathsf{X}[i]$ and real parameters in $\mathsf{V}_P[i]$. The flow rate associates each real-valued variable in $\mathsf{X}[i]$ with ordinary differential equations or inclusions specified as the real polynomial subclass of *Passel* assertions. The flow rate for a variable name $x[i] \in \mathsf{V}_i$ is specified by concatenating *_dot* to the variable name, for example $x[i]_dot$.

Together, the components of a trajectory statement define how variables of $\mathcal{A}(i)$ behave over intervals of time. For SATS, the trajectory statement for base is:

$$\begin{aligned} &\mathbf{inv}: x[i] \leq L_B \\ &\mathbf{stop}: x[i] = L_B \\ &\mathbf{flowrate}: x[i]_dot \geq v_L \wedge x[i]_dot \leq v_U \end{aligned}$$

In addition, the invariant requires that the automaton with index i can have $q[i] = \mathsf{base}$ only as long as $x[i] \leq L_B$. The stopping condition requires that if $x[i] = L_B$, then real time cannot continue to elapse. The

flow rate specifies the special case of *rectangular differential inclusions*, where the time-derivative is specified by an upper and a lower bound in terms of a numerical constant or a parameter name, in this case, lower and upper velocity bounds (v_L and v_U, repsectively).

B. Semantics of Hybrid Automata Networks

The semantics of the hybrid automata network $\mathcal{A}^{\mathcal{N}}$—where an arbitrary number $\mathcal{N} \in \mathbb{N}$ of instances of the template $\mathcal{A}(\mathcal{N}, i)$ operate in parallel—is defined in this section. The semantics are defined in terms of a transition system with a set of variables $\mathsf{V}^{\mathcal{N}}$, a set of states $Q^{\mathcal{N}}$, a set of initial states $\Theta^{\mathcal{N}}$, and a transition relation $T^{\mathcal{N}}$. For networks of hybrid automata, none of these sets is usually finite since variables may have real types.

Definition 2 (Parameterized Network of Hybrid Automata)**.** *For any* $\mathcal{N} \in \mathbb{N}$*, a* parameterized network of hybrid automata *is a tuple* $\mathcal{A}^{\mathcal{N}} \triangleq \left\langle \mathsf{V}^{\mathcal{N}}, Q^{\mathcal{N}}, \Theta^{\mathcal{N}}, T^{\mathcal{N}} \right\rangle$*, where*

(a) $\mathsf{V}^{\mathcal{N}}$ *are the variables of the network,* $\mathsf{V}^{\mathcal{N}} \triangleq \mathsf{V}_G \cup \bigcup_{i=1}^{\mathcal{N}} \mathsf{V}_L[i]$*,*

(b) $Q^{\mathcal{N}} \subseteq val(\mathsf{V}^{\mathcal{N}})$ *is the state-space,*

(c) $\Theta^{\mathcal{N}} \subseteq Q^{\mathcal{N}}$ *is the set of initial states, and*

(d) $T^{\mathcal{N}} \subseteq Q^{\mathcal{N}} \times Q^{\mathcal{N}}$ *is the transition relation, which is partitioned into disjoint sets of discrete transitions* $\mathcal{D}^{\mathcal{N}} \subseteq Q^{\mathcal{N}} \times Q^{\mathcal{N}}$ *and continuous trajectories* $\mathcal{T}^{\mathcal{N}} \subseteq Q^{\mathcal{N}} \times Q^{\mathcal{N}}$*.*

The transition system is said to be parameterized on $\mathcal{N}$ since fixing different values of $\mathcal{N}$ yield different transition systems. This definition allows for proving an invariant property ζ for *every* network of hybrid automata. For instance $\forall \mathcal{N} \in \mathbb{N} : \left(\mathcal{A}^{\mathcal{N}} \models \zeta(\mathcal{N})\right)$ states that, for every choice of $\mathcal{N} \in \mathbb{N}$, the corresponding network of hybrid automata $\mathcal{A}^{\mathcal{N}}$ satisfies the property $\zeta(\mathcal{N})$.

State-Space and Semantics of *Passel* Assertions. Recall that V_i is the set of variable names for the hybrid automaton template $\mathcal{A}(\mathcal{N}, i)$ where $i \in [\mathcal{N}]$. A state $\mathbf{x}$ in $Q^{\mathcal{N}}$ of $\mathcal{A}^{\mathcal{N}}$ is defined in terms of the valuations of all the variables of all its components. For each $i \in [\mathcal{N}]$, the *valuation* of a variable $v \in \mathsf{V}_i$ is a function that associates the variable name v to a value in its type $type(v)$. Elements of the state-space $Q^{\mathcal{N}}$ are called *states* and are denoted by boldface $\mathbf{v}$, $\mathbf{v}'$, etc.

At a state $\mathbf{v}$, the valuation of a particular local variable $x[i] \in \mathsf{V}_L[i]$ for automaton $\mathcal{A}(i)$ is denoted by $\mathbf{v}.x[i]$, and $\mathbf{v}.g$ for some global variable g in $\mathsf{V}_G[i]$. We recall that we refer to the valuations of a local variable $v[i] \in \mathsf{V}_L$ of all $\mathcal{N}$ automata in the network $\mathcal{A}^{\mathcal{N}}$ as an array variable, and denote it $\bar{v}$ which takes values in $type(v[i])^{\mathcal{N}}$. So, for a state $\mathbf{v}$, the valuations of a local variable $v[i] \in \mathsf{V}_L[i]$ for all $i \in \mathcal{N}$ is written $\mathbf{v}.\bar{v}$. The valuation of all the local variables for automaton $\mathcal{A}(i)$ at state $\mathbf{v}$ is denoted by $\mathbf{v}.\mathsf{V}_L[i]$. The valuation of all the local variables for automaton $\mathcal{A}(i)$, as well as the global variables, at state $\mathbf{v}$ is denoted by $\mathbf{v}.\mathsf{V}_i$. The state-space Q_i corresponding to automaton $\mathcal{A}(\mathcal{N}, i)$ in the network $\mathcal{A}^{\mathcal{N}}$ is defined as $Q_i \triangleq val(\mathsf{V}_i)$.

Representing States with Assertions. Subsets of $Q^{\mathcal{N}}$ are often represented by assertions involving the variables. If a state $\mathbf{v}$ satisfies a formula ϕ—that is, the corresponding variable valuations result in ϕ evaluating to *true*—we write $\mathbf{v} \models \phi$. For such a formula ϕ, the corresponding states satisfying ϕ are denoted by $[\![\phi]\!]$. A *model* for an assertion provides interpretation to the elements appearing in the assertion.

Definition 3. *An* n-model M *for an assertion* ψ *is denoted* $M(n, \psi)$ *and provides an interpretation of each the free variables in* ψ *as follows:*

- *the index constants* $\perp$*,* 1*, and* $\mathcal{N}$ *are respectively assigned the values* 0*,* 1*, and* n*,*
- *each pointer variable is assigned a value in the set* $[n]$*,*
- *each discrete variable is assigned a value in* L*,*
- *each real variable is assigned a value in* $\mathbb{R}$*,*
- *each pointer, discrete, and real array is assigned respectively a* $\{0, \ldots, n\}$*-valued,* L*-valued, or real-valued array of length* n*.*

Given an assertion ψ and a model $M(n, \psi)$, if ψ evaluates to true with the interpretations of the free variables given by $M(n, \psi)$, then $M(n, \psi)$ is said to *satisfy* ψ. If there exist models that satisfy ψ, then the assertion is said to be *satisfiable*. If all models of ψ satisfy it, then the assertion is said to be *valid*.

Initial States. The set of initial states $\Theta^{\mathcal{N}} \subseteq Q^{\mathcal{N}}$ is defined as $[\![\mathsf{Init}_i]\!]$, that is, the set of states satisfying the *Passel* assertion Init_i: $\Theta^{\mathcal{N}} \triangleq \{\mathbf{v} \in Q^{\mathcal{N}} \mid \mathbf{v} \models \mathsf{Init}_i\}$. In SATS (Figure 2), the set of initial states specified by line 47 is:

$$\Theta^{\mathcal{N}} \triangleq [\![\mathsf{Init}_i]\!] = \{\mathbf{x} \in Q^{\mathcal{N}} \mid \forall i \in [\mathcal{N}], \mathbf{x}.q[i] = \mathsf{fly} \wedge \mathbf{x}.x[i] = 0.0 \wedge \mathit{last} = \bot\}.$$

Transitions and Trajectories. The evolution of the states of $\mathcal{A}^{\mathcal{N}}$ are describing by a transition relation $T^{\mathcal{N}} \subseteq Q^{\mathcal{N}} \times Q^{\mathcal{N}}$. For a pair $(\mathbf{v}, \mathbf{v}') \in T^{\mathcal{N}}$, we use the notation $\mathbf{v} \rightarrow \mathbf{v}'$, where $\mathbf{v}$ is called the *pre-state* and $\mathbf{v}'$ is called the *post-state*. There are two ways state is updated by $T^{\mathcal{N}}$: discrete transitions $\mathcal{D}^{\mathcal{N}}$ model instantaneous change of state and continuous trajectories $\mathcal{T}^{\mathcal{N}}$ model change of state after a time interval.

Discrete Transitions. Discrete transitions model atomic, instantaneous updates of state due to *one* automaton in the network $\mathcal{A}^{\mathcal{N}}$. There is a discrete transition $\mathbf{v} \rightarrow \mathbf{v}' \in \mathcal{D}^{\mathcal{N}}$ iff:

$$\exists i \in [\mathcal{N}]\ \exists \mathsf{t} \in \mathsf{Trans}_i : \mathbf{v}.\mathsf{V}_i \models \mathbf{grd}(\mathsf{t}, i) \wedge \mathbf{v}'.\mathsf{V}_i \models \mathbf{eff}(\mathsf{t}, i) \wedge \\ (\forall j \in [\mathcal{N}] : \mathbf{v}.\mathsf{V}_j \models \mathbf{ugrd}(\mathsf{t}, j) \wedge j \neq i \Rightarrow \mathbf{v}'.\mathsf{V}_j = \mathbf{v}.\mathsf{V}_j).$$

From the pre-state $\mathbf{v}$, any automaton in the network, with any transition satisfying the guard *may* update its post-state according to transition effect, while the states of the other automata remain unchanged. Informally, a discrete transition from pre-state $\mathbf{v}$ to post-state $\mathbf{v}'$ models the discrete transition of one particular hybrid automaton $\mathcal{A}(i)$ by some transition $\mathsf{t} \in \mathsf{Trans}_i$. The universal guard of transition t depends on the variables in V_j for $j \neq i$.

We recall that the guard is a quantifier-free *Passel* assertion and specifies the enabling condition for the transition, which is a condition that must evaluate to true to allow the transition to update the system state. If the guard or universal guard are not specified, we assume they are *true*, which means a transition $\mathsf{t}(\mathsf{src}, \mathsf{dest})$ may only be taken by automaton $\mathcal{A}(i)$ if $q[i] = \mathsf{src}$. We assume the identity relation for any primed variable $v' \in \mathsf{V}_i'$ not specified in an effect. For example, if $x[i]'$ is not specified in an effect, then we assume the specified effect is conjuncted with $x[i]' = x[i]$.

For the SATS example, the semantics for the discrete transition $\mathsf{t}(\mathsf{base}, \mathsf{runway})$ (line 39) for some automaton $\mathcal{A}(i)$ are defined by:

$$\begin{aligned}(\mathbf{v}, \mathbf{v}') \models \exists i \in [\mathcal{N}]\ &: (q[i] = \mathsf{base} \wedge \mathit{last} = i \wedge x[i] \geq L_B) \wedge \\ &\quad (q[i]' = \mathsf{runway} \wedge x[i]' = 0.0 \wedge \mathit{last}' = \bot) \wedge \\ (\forall j \in [\mathcal{N}]\ &: j \neq i \Rightarrow q[j]' = q[j] \wedge x[j]' = x[j]).\end{aligned}$$

This transition can occur from states $\mathbf{v}$ where $\mathbf{v} \models q[i] = \mathsf{base} \wedge x[i] \geq L_B \wedge \mathit{last} = i$. This transition updates the location of $\mathcal{A}(i)$ to be $q[i]' = \mathsf{runway}$, the real variable $x[i]' = 0$, and sets the global variable last to $\bot$, and additionally, the variables of no other automata $\mathcal{A}(j)$ are updated.

Continuous Trajectories. Continuous trajectories model update of state over intervals of time. There is a trajectory $\mathbf{v} \rightarrow \mathbf{v}' \in \mathcal{T}^{\mathcal{N}}$ iff some amount of time—t_e—can elapse from $\mathbf{v}$, such that, (a) the states of *all* automata in the network $\mathcal{A}^{\mathcal{N}}$ are updated to $\mathbf{v}'$ according to their trajectory statements, (b) while ensuring the invariants of *all* automata along the entire trajectory, and (c) that if the stopping condition of *any* automaton is satisfied, it is at the end of a trajectory. Formally, trajectories are defined as solutions of differential equations or inclusions specified in the trajectory statements of $\mathcal{A}(i)$. The differential equation $\dot{x} = f(x)$ where $x \in \mathbb{R}^n$ and $f : \mathbb{R}^n \rightarrow \mathbb{R}^n$ has a solution for initial condition $x_0 \in \mathbb{R}^n$ if there exists a differentiable function $\gamma(t)$ for $\gamma : \mathbb{R}_{\geq 0} \rightarrow \mathbb{R}^n$ such that $\gamma(0) = x_0$ and, for every $\tau \in [0, t]$, $\dot{\gamma}(\tau) = f(\gamma(\tau))$. A differential inclusion is $\dot{x} \in F(x)$ for $x \in \mathbb{R}^n$, where F is a set-valued function from $\mathbb{R}^n$ to $\mathbb{R}^n$, so that $F(x) \subseteq \mathbb{R}^n$. A solution for the differential inclusion with initial condition x_0 is any differentiable function $\gamma(t)$ for $\gamma : \mathbb{R}_{\geq 0} \rightarrow \mathbb{R}^n$ such that $\gamma(0) = x_0$ and, for every $\tau \in [0, t]$, $\dot{\gamma}(\tau) \in F(\gamma(\tau))$. Sufficiently smooth differential equations satisfying continuity conditions—such as Lipschitz continuity[28]—have unique solutions, whereas differential inclusions have families of solutions.[14, 40]

Thus, to define trajectories for $\mathcal{A}^{\mathcal{N}}$ formally, we first define a set-valued function called $\mathbf{flow}(\mathsf{m}, \mathbf{v}.\mathsf{V}_L[i], t)$ that returns the states of $\mathcal{A}(i)$ when $q[i] = \mathsf{m}$ that can be reached from $\mathbf{v}.\mathsf{V}_L[i]$ in t time. We suppose $\mathbf{flow}(\mathsf{m}, \mathbf{v}.\mathsf{V}_L[i])$ is set-valued, as this subsumes the case when $\mathbf{flowrate}(\mathsf{m}, x[i])$ specifies a differential equation for $\dot{x}[i]$ instead of an inclusion.

Let $n = |\mathsf{X}[i]|$ be the number of continuous local variables in the template $\mathcal{A}(i)$. Let $[\![\mathbf{flowrate}(\mathsf{m}, \mathsf{X}[i])]\!] : \mathbb{R}^n \to \mathbb{R}^n$ be the vector of differential inclusions (and/or equations) for all the continuous variables $\mathsf{X}[i]$ of $\mathcal{A}(i)$, assumed to be ordered lexicographically by the variable names. For example, for SATS in base, $[\![\mathbf{flowrate}(\mathsf{base}, \mathsf{X}[i])]\!]$ is $v_L \le \dot{x}[i] \le v_U$ (Figure 2, line 19) since there is a single continuous variable $x[i]$ specified to evolve according to the rectangular differential inclusion with lower and upper bounds v_L and v_U, respectively.

Here, $\mathbf{v}'.\mathsf{V}_L[i] = \mathbf{flow}(\mathsf{m}, \mathbf{v}.\mathsf{V}_L[i], t)$ iff:[c]

(a) for each real local variable $x[i] \in \mathsf{X}[i]$, $\dot{x}[i]$ with initial condition $\mathbf{v}.x[i]$ has a solution $\gamma(t)$, $\mathbf{v}'.x[i] = \gamma(t)$, and

(b) for each non-real local variable $y[i] \in \mathsf{V}_L[i] \setminus \mathsf{X}[i]$, $\mathbf{v}'.y[i] = \mathbf{v}.y[i]$.

For the SATS example for base (Figure 2, line 16),[d]

$$\mathbf{flow}(\mathsf{base}, \mathbf{v}.x[i], t) \in \ [\mathbf{v}.x[i] + v_L * t, \mathbf{v}.x[i] + v_U * t] \, .$$

Thus far, we have not included the invariant and stopping condition, so we include these to complete the definition of trajectories. There is a trajectory $\mathbf{v} \to \mathbf{v}' \in \mathcal{T}^{\mathcal{N}}$ iff:

$$\begin{aligned} \exists t_e \in \mathbb{R}_{\ge 0}\ \forall i \in [\mathcal{N}]\ \exists \mathsf{m} \in \mathsf{L}\ \forall t_p \le t_e\ :\ & \mathbf{flow}(\mathsf{m}, \mathbf{v}.\mathsf{X}[i], t_p) \models \mathbf{inv}(\mathsf{m}, i)\ \wedge \\ & (\mathbf{flow}(\mathsf{m}, \mathbf{v}.\mathsf{X}[i], t_p) \models \mathbf{stop}(\mathsf{m}, i) \Rightarrow t_p = t_e)\ \wedge \\ & \mathbf{v}'.\mathsf{X}[i] \in \mathbf{flow}(\mathsf{m}, \mathbf{v}.\mathsf{X}[i], t_e). \end{aligned}$$

For each $i \in [\mathcal{N}]$ and each real variable $x[i] \in \mathsf{X}[i]$, $\mathbf{v}.x[i]$ must evolve to the valuations $\mathbf{v}'.x[i]$, in exactly t_e time in some location $\mathsf{m} \in \mathsf{L}$ according to the flow rates allowed for $x[i]$ in that location. All intermediate states along the trajectory must also satisfy the invariant $\mathbf{inv}(\mathsf{m}, i)$, and if an intermediate state satisfies $\mathbf{stop}(\mathsf{m}, i)$, then that state must be $\mathbf{v}'$ (that is, the end of a trajectory).

If no flow rate is specified for some variable $x[i] \in \mathsf{X}[i]$, then $x[i]$ is assumed to remain constant along the trajectory (that is, $\dot{x}[i] = 0$). If no invariant is specified, then it is assumed to be *true*, which specifies that the automaton *may* remain indefinitely in the corresponding location. If no stopping condition is specified, it is assumed to be *false*, which specifies that real time may elapse indefinitely in the corresponding location.

Executions, Invariants, and Inductive Invariants. An execution of the network $\mathcal{A}^{\mathcal{N}}$ models a particular behavior of all the automata in the network. An *execution* of $\mathcal{A}^{\mathcal{N}}$ is a sequence of states $\alpha = \mathbf{v}_0, \mathbf{v}_1, \ldots$ such that $\mathbf{v}_0 \in \Theta^{\mathcal{N}}$, and for each index k appearing in the sequence $(\mathbf{v}_k, \mathbf{v}_{k+1}) \in T^{\mathcal{N}}$. A state $\mathbf{x}$ is *reachable* if there is a finite execution ending with $\mathbf{x}$. The set of reachable states for $\mathcal{A}^{\mathcal{N}}$ is $\mathsf{Reach}(\mathcal{A}^{\mathcal{N}})$. The set of reachable states for $\mathcal{A}^{\mathcal{N}}$ starting from a subset $\mathbf{V}_0 \subseteq Q^{\mathcal{N}}$ is $\mathsf{Reach}(\mathcal{A}^{\mathcal{N}}, \mathbf{V}_0)$.

An invariant for $\mathcal{A}^{\mathcal{N}}$ is any set of states that contains $\mathsf{Reach}(\mathcal{A}^{\mathcal{N}})$. In general, any assertion over the variables of the automata in $\mathcal{A}^{\mathcal{N}}$ defines a subset of $Q^{\mathcal{N}}$. The dependence of such assertions on $\mathcal{N}$ is made explicit by using names like $\zeta(\mathcal{N})$. A network $\mathcal{A}^{\mathcal{N}}$ is safe with respect to an assertion $\zeta(\mathcal{N})$ if all its reachable states satisfy it, that is, $\mathsf{Reach}(\mathcal{A}^{\mathcal{N}}) \subseteq [\![\zeta(\mathcal{N})]\!]$. Given a template hybrid automaton $\mathcal{A}(\mathcal{N}, i)$ and a property $\zeta(\mathcal{N})$, we aim to prove for all $\mathcal{N} \in \mathbb{N}$, that every network is safe—that is, $\forall \mathcal{N} \in \mathbb{N}$, $\mathsf{Reach}(\mathcal{A}^{\mathcal{N}}) \subseteq [\![\zeta(\mathcal{N})]\!]$. To prove that $\mathcal{A}^{\mathcal{N}}$ is safe with respect to some unsafe set or property—that is, $\neg\zeta(\mathcal{N})$—it suffices to find an invariant $\Gamma(\mathcal{N}) \subseteq Q^{\mathcal{N}}$ such that $[\![\Gamma(\mathcal{N})]\!] \cap [\![\neg\zeta(\mathcal{N})]\!] = \emptyset$.

Several subclasses of hybrid automata have been identified for which safety verification by computing reachable sets is decidable—such as initialized rectangular hybrid automata (IRHA)[40,20] and order-minimal (o-minimal) hybrid automata[29]—and several automated model checking tools have been developed, such as HyTech,[19] PHAVer,[16] and SpaceEx.[17] However, the general model checking of even safety properties for hybrid automata is undecidable, so a standard approach is to use overapproximations of reachable states

[c] We have excluded continuous global variables to make the presentation clearer.

[d] This is an overapproximation of the set of solutions of the rectangular differential inclusion, as it excludes the requirement that the time derivative of any solution is in the differential inclusion $\dot{x}[i] \in [v_L, v_U]$.

for checking safety properties, such as the methods implemented in PHAVer for affine (linear) dynamics[16] and SpaceEx for affine dynamics as well[17] using the Le Geurnic-Girard (LGG) algorithm.[18] An alternative approach is to prove stronger inductive invariant assertions that imply a desired safety property, as originally used in Floyd-Hoare proofs[15,21] and the predicate transformers of Dijkstra.[11]

Definition 4. *An assertion $\Gamma(\mathcal{N})$ is an* inductive invariant *for the parameterized network $\mathcal{A}^{\mathcal{N}}$ if, for all $\mathcal{N} \in \mathbb{N}$, the following conditions hold:*

(A) ***initiation****: for each initial state $\mathbf{v} \in \Theta^{\mathcal{N}} \Rightarrow \mathbf{v} \models \Gamma(\mathcal{N})$,*

(B) ***transition consecution****: for each transition $(\mathbf{v}, \mathbf{v}') \in \mathcal{D}^{\mathcal{N}}$, if $\mathbf{v} \models \Gamma(\mathcal{N})$, then $\mathbf{v}' \models \Gamma(\mathcal{N})$, and*

(C) ***trajectory consecution****: for each trajectory $(\mathbf{v}, \mathbf{v}') \in \mathcal{T}^{\mathcal{N}}$, if $\mathbf{v} \models \Gamma(\mathcal{N})$, then $\mathbf{v}' \models \Gamma(\mathcal{N})$.*

Proving that a parameterized network satisfies a property is the *uniform verification* problem.

Definition 5. *The* uniform verification problem *is proving for any $\mathcal{N} \in \mathbb{N}$, for a property $\zeta(\mathcal{N})$ and parameterized network $\mathcal{A}^{\mathcal{N}}$, that $\mathcal{A}^{\mathcal{N}}$ satisfies $\zeta(\mathcal{N})$, written $\mathcal{A}^{\mathcal{N}} \models \zeta(\mathcal{N})$.*

The problem of uniform verification of parameterized networks is over arbitrary compositions of potentially infinite-state automata, so in general we may need to query a theorem prover to check conditions (A), (B), and (C). The next standard theorem states that if an assertion $\Gamma(\mathcal{N})$ is an inductive invariant—that is, $\Gamma(\mathcal{N})$ satisfies the conditions for inductive invariance (Definition 4)—then $\Gamma(\mathcal{N})$ is an invariant.[32,34,22]

Theorem 5.1. *If $\forall \mathcal{N} \in \mathbb{N}$, $\Gamma(\mathcal{N})$ is an inductive invariant, then it is also an invariant:*

$$\forall \mathcal{N} \in \mathbb{N}, \mathsf{Reach}(\mathcal{A}^{\mathcal{N}}) \subseteq [\![\Gamma(\mathcal{N})]\!] .$$

Theorem provers such as PVS have been augmented with support for verifying such networks.[9,44,4] The KeYmaera theorem prover also has support for verifying this type of networks.[37,31] These environments provide partially automatic means of proving inductive invariants of networks $\mathcal{A}^{\mathcal{N}}$. It is well known that the converse of Theorem 5.1 does not hold. If we can find an assertion $\Gamma(\mathcal{N})$ that is an inductive invariant and implies some desired safety property $\zeta(\mathcal{N})$, we say that $\Gamma(\mathcal{N})$ is sufficient to prove the safety property $\zeta(\mathcal{N})$.

Definition 6. *For any $\mathcal{N} \in \mathbb{N}$, an assertion $\Gamma(\mathcal{N})$ is sufficient to prove a safety property $\zeta(\mathcal{N})$ if $\Gamma(\mathcal{N})$ is an inductive invariant and $\Gamma(\mathcal{N}) \Rightarrow \zeta(\mathcal{N})$, so that $[\![\zeta(\mathcal{N})]\!] \subseteq [\![\Gamma(\mathcal{N})]\!]$.*

III. Synthesizing Inductive Invariants

In this section, we describe a method for finding candidate inductive invariants for parameterized networks of hybrid automata. The invariant synthesis method generates quantified inductive invariants by transforming the set of reachable states of finite instantiations of the network. This is an extension to hybrid automata of the invisible invariants method for synthesizing inductive invariants for parameterized networks of discrete automata.[5,39,6,7,36,12,33] These candidate inductive invariants are then checked for the conditions of Definition 4, and prove the safety property of interest if the inductive invariants imply the safety property.

For finding candidate inductive invariants for hybrid networks, our approach builds upon the invisible invariant method used for discrete transition systems.[5,39,7,36,33] The invisible invariants method combines the standard inductive invariance proof method—recall Definition 4—with reachability computations to automatically perform uniform verification of safety, that is, to prove a safety property $\zeta(\mathcal{N})$ for any network $\mathcal{A}^{\mathcal{N}}$ of any size $\mathcal{N}$ (Definition 5). The invisible invariant method starts by computing the set of reachable states for a small instantiation of the network. Say for $\mathsf{N} = 3$, the reach set (or its approximation) for the network $\mathcal{A}^3 \triangleq \mathcal{A}(1) \| \mathcal{A}(2) \| \mathcal{A}(3)$ is computed. Then this set is projected onto a smaller instance of size $\mathsf{P} < \mathsf{N}$. Finally, this projected subset is generalized to produce a candidate invariant for a network of arbitrary size. The user's choice of P determines the shape of the generated invariant. For $\mathsf{P} = 1$ the invariant asserts properties about the variables of a single automaton, for $\mathsf{P} = 2$ the properties may include linear inequalities involving pairs of automata, and so forth. In our methodology, the user may choose the projection to be made onto a subset of the variables of the automata in the P-sized network, such as only the real or discrete variables. This choice proves to be crucial in some of the case studies. If the generated candidate invariant is

```
1   function inductiveInvariance(𝒜(𝒩,i), ζ(𝒩), Init, N, P) {
      // synthesize candidate inductive invariants from finite instances
3     Γ(𝒩) ← pg(𝒜(N,i), Init, N, P)

5     // inductive invariance check for any 𝒩
      if (∀𝒩 ∈ ℕ Init(𝒩) ⇒ Γ(𝒩) is valid and // per Definition 4 (A)
7         ∀𝒩 ∈ ℕ transitionConsecution(𝒜(𝒩,i), 𝒩, Γ(𝒩)) is valid and // per Definition 4 (B)
          ∀𝒩 ∈ ℕ trajectoryConsecution(𝒜(𝒩,i), 𝒩, Γ(𝒩)) is valid and // per Definition 4 (C)
9         ∀𝒩 ∈ ℕ Γ(𝒩) ⇒ ζ(𝒩) is valid) { // per Definition 6
            return ζ(𝒩) is invariant for all 𝒩
11    }
      else {
13      return potential counterexample
      }
15  }
```

Figure 3. Inductive invariance proof method with inductive invariant synthesis. The inputs are an automaton specification $\mathcal{A}(\mathcal{N}, i)$, a desired safety property $\zeta(\mathcal{N})$, an initial condition assert Init, and two constants, N and P. The output is either a proof of the safety property $\zeta(\mathcal{N})$ for all $\mathcal{N} \in \mathbb{N}$, or a potential counterexample. The latter either indicates $\mathcal{A}(\mathcal{N}, i)$ has a bug and does not satisfy $\zeta(\mathcal{N})$ or that the synthesized invariants are not strong enough to prove $\zeta(\mathcal{N})$.

inductive and sufficient to prove some safety property $\zeta(\mathcal{N})$, then a completely automatic inductive invariance proof is obtained.

The project-generalize method is incomplete even for discrete systems.[39] The candidate invariants generated by our method may not be inductive nor are they guaranteed to prove $\zeta(\mathcal{N})$. However, we have found the heuristic to be practically useful for several examples, such as SATS. We implement the method in *Passel* to yield the first fully automatic proof of correctness for several nontrivial hybrid networks. Notable among these is the core of the SATS landing protocol.[2,1,35,44,45,25] The synthesis method finds non-trivial inductive invariants sufficient to prove safe separation for SATS. In the following, N and P are fixed natural numbers with $\mathsf{P} < \mathsf{N}$ and $\mathsf{N} \geq 2$ (e.g., $\mathsf{P} = 2$, $\mathsf{N} = 3$), and we recall $\mathcal{N}$ is a symbol denoting an arbitrary natural number.

A. Synthesizing Inductive Invariants with the Project-and-Generalize Method

This section describes methods for synthesizing inductive invariants for parameterized networks of hybrid automata. If a safety property $\zeta(\mathcal{N})$ itself is not an inductive invariant for $\mathcal{A}^{\mathcal{N}}$, as is often the case, then we attempt to find stronger inductive invariants that imply $\zeta(\mathcal{N})$.

The overall methodology of finding and checking inductive invariants is described by the pseudocode of Figure 3. Either a user must supply a sufficiently strong candidate inductive invariant to prove a safety property, or the verification tool may try to come up with one. The project-and-generalize method is shown in Figure 4. This method takes two input parameters N and P. The method first computes the reachable states of a network $\mathcal{A}^{\mathsf{N}}$ of size N, and then through a sequence of transformations generates a candidate inductive invariant with P universally quantified index variables for a network $\mathcal{A}^{\mathcal{N}}$ of arbitrary size $\mathcal{N}$.

Reachability Computation (line 4). The reach set $\mathsf{Reach}(\mathcal{A}^{\mathsf{N}})$ or its overapproximation is computed for the hybrid network $\mathcal{A}^{\mathsf{N}}$ with N automata. For general hybrid automata, computing the exact reach set is undecidable, however, there are several tools available for computing bounded-time overapproximations like HyTech,[19] PHAVer,[16] or SpaceEx.[17] This step can use any such tool. In the results presented in this paper (see Section IV), *Passel* uses PHAVer.[16] The output of this step is $\mathsf{Reach}(\mathcal{A}^{\mathsf{N}})$ as a disjunctive normal form (DNF) formula over the variables of $\mathcal{A}(1)$, ..., $\mathcal{A}(\mathsf{N})$.

Assumption 7. *For a given hybrid automaton template $\mathcal{A}(i)$ and natural number N, the reachability computation of the network $\mathcal{A}^{\mathsf{N}}$ (Definition 2) at line 4 terminates and is exact, yielding $\mathsf{Reach}(\mathcal{A}^{\mathsf{N}})$.*

Projection of $\mathsf{Reach}(\mathcal{A}^{\mathsf{N}})$ (loop lines 5 through 18). The loop iterates over each of the clauses $r \in \mathsf{Reach}(\mathcal{A}^{\mathsf{N}})$.[e] Given a clause r in $\mathsf{Reach}(\mathcal{A}^{\mathsf{N}})$, we project away the variables of any automata with indices greater than P. Recall that P specifies the number of universally quantified index variables in the invariant to be synthesized. *Passel* computes the projection using quantifier elimination procedures—represented by

[e] Since existential quantification distributes over disjunction, we consider each clause at a time.

```
1   function pg(𝒜(𝒩, i), Init, N, P) {
      V ← ∪_{i∈[N]\[P]} V_i
3     𝒜^N ← 𝒜_1‖𝒜_2‖...‖𝒜_N
      R ← Reach(𝒜^N, Init(N)) // assume in DNF: R = r_1 ∨ r_2 ∨ ...
5     foreach r in R {
        // project onto variables of processes 1,2,...,P
7       QF[r] ← QuantElim(∃V : r)
        // syntactically substitute 1,2,...,P to symbols i_1,...,i_P
9       foreach n in {1,2,...,P} {
          QF[r] ← Substitute(QF[r], "n", "i_n")
11      }
        // abstract index-valued variable valuations that are > P
13      foreach variable v in V_i with type(v) = [N]_⊥ {
          foreach n in {P+1,P+2,...,N} {
15          QF[r] ← Substitute(QF[r], "v = n", "v ≠ i_1 ∧ ... ∧ v ≠ i_P")
          }
17      }
      }
19    ψ(i_1,...,i_P) ← ⋁_{r∈R} QF[r]
      return ∀̇i_1,...,i_P ∈ [𝒩] : ψ(i_1,...,i_P)
21  }
```

Figure 4. Inductive invariant synthesis method. The input arguments are a hybrid automaton template $\mathcal{A}(\mathcal{N}, i)$, an initial condition assertion Init, a constant natural number N, and a constant natural number P, where $\mathsf{P} < \mathsf{N}$. The method computes the set of reachable from Init for a network of N automata, then transforms this reach set into an assertion $\psi(i_1, \ldots, i_\mathsf{P})$ over the variables of automata with (symbolic) indices i_1, i_2, $\ldots$, i_P. The output of the method is a universally quantified candidate inductive invariant $\psi(i_1, \ldots, i_\mathsf{P})$.

function $\mathsf{QuantElim}$ (line 7)—over the types of the variables V_i. These formulas (predicates over Booleans, linear real arithmetic, bounded integers, and their combinations) admit quantifier elimination. Based on the value of P, the quantifier elimination on line 7 is applied to $QF[r] \triangleq \exists \cup_{i\in[\mathsf{N}]\setminus[\mathsf{P}]} \mathsf{V}_i : r$, which projects away the variables of all automata with indices higher than P. In general, *Passel* projects away some subset of the variables V_i, for example, onto only the variables with discrete types or real types.

For example, in SATS, one of the clauses in $\mathsf{Reach}(\mathcal{A}^\mathsf{N})$ is:

$$r \triangleq (q[1] = \mathsf{base} \wedge q[2] = \mathsf{base} \wedge 3x[2] > 4x[1] + 56 \wedge \mathit{last} = 3 \wedge 28 \geq x[2] \geq 21). \quad (2)$$

For $\mathsf{P} = 1$, after executing $\mathsf{QuantElim}$, the variables of automaton 2 are eliminated to yield:[f]

$$QF[r] \triangleq \exists \mathsf{V}_2 : r \equiv \exists q[2] \in \mathsf{L}, \exists x[2] \in \mathbb{R} : r \equiv (q[1] = \mathsf{base} \wedge 7 \geq x[1] \geq 0 \wedge \mathit{last} = 3). \quad (3)$$

Generalization of Projected Clauses (lines 9 through 17). Next, the $\mathsf{Substitute}$ function syntactically substitutes expressions in $QF[r]$. The generalization syntactically replaces all valuations of index variables equal to a value in $[\mathsf{P}]$ with fresh index symbols $i_1, i_2, \ldots, i_\mathsf{P}$ (lines 9 through 11). Continuing with the r from the Equation 3 example, the index 1 is replaced with i_1 yielding:

$$QF[r] \triangleq (q[i_1] = \mathsf{base} \wedge 7 \geq x[i_1] \geq 0 \wedge \mathit{last} = 3). \quad (4)$$

The valuations of index-valued variables in $\mathsf{Reach}(\mathcal{A}^\mathsf{N})$ that exceed P are transformed to be not equal to any of the symbols i_1, $\ldots$, i_P (lines 13 through 17). In the example, $QF[r]$ has index 3 for valuations of the index-valued global variable *last* after projection and replacing 1 with i_1. We abstract such valuations by looking at each index-valued variable v, if the valuation $v = k$ where $k \leq \mathsf{P}$, then set $v = i_k$, and otherwise for $k > \mathsf{P}$ or $k = \perp$, set $v \neq i_1 \wedge \ldots \wedge v \neq i_\mathsf{P}$. Continuing the example r from Equation 4, we have:

$$QF[r] \triangleq (q[i_1] = \mathsf{base} \wedge 7 \geq x[i_1] \geq 0 \wedge \mathit{last} \neq i_1),$$

which contains symbolic indices i_1, $\ldots$, i_P, but no numerical indices.

Combining Clauses. Following these transformations of all r's, we take the disjunction of $QF[r]$ for all $r \in R$ (line 19). This is the formula $\psi(i_1, \ldots, i_\mathsf{P})$. A quantified formula is then created as (line 20):

$$\dot{\forall} i_1, i_2, \ldots, i_\mathsf{P} \in [\mathsf{N}] : \psi(i_1, i_2, \ldots, i_\mathsf{P}),$$

[f]We note that $q[2]$ and $x[2]$ are constants for the quantifier elimination and not functions mapping indices to their types, as otherwise this would fall into second-order logic.

where $\dot{\forall}$ indicates that all the quantified indices are distinct. However, for an arbitrary $\mathcal{N}$ like in the inductive invariance checks in Figure 3, *Passel* uses the quantifiers as in Figure 3, line 3.

SUMMARY. This section presents a method for finding candidate inductive invariants for parameterized networks of hybrid automata. The project-and-generalize method pg—inspired directly from the original works on finding invariants for discrete networks using the invisible invariants method[5,39]—computes the reachable states for small instantiations of the network, then transforms this set by projecting and generalizing it to a candidate invariant with a certain shape of quantification for the parameterized network. The function pg computes the reach set $\mathsf{Reach}(\mathcal{A}^{\mathsf{N}})$—a subset of the state-space Q^{N}—then projects this onto a smaller state-space Q^{P}, and then lifts this back to Q^{N}. Although our description above is in terms of the syntactic objects and logical formulas, these operations can be described in terms of mappings between the subsets of Q^{N} and Q^{P}. The method is necessarily incomplete, in that it fails to generate candidate invariants for networks even of a particular, restricted shape of quantification (for example, all universally quantified indices, $\forall i_1, i_2, \ldots$). We implement this method in the *Passel* verification tool. We experimentally evaluate the method in Section IV, where we show it to enable fully automatic safety verification for aerospace case studies such as SATS.

IV. Experimental Results Using the *Passel* Verification Tool

We have implemented the synthesis procedure described in Section III in a software tool called *Passel*, which relies on the Z3 satisfiability modulo theories (SMT) solver version 4.1[10] and PHAVer for reachability computations.[16] We previously implemented an automatic way to check the inductive invariance conditions (Definition 4) in *Passel*.[26] Thus, the extension to *Passel* we present in this paper is in automatically finding inductive invariants (recall Figure 3). The runtime for the entire procedure to verify safe separation (Equation 1) for SATS (Figure 2) was under 6.5 minutes, executed on a modern laptop running Ubuntu with an Intel Core i7 processor and 4GB RAM.

Passel synthesized 202 candidate invariants for SATS, and proved 119 of these candidates to satisfy the conditions of Definition 4. These inductive invariants were sufficient to establish collision of avoidance (Equation 1) for any choice of the number of interacting aircraft (i.e., $\forall \mathcal{N} \in \mathbb{N}$) automatically. The heuristic enabled a fully automatic verification of safe separation for SATS. Additionally, we have experimented with other formulations of SATS and have had success in proving safe separation for these as well.[g]

V. Conclusion

In this paper, we have described our preliminary results in automatically verifying parameterized cyber-physical aerospace systems, using a simplified model of SATS as a case study. Our method relies on synthesizing candidate assertions and then checking if these candidates are inductive invariants using our software tool *Passel*. The experimental results are promising, such as allowing for fully automatic verification of safe separation for a simplified model of SATS. There are several challenges to overcome to extend these results for systems with more complex continuous dynamics. For instance, to be able to model some flocking algorithms, we would need to allow linear dynamics,[24] or to be able to model conflict resolution maneuvers, we would likely use nonlinear dynamics.[43,42] As the autonomy of such safety-critical aerospace systems continues to increase, methods such as the ones developed in this paper will become more critical to analyze their complex behaviors. Alternatively, the class of properties under consideration could be expanded, such as to stability or liveness properties.[13]

Acknowledgments

This work was supported supported by the U.S. Air Force Office of Scientific Research (AFOSR) through the Young Investigator Research Program.

[g]The *Passel* tool and many case studies may be downloaded from: https://publish.illinois.edu/passel-tool/.

References

[1]T. S. Abbott, M. C. Consiglio, B. T. Baxley, D. M. Williams, K. M. Jones, and C. A. Adams. Small aircraft transportation system higher volume operations concept. Technical Report NASA/TP-2006-214512, L-19215, NASA, Oct. 2006.

[2]T. S. Abbott, K. M. Jones, M. C. Consiglio, D. M. Williams, and C. A. Adams. Small aircraft transportation system, higher volume operations concept: Normal operations. Technical Report NASA/TM-2004-213022, NASA, Aug. 2004.

[3]R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138(1):3–34, 1995.

[4]M. Archer, H. Lim, N. Lynch, S. Mitra, and S. Umeno. Specifying and proving properties of timed I/O automata using Tempo. *Design Automation for Embedded Systems*, 12:139–170, 2008.

[5]T. Arons, A. Pnueli, S. Ruah, Y. Xu, and L. Zuck. Parameterized verification with automatically computed inductive assertions? In G. Berry, H. Comon, and A. Finkel, editors, *Computer Aided Verification*, volume 2102 of *LNCS*, pages 221–234. Springer, 2001.

[6]I. Balaban, Y. Fang, A. Pnueli, and L. Zuck. IIV: An invisible invariant verifier. In *Computer Aided Verification*, volume 3576 of *LNCS*, pages 293–299. Springer, 2005.

[7]I. Balaban, A. Pnueli, and L. Zuck. Invisible safety of distributed protocols. In M. Bugliesi, B. Preneel, V. Sassone, and I. Wegener, editors, *Automata, Languages and Programming*, volume 4052 of *Lecture Notes in Computer Science*, pages 528–539. Springer Berlin / Heidelberg, 2006.

[8]A. Bayen, I. M. Mitchell, M. M. K. Oishi, and C. J. Tomlin. Aircraft autolander safety analysis through optimal control-based reach set computation. *Journal of Guidance, Control, and Dynamics*, 30(1), Jan. 2007.

[9]G. Chockler, N. Lynch, S. Mitra, and J. Tauber. Proving atomicity: An assertional approach. In P. Fraigniaud, editor, *Distributed Computing*, volume 3724 of *Lecture Notes in Computer Science*, pages 152–168. Springer Berlin Heidelberg, 2005.

[10]L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proc. of 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340. Springer-Verlag, 2008.

[11]E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, Aug. 1975.

[12]M. Emmi, R. Majumdar, and R. Manevich. Parameterized verification of transactional memories. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 134–145, New York, NY, USA, June 2010. ACM.

[13]Y. Fang, K. McMillan, A. Pnueli, and L. Zuck. Liveness by invisible invariants. In E. Najm, J.-F. Pradat-Peyre, and V. Donzeau-Gouge, editors, *Formal Techniques for Networked and Distributed Systems*, volume 4229 of *Lecture Notes in Computer Science*, pages 356–371. Springer, 2006.

[14]A. Filippov. Classical solutions of differential equations with multi-valued right-hand side. *SIAM Journal on Control*, 5(4):609–621, 1967.

[15]R. Floyd. Assigning meanings to programs. *Mathematical Aspects of Computer Science*, 19(19-32):1, 1967.

[16]G. Frehse. PHAVer: Algorithmic verification of hybrid systems past HyTech. *International Journal on Software Tools for Technology Transfer (STTT)*, 10:263–279, 2008.

[17]G. Frehse, C. Le Guernic, A. Donzé, S. Cotton, R. Ray, O. Lebeltel, R. Ripado, A. Girard, T. Dang, and O. Maler. SpaceEx: Scalable verification of hybrid systems. In *Computer Aided Verification (CAV)*, LNCS. Springer, 2011.

[18]C. L. Guernic and A. Girard. Reachability analysis of linear systems using support functions. *Nonlinear Analysis: Hybrid Systems*, 4(2):250–262, 2010.

[19]T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. HyTech: A model checker for hybrid systems. *Journal on Software Tools for Technology Transfer*, 1:110–122, 1997.

[20]T. A. Henzinger, P. W. Kopke, A. Puri, and P. Varaiya. What's decidable about hybrid automata? *Journal of Computer and System Sciences*, 57:94–124, 1998.

[21]C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, Oct. 1969.

[22]T. T. Johnson. *Uniform Verification of Safety for Parameterized Networks of Hybrid Automata*. PhD thesis, University of Illinois at Urbana-Champaign, Urbana, IL 61801, 2013.

[23]T. T. Johnson, J. Green, S. Mitra, R. Dudley, and R. S. Erwin. Satellite rendezvous and conjunction avoidance: Case studies in verification of nonlinear hybrid systems. In D. Giannakopoulou and D. Méry, editors, *Proceedings of the 18th International Conference on Formal Methods (FM 2012)*, volume 7436, pages 252–266. Springer Berlin Heidelberg, Paris, France, Aug. 2012.

[24]T. T. Johnson and S. Mitra. Safe flocking in spite of actuator faults. In S. Dolev, J. Cobb, M. Fischer, and M. Yung, editors, *12th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2010)*, volume 6366 of *Lecture Notes in Computer Science*, pages 588–602. Springer Berlin / Heidelberg, Sept. 2010.

[25]T. T. Johnson and S. Mitra. Parameterized verification of distributed cyber-physical systems: An aircraft landing protocol case study. In *ACM/IEEE 3rd International Conference on Cyber-Physical Systems*, Apr. 2012.

[26]T. T. Johnson and S. Mitra. A small model theorem for rectangular hybrid automata networks. In *Proceedings of the IFIP International Conference on Formal Techniques for Distributed Systems, Joint 14th Formal Methods for Open Object-Based Distributed Systems and 32nd Formal Techniques for Networked and Distributed Systems (FMOODS-FORTE)*, volume 7273 of *LNCS*. Springer, June 2012.

[27]T. T. Johnson, S. Mitra, and C. Langbort. Stability of digitally interconnected linear systems. In *Proceedings of the 50th IEEE Conference on Decision and Control and European Control Conference (CDC ECC 2011)*, pages 2687–2692, Orlando, Florida, USA, Dec. 2011.

[28]H. K. Khalil. *Nonlinear Systems*. Prentice Hall, Upper Saddle River, NJ, third edition, 2002.

[29]G. Lafferriere, G. Pappas, and S. Yovine. A new class of decidable hybrid systems. In F. Vaandrager and J. van Schuppen, editors, *Hybrid Systems: Computation and Control*, volume 1569 of *Lecture Notes in Computer Science*, pages 137–151. Springer Berlin / Heidelberg, 1999.

[30]C. Livadas, J. Lygeros, and N. A. Lynch. High-level modeling and analysis of TCAS. In *Proceedings of the 20th IEEE Real-Time Systems Symposium (RTSS'99)*, pages 115–125, Dec. 1999.

[31]S. M. Loos, A. Platzer, and L. Nistor. Adaptive cruise control: Hybrid, distributed, and now formally verified. In M. Butler and W. Schulte, editors, *Formal Methods*, LNCS. Springer, 2011.

[32]N. Lynch, R. Segala, and F. Vaandrager. Hybrid I/O automata. *Inf. Comput.*, 185(1):105–157, 2003.

[33]K. McMillan and L. Zuck. Invisible invariants and abstract interpretation. In E. Yahav, editor, *Static Analysis*, volume 6887 of *Lecture Notes in Computer Science*, pages 249–262. Springer Berlin / Heidelberg, 2011.

[34]S. Mitra. *A Verification Framework for Hybrid Systems.* PhD thesis, Massachusetts Institute of Technology, Cambridge, MA 02139, Sept. 2007.

[35]C. Muñoz, V. Carreño, and G. Dowek. Formal analysis of the operational concept for the small aircraft transportation system. In M. Butler, C. Jones, A. Romanovsky, and E. Troubitsyna, editors, *Rigorous Development of Complex Fault-Tolerant Systems*, volume 4157 of *LNCS*, pages 306–325. Springer Berlin / Heidelberg, 2006.

[36]K. Namjoshi. Symmetry and completeness in the analysis of parameterized systems. In B. Cook and A. Podelski, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 4349 of *Lecture Notes in Computer Science*, pages 299–313. Springer Berlin / Heidelberg, 2007.

[37]A. Platzer. Quantified differential dynamic logic for distributed hybrid systems. In *Computer Science Logic*, volume 6247 of *LNCS*, pages 469–483, 2010.

[38]A. Platzer and E. Clarke. Formal verification of curved flight collision avoidance maneuvers: A case study. In A. Cavalcanti and D. Dams, editors, *Formal Methods*, volume 5850 of *LNCS*, pages 547–562. Springer, 2009.

[39]A. Pnueli, S. Ruah, and L. Zuck. Automatic deductive verification with invisible invariants. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *LNCS*, pages 82–97. Springer, 2001.

[40]A. Puri and P. Varaiya. Decidability of hybrid systems with rectangular differential inclusions. In D. Dill, editor, *Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 95–104. Springer Berlin / Heidelberg, 1994.

[41]J. Rushby. Formal methods and the certification of critical systems. Technical Report CSL-93-7, Computer Science Laboratory, SRI International, Menlo Park, CA 94025, USA, Dec. 1993.

[42]C. Tomlin, I. Mitchell, and R. Ghosh. Safety verification of conflict resolution maneuvers. *Intelligent Transportation Systems, IEEE Transactions on*, 2(2):110–120, June 2001.

[43]C. Tomlin, G. Pappas, and S. Sastry. Conflict resolution for air traffic management: A study in multiagent hybrid systems. *IEEE Trans. Autom. Control*, 43(4):509–521, Apr. 1998.

[44]S. Umeno and N. Lynch. Proving safety properties of an aircraft landing protocol using I/O automata and the PVS theorem prover: A case study. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *Formal Methods*, volume 4085 of *LNCS*, pages 64–80. Springer, 2006.

[45]S. Umeno and N. Lynch. Safety verification of an aircraft landing protocol: A refinement approach. In *Hybrid Systems: Computation and Control*, volume 4416 of *LNCS*, pages 557–572. Springer, 2007.