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
$\mathcal{N}$ Fixed number of participants
$[\mathcal{N}]$ Set of arbitrary participant identifiers
$\mathbb{N}$ Set of fixed, finite number of participant identifiers
$A(\mathcal{N},i)$ Hybrid automaton template specifying participant $i \in [\mathcal{N}]$
$A(i)$ Shorthand for $A(\mathcal{N},i)$ when $\mathcal{N}$ is clear from context
$A^\mathcal{N}$ 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 theorems 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 \(A(i)\) may read the variables of another hybrid automaton \(A(j)\) by maintaining a pointer to \(A(j)\). 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 \(\dot{x}[i] \in [v_U, v_U]\) for \(0 < v_U \leq v_U\). 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:

\[
\forall i, j \in \mathcal{N}. (i \neq j \land q[i] = \text{base} \land q[j] = \text{base} \land x[i] > x[j]) \Rightarrow (x[i] - x[j] \geq L_S).
\]  

(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 \(\mathcal{N} = 3\), 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.
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 generalizes and unifies the modeling frameworks developed in our prior work. 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., \( N = 3 \)). 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 \([n] \doteq \{1, \ldots, n\}\), and we use the sets \([N]\) and \([\mathcal{N}]\) for indexing automata. For a set \( S \), we define \( S_{\bot} \doteq S \cup \{\bot\} \).

For any natural number \( \mathcal{N} \) and \( i \in [\mathcal{N}] \), an individual hybrid automaton \( \mathcal{A}(\mathcal{N}, i) \) is a (possibly non-deterministic) 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 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 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.
A variable is a name used for referring to state. A variable $v$ is associated with a type—denoted $\text{type}(v)$—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) $[N]_\perp$: the set of automaton indices (identifiers) with the special element $\perp$ that is not equal to any index. A variable of this type is called a pointer variable or a pointer in short.
(c) $R$: the set of real numbers.

A variable may be local with a name of the form $\text{variable\_name}[i]$, or global, in which case the name does not have the index $[i]$. For example, $q[i] : L$, $p[i] : [N]_\perp$, and $x[i] : R$ respectively define location, pointer, real, and typed local variables, while $q : [N]_\perp$ is a global variable of index type. For a local variable $q[i]$, the array of variables $(q[1], q[2], \ldots, q[N])$ is denoted by $q^N$. We write $\overline{q}$ when $N$ 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 $A(N, i)$. 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:

$$\begin{align*}
\text{ITerm} &::= \perp \mid 1 \mid N \mid i \mid p[i] \\
\text{DTerm} &::= l_c \mid q \mid q[l\text{Term}] \\
\text{RTerm} &::= 0 \mid 1 \mid r_c \mid x \mid x[l\text{Term}]
\end{align*}$$

An index term (ITerm) is either (a) one of the constants $\perp, 1, N$, 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 ITerm by restricting ITerm from being produced by $p[l\text{Term}]$—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 $q[l\text{Term}]$ 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 $x[l\text{Term}]$ is a real array referenced at an index specified by an ITerm.

Real polynomials and constraints are built using the following grammar.

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

$\text{RPoly}_1$ and $\text{RPoly}_2$ are shorter real polynomials joined by arithmetic operators—addition $+$, subtraction $-$, or multiplication $\ast$—to obtain longer polynomials. $\text{RAtom}$ are used for specifying real constraints. Other comparison operators—less than or equal ($\leq$), greater than or equal ($\geq$), greater than ($>$), and equality ($=$)—will be expressed using negation ($\neg$) and conjunction ($\land$) in the formulas we define next.

For a polynomial $p$ generated by $\text{RPoly}$ over $n$ real variables $x_1, \ldots, x_n$ with $k$ additive terms, $p = a_1 x_1^{e_1} \ast \ldots \ast x_n^{e_n} + \ldots + a_k x_1^{e_1-k} \ast \ldots \ast x_n^{e_n-k}$, 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., $\ast$ before $+$, etc.).

Using these terms and constraints, formulas are defined next:

$$\begin{align*}
\text{Atom} &::= \text{ITerm}_1 \lt \text{ITerm}_2 \mid \text{DTerm}_1 = \text{DTerm}_2 \mid \text{RAtom} \\
\text{Formula} &::= \text{Atom} \mid \text{\neg Formula} \mid \text{Formula}_1 \land \text{Formula}_2 \mid \exists x \text{ Formula}
\end{align*}$$

Here, $x$ is called a bound variable, and is a variable of one of the types. $\text{Formula}_1$ and $\text{Formula}_2$ are shorter formulas that are joined by Boolean operators to obtain a longer formula. By combining the Boolean operators $\land$ and $\neg$ with the $\lt$ 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 $\neg(p_1[i] < p_2[j]) \land (p_2[j] < p_1[i])$. Universally quantified variables can be expressed by $\exists x \colon \text{Formula} \equiv \forall x : \neg \text{Formula}$. 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 \( \lor \), 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 \( \phi \) is in conjunctive normal form (CNF) if and only if it is a conjunction of conjunctive clauses, where a conjunctive clause is one or more conjunctions of one or more atoms. A formula \( \phi \) is in disjunctive normal form (DNF) 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 \( \phi \), let \( \text{vars}(\phi) \) be the set of variables appearing in \( \phi \). For a formula \( \phi \), let \( \text{ivars}(\phi) \) be the set of distinct index variables appearing in \( \phi \). For a formula \( \phi \), let \( \text{free}(\phi) \) be the set of free variables appearing in \( \phi \). For a formula \( \phi \), let \( \text{bound}(\phi) \) be the set of bound variables appearing in \( \phi \). For a quantified formula \( \phi \), let \( \text{body}(\phi) \) be the body of the quantifier, with all bound variables \( \text{bound}(\phi) \) replaced with universally instantiated variables with the same names. For a set of variable names \( V \) and a formula \( \phi \), if \( \text{free}(\phi) \subseteq V \), then \( \phi \) is over \( V \). For a set of variable names \( V \) and a formula \( \phi \), if \( \text{free}(\phi) = V \), then \( \phi \) 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 \( A(N, i) \).

**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 \( N \in \mathbb{N} \) and \( i \in [N] \), a hybrid automaton template \( A(N, i) \) 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: an initial condition, which is a Passel assertion over \( V_i \),

(d) Trans: 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 \( V_i \cup V'_i \), where \( V'_i \) is the set of primed variable names corresponding to \( V_i \), and

(e) Flow: 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 \( V_i \cup V_{dot} \), where \( V_{dot} \) 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 \( A(N, i) \) is written \( A(i) \) when \( N \) 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 \( A(i) \). 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 \{fly, hold, base, runway\} (lines 12, 14, 16, and 20). Locations are depicted graphically as the circles in Figure 1. Each automaton \( A(i) \) 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.
implies (x[i] >= x[j] + L_S )

Initial Conditions. The initial condition assertion following the keyword initially (lines 1, 2, 3, 4, and 5).

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 8 through 10. It has two local variables, $q[i]$ and $x[i]$, with types $\mathbb{L}$ and $\mathbb{R}$, and a single global variable last of type $\mathbb{N}$.

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

initially: for all $i$ (q[i] = fly and next[i] = ⊥ and last = ⊥)

property: for all $i$, $j$ ($i$ != $j$ and q[i] = base and q[j] = base and $x[i] > x[j]$)

implies (x[i] >= x[j] + L_B)

Figure 2. Passel input file specifying hybrid automaton $\mathcal{A}(i)$ for SSATS, a simplified SATS protocol.
where $i$ is implicitly quantified over $[N]$. The initial condition assertion for SATS asserts that, for each index $i \in [N]$, the variables of $A(i)$ have the constraints $q[i] = \text{fly}$ and $x[i] = 0$, and that the global variable $last = \perp$. If a variable $v \in V_i$ is not specified in the initial condition, it is assumed that $v$ is initially an arbitrary value in its type $\text{type}(v)$. Note that the $\text{Init}_i$ assertion may specify constraints over all automata in the network using universally index-quantified $\text{Passel}$ assertions.

**Candidate Invariant Properties.** Candidate invariant properties are specified as $\text{Passel}$ 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 $N \in \mathbb{N}$ and any $i \in [N]$, the set of discrete transitions $\text{Trans}_i$ 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 $\text{src}$ to location $\text{dest}$ by $t(\text{src}, \text{dest}) \in \text{Trans}_i$, which is written as $t$ when the from-to locations are clear from context.

Each transition $t \in \text{Trans}_i$ may specify a guard following the keyword $\text{grd}$, a universal guard following the keyword $\text{ugrd}$, and an effect following the keyword $\text{eff}$. The guard, universal guard, and effect are quantifier-free $\text{Passel}$ assertions, and they are denoted by $\text{grd}(t, i)$, $\text{ugrd}(t, i)$, and $\text{eff}(t, i)$ for $A(i)$, respectively. If $i$ is clear from context, we drop it and write $\text{grd}(t)$, $\text{ugrd}(t)$, or $\text{eff}(t)$. The universal guard is a quantifier-free $\text{Passel}$ assertion involving the variables $V_j$, for $j \neq i$, and we recall $i$ is the index of the template $A(N, i)$. 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 $\text{Passel}$ assertion over the variables $V_i \cup V'_i$, where $V'_i$ concatenates a prime ($t$) 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 $\text{hold}$ to $\text{base}$ for SATS with index $i$ (line 26), where

$$\begin{align*}
\text{grd} : & \quad last = i, \\
\text{eff} : & \quad x[i]' = 0.0
\end{align*}$$

specifies that automaton $i$ may nondeterministically transition from a state where $q[i] = \text{hold}$ to a state where $q[i]' = \text{base}$, 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 $q[i] = \text{fly}$.

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

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

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

$$\begin{align*}
\text{inv} : & \quad x[i] \leq L_B \\
\text{stop} : & \quad x[i] = L_B \\
\text{flowrate} : & \quad x[i]_{\text{dot}} \geq v_L \land x[i]_{\text{dot}} \leq v_U
\end{align*}$$

In addition, the invariant requires that the automaton with index $i$ can have $q[i] = \text{base}$ 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 \((v_L, v_U)\), respectively.

B. Semantics of Hybrid Automata Networks

The semantics of the hybrid automata network \(\mathcal{A}^N\)—where an arbitrary number \(N \in \mathbb{N}\) of instances of the template \(\mathcal{A}(N, i)\) operate in parallel—is defined in this section. The semantics are defined in terms of a transition system with a set of variables \(V^N\), a set of states \(Q^N\), a set of initial states \(\Theta^N\), and a transition relation \(T^N\).

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 \(N \in \mathbb{N}\), a parameterized network of hybrid automata is a tuple \(\mathcal{A}^N \triangleq \langle V^N, Q^N, \Theta^N, T^N \rangle\), where

(a) \(V^N\) are the variables of the network, \(V^N \triangleq V_G \cup \bigcup_{i=1}^{N} V_L[i]\).

(b) \(Q^N \subseteq \text{val}(V^N)\) is the state-space,

(c) \(\Theta^N \subseteq Q^N\) is the set of initial states, and

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

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

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

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

**Representing States with Assertions.** Subsets of \(Q^N\) are often represented by assertions involving the variables. If a state \(v\) satisfies a formula \(\phi\)—that is, the corresponding variable valuations result in \(\phi\) evaluating to true—we write \(v \models \phi\). For such a formula \(\phi\), the corresponding states satisfying \(\phi\) 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 \(\psi\) is denoted \(M(n, \psi)\) and provides an interpretation of each the free variables in \(\psi\) as follows:

- the index constants \(\bot, 1, \text{and } N\) 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 $\psi$ and a model $M(n, \psi)$, if $\psi$ evaluates to true with the interpretations of the free variables given by $M(n, \psi)$, then $M(n, \psi)$ is said to satisfy $\psi$. If there exist models that satisfy $\psi$, then the assertion is said to be satisfiable. If all models of $\psi$ satisfy it, then the assertion is said to be valid.

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

$$\Theta^N \triangleq \{ x \in Q^N \mid \forall i \in [N], x.q[i] = \texttt{fly} \land x.x[i] = 0.0 \land \texttt{last} = \bot \}.$$

**Transitions and Trajectories.** The evolution of the states of $\mathcal{A}^N$ are described by a transition relation $T^N \subseteq Q^N \times Q^N$. For a pair $(\mathbf{v}, \mathbf{v}') \in T^N$, 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 $T^N$: discrete transitions $D^N$ model instantaneous change of state and continuous trajectories $T^N$ 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 $\mathcal{A}^N$. There is a discrete transition $\mathbf{v} \rightarrow \mathbf{v}' \in D^N$ iff:

$$\exists i \in [N] \exists t \in \texttt{Trans}_i : \mathbf{v}.V_i \models \texttt{grd}(t, i) \land \mathbf{v}'.V_i \models \texttt{eff}(t, i) \land \left( \forall j \in [N] : \mathbf{v}.V_j \models \texttt{ugrd}(t, j) \land j \neq i \Rightarrow \mathbf{v}'.V_j = \mathbf{v}.V_j \right).$$

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 $t \in \texttt{Trans}_i$. 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 $t(\texttt{src}, \texttt{dest})$ may only be taken by automaton $\mathcal{A}(i)$ if $q[i] = \texttt{src}$. We assume the identity relation for any primed variable $v' \in V'_i$ 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 $t(\texttt{base}, \texttt{runway})$ (line 39) for some automaton $\mathcal{A}(i)$ are defined by:

$$\left( \mathbf{v}, \mathbf{v}' \right) \models \exists i \in [N] : (q[i] = \texttt{base} \land \texttt{last} = i \land x[i] \geq L_B) \land (q[i]' = \texttt{runway} \land x[i]' = 0.0 \land \texttt{last}' = \bot) \land (\forall j \in [N] : j \neq i \Rightarrow q[j]' = q[j] \land x[j]' = x[j]).$$

This transition can occur from states $\mathbf{v}$ where $\mathbf{v} \models q[i] = \texttt{base} \land x[i] \geq L_B \land \texttt{last} = i$. This transition updates the location of $\mathcal{A}(i)$ to be $q[i]' = \texttt{runway}$, 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 $\mathbf{v} \rightarrow \mathbf{v}' \in T^N$ iff some amount of time---$t_e$---can elapse from $\mathbf{v}$, such that, (a) the states of all automata in the network $\mathcal{A}^N$ 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 $\gamma : [0, t] \rightarrow \mathbb{R}^n$ such that $\gamma(0) = x_0$ and, for every $\tau \in [0, t]$, $\gamma(\tau) = f(\gamma(\tau))$. 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 $\gamma : [0, t] \rightarrow \mathbb{R}^n$ such that $\gamma(0) = x_0$ and, for every $\tau \in [0, t]$, $\gamma(\tau) \in F(\gamma(\tau))$. Sufficiently smooth differential equations satisfying continuity conditions---such as Lipschitz continuity---have unique solutions, whereas differential inclusions have families of solutions.
Thus, to define trajectories for $\mathcal{A}^N$ formally, we first define a set-valued function called $\text{flow}(m, vV_L[i], t)$ that returns the states of $\mathcal{A}(i)$ when $q[i] = m$ that can be reached from $vV_L[i]$ in $t$ time. We suppose $\text{flow}(m, vV_L[i])$ is set-valued, as this subsumes the case when $\text{flowrate}(m, x[i])$ specifies a differential equation for $\dot{x}[i]$ instead of an inclusion.

Let $n = |X[i]|$ be the number of continuous local variables in the template $\mathcal{A}(i)$. Let $\text{flowrate}(m, X[i]) : \mathbb{R}^n \rightarrow \mathbb{R}^n$ be the vector of differential inclusions (and/or equations) for all the continuous variables $X[i]$ of $\mathcal{A}(i)$, assumed to be ordered lexicographically by the variable names. For example, for SATS in base, $\text{flowrate}(\text{base}, X[i])$ is $v_L \leq \dot{x}[i] \leq v_U$ (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, $v'.V_L[i] = \text{flow}(m, vV_L[i], t)$ iff:

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

(b) for each non-real local variable $y[i] \in V_L[i] \setminus X[i]$, $v'.y[i] = v.y[i]$.

For the SATS example for $\text{base}$ (Figure 2, line 16),

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

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 $v \rightarrow v' \in \mathcal{T}^N$ iff:

$$\exists t_e \in \mathbb{R}_{\geq 0} \forall i \in [N] \exists m \in L \forall t_p \leq t_e : \text{flow}(m, vX[i], t_p) \models \text{inv}(m, i) \land$$

$$\text{flow}(m, vX[i], t_p) \models \text{stop}(m, i) \Rightarrow t_p = t_e \land$$

$$v'.X[i] \in \text{flow}(m, vX[i], t_e).$$

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

If no flow rate is specified for some variable $x[i] \in X[i]$, 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 $\mathcal{A}^N$ models a particular behavior of all the automata in the network. An execution of $\mathcal{A}^N$ is a sequence of states $\alpha = v_0, v_1, \ldots$ such that $v_0 \in \Theta^N$, and for each index $k$ appearing in the sequence ($v_k, v_{k+1}$) $\in T^N$. A state $x$ is reachable if there is a finite execution ending with $x$. The set of reachable states for $\mathcal{A}^N$ is $\text{Reach}(\mathcal{A}^N)$. The set of reachable states for $\mathcal{A}^N$ starting from a subset $V_0 \subseteq Q^N$ is $\text{Reach}(\mathcal{A}^N, V_0)$.

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

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) and order-minimal (o-minimal) hybrid automata—and several automated model checking tools have been developed, such as HyTech, PHAVer, and SpaceEx. 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.

\footnote{We have excluded continuous global variables to make the presentation clearer.}

\footnote{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\textsuperscript{16} and SpaceEx for affine dynamics as well\textsuperscript{17} using the Le Geurnic-Girard (LGG) algorithm.\textsuperscript{18} An alternative approach is to prove stronger inductive invariant assertions that imply a desired safety property, as originally used in Floyd-Hoare proofs\textsuperscript{15,21} and the predicate transformers of Dijkstra.\textsuperscript{11}

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

- (A) **initiation:** for each initial state $v \in \Theta^N \Rightarrow v = \Gamma(N)$,
- (B) **transition consecution:** for each transition $(v, v') \in D^N$, if $v = \Gamma(N)$, then $v' = \Gamma(N)$, and
- (C) **trajectory consecution:** for each trajectory $(v, v') \in T^N$, if $v = \Gamma(N)$, then $v' = \Gamma(N)$.

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

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

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 $\Gamma(N)$ is an inductive invariant—then $\Gamma(N)$ satisfies the conditions for inductive invariance (Definition 4)—then $\Gamma(N)$ is an invariant.\textsuperscript{32,34,22}

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

$$\forall N \in \mathbb{N}, \text{Reach}(A^N) \subseteq \Theta(\Gamma(N)).$$

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

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

### 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.\textsuperscript{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.\textsuperscript{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 $\zeta(N)$ for any network $A^N$ of any size $N$ (Definition 5). The invisible invariant method starts by computing the set of reachable states for a small instantiation of the network. Say for $N = 3$, the reach set (or its approximation) for the network $A^N \triangleq A(1)\|A(2)\|A(3)$ is computed. Then this set is projected onto a smaller instance of size $P < N$. 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 $P = 1$ the invariant asserts properties about the variables of a single automaton, for $P = 2$ 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
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 \( \zeta(N) \) itself is not an inductive invariant for \( \mathcal{A}^N \), as is often the case, then we attempt to find stronger inductive invariants that imply \( \zeta(N) \).

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 \( \mathcal{A}^N \) of size \( N \), and then through a sequence of transformations generates a candidate inductive invariant with \( P \) universally quantified index variables for a network \( \mathcal{A}^N \) of arbitrary size \( N \).

Reachability Computation (Line 4). The reach set \( \text{Reach}(\mathcal{A}^N) \) or its overapproximation is computed for the hybrid network \( \mathcal{A}^N \) 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, \(^{19}\text{PHAVer,}^{16}\) 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 \( \text{Reach}(\mathcal{A}^N) \) as a disjunctive normal form (DNF) formula over the variables of \( \mathcal{A}(1), \ldots, \mathcal{A}(N) \).

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

Projection of \( \text{Reach}(\mathcal{A}^N) \) (Loop Lines 5 through 18). The loop iterates over each of the clauses \( r \in \text{Reach}(\mathcal{A}^N) \).\(^{e}\) Given a clause \( r \) in \( \text{Reach}(\mathcal{A}^N) \), 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.
function $pg(A(N,i),\text{Init},N,P)$ 

V ← $\sqcup_{i\in[N]\setminus P}$V_i 

$A_\text{init} \leftarrow A_1 \parallel A_2 \parallel \ldots \parallel A_N$ 

$R \leftarrow \text{Reach}(A_\text{init}(\text{Init}))$ // assume in DNF: $R = r_1 \lor r_2 \lor \ldots$ 

foreach $r$ in $R$ 

// project onto variables of processes $1,2,\ldots,P$ 

$QF[r] \leftarrow \text{QuantElim}(\exists V : r)$ 

// syntactically substitute $1,2,\ldots,P$ to symbols $i_1,\ldots,i_P$ 

foreach $n$ in $(1,2,\ldots,P)$ 

$QF[r] \leftarrow \text{Substitute}(QF[r], "n", "i_n")$ 

} 

// abstract index-valued variable valuations that are $> P$ 

foreach variable $v$ in $V_i$ with type $(v) = [N]_{\bot}$ 

foreach $n$ in $(P+1,P+2,\ldots,N)$ 

$QF[r] \leftarrow \text{Substitute}(QF[r], "v = n", "v \neq i_1 \land \ldots \land v \neq i_P")$ 

} 

$\psi(i_1,\ldots,i_P) \leftarrow V_r \in R QF[r]$ 

return $\forall[i_1,\ldots,i_P] 

\psi(i_1,\ldots,i_P)$ 

Figure 4. Inductive invariant synthesis method. The input arguments are a hybrid automaton template $A(N,i)$, an initial condition assertion Init, a constant natural number $N$, and a constant natural number $P$, where $P < N$. The method computes the set of reachable from Init for a network of $N$ automata, then transforms this reach set into an assertion $\psi(i_1,\ldots,i_P)$ 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 $\psi(i_1,\ldots,i_P)$.

The values of index-valued variables in Reach($A_\text{init}$) 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 values of the index-valued variable last after projection and replacing 1 with $i_1$. We abstract such values by looking at each index-valued variable $v$, if the valuation $v = k$ where $k \leq P$, then set $v = i_k$, and otherwise for $k > P$ or $k = \bot$, set $v \neq i_1 \land \ldots \land v \neq i_P$. Continuing the example $r$ from Equation 4, we have:

\[ QF[r] \triangleq (q[i_1] = \text{base} \land 7 \geq x[i_1] \geq 0 \land last \neq i_1), \]

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 $\psi(i_1,\ldots,i_P)$. A quantified formula is then created as (line 20):

\[ \forall[i_1,\ldots,i_P] \in [N] : \psi(i_1,\ldots,i_P), \]

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 $\forall$ indicates that all the quantified indices are distinct. However, for an arbitrary $N$ 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 $\text{pg}$—inspired directly from the original works on finding invariants for discrete networks using the invisible invariants method—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 $\text{pg}$ computes the reach set $\text{Reach}(A^N)$—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 4.110 and PHAVer for reachability computations. We previously implemented an automatic way to check the inductive invariance conditions (Definition 4) in Passel. 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., $\forall N \in N$) 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.

**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, or to be able to model conflict resolution maneuvers, we would likely use nonlinear dynamics. 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.

**Acknowledgments**

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

---

6The Passel tool and many case studies may be downloaded from: [https://publish.illinois.edu/passel-tool/](https://publish.illinois.edu/passel-tool/).
References


