

Anonymized Reachability of Rectangular Hybrid Automata Networks

Taylor T. Johnson¹ and Sayan Mitra²

¹ University of Texas at Arlington, Arlington, TX 76019, USA
taylor.johnson@uta.edu,

² University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA
mitras@illinois.edu,

Abstract. In this paper, we present a method for computing the set of reachable states for networks consisting of the parallel composition of a finite number of the same hybrid automaton template with rectangular dynamics. The method utilizes a symmetric representation of the set of reachable states (modulo the automata indices) that we call anonymized states, which makes it scalable. Rather than explicitly enumerating all the automaton indices in formulas representing sets of states, the anonymized representation encodes only: (a) the classes of automata, which are the states of automata represented with formulas over symbolic indices, and (b) the number of automata in each of the classes. We present an algorithm for overapproximating the reachable states by computing state transitions in this anonymized representation. Unlike symmetry reduction techniques used in finite state models, the timed transition of a network composed of hybrid automata causes the continuous variables of all the automata to evolve simultaneously. The anonymized representation is amenable to both reducing the discrete and continuous complexity. We evaluate a prototype implementation of the representation and reachability algorithm in our SMT-based tool, Passel. Our experimental results are promising, and generally allow for scaling to networks composed of tens of automata, and in some instances, hundreds of automata.

Keywords: hybrid automata network, reachability, verification, symmetry

1 Introduction

Networks consisting of automata that communicate via shared variables are useful for modeling distributed algorithms such as mutual exclusion algorithms, media access control (MAC) such as time-division multiple access (TDMA) protocols, and distributed cyber-physical systems (CPS) such as air-traffic control systems. However, as the discrete state-space of the network consisting of parallel compositions of these automata grows exponentially in the number of automata, automated analysis is challenging, and is particularly challenging for timed and hybrid systems, where the number of continuous variables (dimensions) also grows. Such networks are often specified in a symmetric manner—such as being composed of instantiations of an automaton template—and are often amenable to methods that exploit symmetries. Formal analysis and state-space construction methods that exploit symmetries have been thoroughly

investigated for many classes of system models, because such methods ameliorate the state-space explosion problem [1–13]. Several methods exploiting symmetry have been developed and implemented for the Mur φ verification system [14] for discrete systems. The *scalarset* data structure, which is a finite unordered set, is developed and added to Mur φ in [2], and was one of the first approaches of automatically detecting and exploiting symmetries in model checking. The *repetitive id* data structure is applied to several discrete parameterized systems like cache coherence protocols in [4].

Advances in tools like UPAAAL [15] and PAT [16] that exploit state-space symmetries to reduce its size vastly have enabled scaling to larger models. For instance, the scalar set technique developed for Mur φ was extended for timed systems and implemented in UPAAAL [7, 17], and a clock-symmetry reduction method [13] has been implemented in the PAT model checker. Quasi-equal clocks and variables for timed [18] and hybrid [19] automata networks also allow reductions in state-space explosion, but do not require automata in the network to be identical (modulo identifiers), as we do. We focus on safety properties, and to the best of our knowledge, before this paper, such techniques have not yet been applied to systems with more general continuous dynamics like the rectangular differential inclusions we consider. The method described in this paper and implemented in our *Passel* verification tool [20–22] uses the SMT solver Z3 [23]. The method is used as a subroutine in methods for performing uniform verification of parameterized networks of hybrid automata (e.g., verification for all network sizes, $\forall N \in \mathbb{N}, \mathcal{A}_1 \parallel \dots \parallel \mathcal{A}_N \models \zeta(N)$), although we highlight that this paper address fixed, constant choices of N only.

2 Hybrid Automata Network Syntax and Semantics

We specify the behavior of each participant in the network using a syntactic structure called a hybrid automaton template, denoted by $\mathcal{A}(N, i)$.³ The special symbols N and i are natural numbers that respectively refer to the number of automata, and the i^{th} automaton. For a natural number n , the set $[n]$ is $\{1, \dots, n\}$. For a set S , the set S_\perp is $S \cup \{\perp\}$. Fixing a particular value of N gives concrete instances of $[N]$ and $[N]_\perp$.

Terms and Formulas. We use a class of formulas to: (a) specify the syntactic components of a hybrid automaton template $\mathcal{A}(N, i)$, and (b) represent sets of states symbolically in the reachability computation. Formulas are built-up from constants, variables, and terms of several types. The grammar for formulas is:

$$\begin{aligned} \text{ITerm} &::= \perp \mid 1 \mid N \mid i \mid p[i] \\ \text{DTerm} &::= l_c \mid q \mid q[\text{ITerm}] \\ \text{RTerm} &::= 0 \mid 1 \mid r_c \mid x \mid x[\text{ITerm}] \\ \text{RPoly} &::= \text{RTerm} \mid \text{RPoly}_1 + \text{RPoly}_2 \mid \text{RPoly}_1 - \text{RPoly}_2 \mid (\text{RPoly}_1 * \text{RPoly}_2) \\ \text{Atom} &::= \text{ITerm}_1 = \text{ITerm}_2 \mid \text{DTerm}_1 = \text{DTerm}_2 \mid \text{RPoly} < 0 \\ \text{Formula} &::= \text{Atom} \mid \neg \text{Formula} \mid \text{Formula}_1 \wedge \text{Formula}_2 \mid \exists x \text{ Formula} \end{aligned}$$

³ Readers interested in additional technical details are referred to [22, Chapters 2 and 4].

The grammar is composed of *index terms* (ITerm) with type $[N]_{\perp}$, *discrete terms* (DTerm) with type L , and *real terms* (RTerm) with type \mathbb{R} . For a discrete term, l_c is constant from L and q is a discrete variable. For a real term, r_c is a real numerical constant and x is a real variable. Index ($p[i]$), discrete ($q[\text{ITerm}]$), and real ($x[\text{ITerm}]$) *pointer variables* are names for arrays composed of N elements of the corresponding type, respectively referenced at an index variable i , or an evaluation of an index term ITerm. Atoms (Atom) are composed of ordered relations between real polynomials (RPoly), as well as equality relations between index terms and discrete terms. Formulas are composed of Boolean combinations of atoms and shorter formulas. Comparison operators are expressed using negation (\neg) and conjunction (\wedge) in formulas. By combining the Boolean operators \wedge and \neg with the $<$ operator, other comparison operators like $=$, \neq , \leq , $>$, and \geq , can be expressed in formulas for indices and reals. Universal quantification can be expressed by $\neg\exists x : \text{Formula} \equiv \forall x : \neg\text{Formula}$, where x is called a *bound variable*, and is a variable of one of the types. We assume the language contains the standard quantifiers and Boolean operators, even if not explicitly specified in the grammar (e.g., universal quantification \forall , implication \Rightarrow , disjunction \vee , less-than-or-equal \leq , etc.).

Variables. A hybrid automaton $\mathcal{A}(N, i)$ has a set of *variables*, each of which is a name used for referring to state and is a term in the grammar just defined. As specified in the grammar, each 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 of locations names, (b) $[N]_{\perp}$: a set of automaton indices—called pointers—with the special element \perp that is not equal to any automaton’s index, or (c) \mathbb{R} : the set of real numbers. A variable may be a *local* variable with a name of the form $\text{variable_name}[i]$, or *global*, in which case its name does not have a symbolic index $[i]$. For example, $q[i] : L$, $p[i] : [N]_{\perp}$, and $x[i] : \mathbb{R}$ respectively define location, pointer, and real typed local variables, while $g : [N]_{\perp}$ is a global variable of pointer type. The sets of local and global variables are denoted by $V_L(N, i)$ and $V_G(N, i)$, respectively. The *valuation* of a variable v is a function that associates the variable name v to a value in its type $\text{type}(v)$. For a set of variables V , $\text{val}(V)$ is the set of valuations of each $v \in V$. For a set of variables V , $V' \triangleq \{v' | v \in V\}$ and $\dot{V} \triangleq \{v | v \in V \wedge \text{type}(v) = \mathbb{R}\}$. V' is used for specifying effects (resets) of discrete transitions and \dot{V} is used for specifying continuous dynamics. For a formula ϕ , let: (a) $\text{vars}(\phi)$ be the set of variables appearing in ϕ , (b) $\text{ivars}(\phi)$ be the set of distinct index variables appearing in ϕ , specifically $\text{ivars}(\phi) \triangleq \{v \in \phi | v \notin V(i) \wedge \text{type}(v) = [N]\}$, (c) $\text{free}(\phi)$ be the set of free variables appearing in ϕ , and (d) $\text{bound}(\phi)$ be the set of bound variables appearing in ϕ .

Definition 1. Let N be a symbol representing an arbitrary natural number and i be a symbol representing an arbitrary element of $[N]$. A hybrid automaton template $\mathcal{A}(N, i)$ is specified by the following syntactic components: (a) $V(N, i)$: a finite set of variable names with associated types. (b) L : a finite set of location names. (c) $\text{Init}(N, i)$: an initial condition formula over $V(N, i)$. (d) $\text{Trans}(N, i)$: a finite set of discrete transition statements, each of which is a tuple $\langle \text{from}, \text{to}, \text{grd}, \text{eff} \rangle$, where $\text{from}, \text{to} \in L$, grd is a formula over $V(N, i)$ called a guard and eff is a formula over $V(N, i) \cup V'(N, i)$ called an effect. (e) $\text{Traj}(N, i)$: for each element in L , there is a trajectory statement, each of which is a tuple $\langle \text{loc}, \text{inv}, \text{frate} \rangle$, where $\text{loc} \in L$, inv is a formula over $V(N, i)$

called an invariant, and **frate** is a formula over $V(N, i) \cup \dot{V}(N, i)$ called a flowrate that specifies how real variables evolve over time.

When clear from context, we drop the parameter N , for instance, a hybrid automaton template $\mathcal{A}(N, i)$ is written $\mathcal{A}(i)$, the set of variables is written $V(i)$, etc. The guard is an enabling condition that must be satisfied so that a transition *may* be taken. The effect models the update of state, and is a formula over the variables $V(i) \cup V(i)'$. A trajectory statement consists of an invariant condition **inv** and a flow rate **frate**. The invariant is an assertion involving only real variables of $\mathcal{A}(i)$. The flow rate associates each real-valued variable of $\mathcal{A}(i)$ with a rectangular differential inclusion.

2.1 Semantics of Hybrid Automata Networks

For a hybrid automaton template $\mathcal{A}(N, i)$, we define a transition system to formalize the semantics of the network where N instantiations of $\mathcal{A}(N, i)$ operate concurrently.

Definition 2. Let N be a symbol representing an arbitrary natural number. A hybrid automata network 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 sets of discrete transitions $\mathcal{D}^N \subseteq Q^N \times Q^N$ and continuous trajectories $\mathcal{T}^N \subseteq Q^N \times Q^N$.

A state \mathbf{x} in Q^N of \mathcal{A}^N is defined in terms of the valuations of all the variables of all its components. A *state* is a valuation of *all* the variables in V^N and is denoted by boldface \mathbf{v} , \mathbf{v}' , etc. The set of all states is called the state-space and is denoted Q^N . If a state $\mathbf{v} \in Q^N$ satisfies a formula ϕ —that is, the corresponding variable valuations result in ϕ evaluating to *true*—we write $\mathbf{v} \models \phi$. For a formula ϕ with $\text{vars}(\phi) \subseteq V(i)$, the corresponding states $\mathbf{x} \in Q^N$ satisfying ϕ are $\llbracket \phi \rrbracket \triangleq \{\mathbf{x} \in Q^N \mid \mathbf{v} \models \phi\}$. For instance, the initial states $\Theta^N \triangleq \llbracket \text{Init}(i) \rrbracket$ are the states satisfying $\text{Init}(i)$. For some state \mathbf{v} , the valuation of a particular local variable $x[i] \in V_L(i)$ for automaton $\mathcal{A}(i)$ is denoted by $\mathbf{v}.x[i]$, and $\mathbf{v}.g$ for some global variable g in $V_G(i)$. For a set of variables V , the valuations of each $v \in V$ at state \mathbf{v} is denoted by $\mathbf{v}.V$. For a formula ϕ and a set of variables $V \subseteq \text{vars}(\phi)$, let $\phi \downarrow V$ be the projection of ϕ onto the variables V , such that $\text{vars}(\phi \downarrow V) = V$ and $\llbracket \phi \rrbracket \subseteq \llbracket \phi \downarrow V \rrbracket$, which can be computed by eliminating the existential quantifiers from the formula $\exists \text{vars}(\phi) \setminus V : \phi$. The evolution of the states of \mathcal{A}^N are describing 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 variables may be updated by T^N . Discrete transitions \mathcal{D}^N model instantaneous changes and continuous trajectories \mathcal{T}^N model evolution over a real time interval. When necessary to disambiguate state updates owing to either discrete transitions and continuous trajectories, we write $\mathbf{v} \rightarrow_{\mathcal{D}^N} \mathbf{v}'$ or $\mathbf{v} \rightarrow_{\mathcal{T}^N} \mathbf{v}'$, respectively.

Discrete Transitions. Discrete transitions model atomic, instantaneous updates of state due to *one* automaton in the network \mathcal{A}^N . 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 $t \in \text{Trans}(i)$. There is a discrete transition $\mathbf{v} \rightarrow$

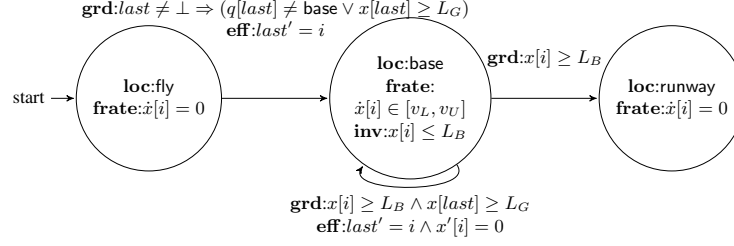


Fig. 1. Hybrid automaton template of aircraft $i \in [N]$ for a simplified SATS protocol. Here, $x[i]$ is the position of aircraft i from the start of the base location, $q[i]$ is the discrete location, $last$ is the index of the last aircraft to enter the base, L_G is the spacing length aircraft must have traversed in the base before another aircraft may enter the base, and L_B is the length of the base [24].

$\mathbf{v}' \in \mathcal{D}^N$ iff: $\exists i \in [N] \exists t \in \text{Trans}(i) : \mathbf{v}.V(i) \models \text{grd}(t, i) \wedge \mathbf{v}'.V(i) \models \text{eff}(t, i) \wedge (\forall j \in [N] : j \neq i \Rightarrow \mathbf{v}'.V(j) = \mathbf{v}.V(j))$. From the pre-state \mathbf{v} , any automaton $\mathcal{A}(i)$ in the network \mathcal{A}^N that has some transition where \mathbf{v} satisfies its guard may update its post-state according to the transition's effect, while the variable valuations of all the other automata in \mathcal{A}^N remain unchanged.

Continuous Trajectories. Continuous trajectories model update of state over intervals of real time. Informally, there is a trajectory $\mathbf{v} \rightarrow \mathbf{v}' \in \mathcal{T}^N$ 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 individual trajectory statements, and (b) while ensuring the invariants of *all* automata along the entire trajectory. Formally, trajectories are defined as solutions of differential equations or inclusions specified in the trajectory statements of $\mathcal{A}(i)$. Let $X(N, i) \triangleq \{v \in V(N, i) \mid \text{type}(v) = \mathbb{R}\}$ be the set of variables of $\mathcal{A}(i)$ with real type. For a state \mathbf{v} , a location m , a real time t , and a variable $v \in X(i)$ with real type, let $\text{flow}(\mathbf{v}, m, v, t) = \mathbf{v}.v + \int_{\tau=0}^t \text{frate}(m, v) d\tau$. Since frate may specify a differential inclusion, flow is a set-valued function. There is a trajectory $\mathbf{v} \rightarrow \mathbf{v}' \in \mathcal{T}^N$ iff: $\exists t_e \in \mathbb{R}_{\geq 0} \forall t_p \in \mathbb{R}_{\geq 0} \forall i \in [N] \exists m \in L : t_p \leq t_e \wedge \text{flow}(\mathbf{v}, m, X(i), t_p) \models \text{inv}(m, i) \wedge \mathbf{v}'.X(i) \in \text{flow}(\mathbf{v}, m, X(i), t_e)$. For each $i \in [N]$ and each real variable $x[i]$, $\mathbf{v}.x[i]$ must evolve to the valuations $\mathbf{v}'.x[i]$, in exactly t_e time in some location $m \in L$ according to the flow rates allowed for $x[i]$ in location m . In addition, all intermediate states along the trajectory must satisfy the invariant $\text{inv}(m, i)$.

Executions and 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 $\alpha = \mathbf{v}_0, \mathbf{v}_1, \dots$ such that $\mathbf{v}_0 \in \Theta^N$, and for each index k appearing in the sequence, $(\mathbf{v}_k, \mathbf{v}_{k+1}) \in T^N$. A state \mathbf{x} is *reachable* if there is a finite execution ending with \mathbf{x} . The set of reachable states for \mathcal{A}^N is $\text{Reach}(\mathcal{A}^N)$. The set of reachable states for \mathcal{A}^N starting from an arbitrary subset $\mathbf{V}_0 \subseteq Q^N$ is $\text{Reach}(\mathcal{A}^N, \mathbf{V}_0)$. An *invariant* for \mathcal{A}^N is any set of states that contains $\text{Reach}(\mathcal{A}^N)$.

Example 1. Each aircraft in the Small Aircraft Transportation System (SATS) [24] can be modeled as an instance of the template $\mathcal{A}(i)$ [20, 21, 25], and we present a simplified model of SATS in Figure 1. Aircraft communicate by reading the valuations of discrete variables and continuous positions using pointer variables to reach the runway location while maintaining a safe separation. Before attempting the landing procedure, aircraft i checks if any other aircraft already attempting to land is sufficiently far away— L_G

distance—from the geographic start of the approach to the runway, measured along one-dimension. If so, aircraft i 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 base location, the aircraft may either land, or return to the start of the base location. The main safety property in SATS is safe separation, which is informally specified as: if any aircraft i in the base location is ahead of any other aircraft j in the base location, then the positions of the aircraft are actually separated by at least L_S distance. Formally, safe separation is: $\forall i, j \in [N] : (i \neq j \wedge q[i] = \text{base} \wedge q[j] = \text{base} \wedge x[i] > x[j]) \Rightarrow (x[i] - x[j] \geq L_S)$.

3 Anonymized State-Space Representation

For any fixed $N \in \mathbb{N}$, let i be a symbol representing an arbitrary element of $[N]$, and for the hybrid automaton template $\mathcal{A}(N, i)$, the composed automaton modeling a network of size N is \mathcal{A}^N (Definition 2). We fix \mathcal{A}^N and present an algorithm for computing $\text{Reach}(\mathcal{A}^N)$ that takes advantage of the symmetries in the template $\mathcal{A}(i)$ instantiated in \mathcal{A}^N . The representation of $\text{Reach}(\mathcal{A}^N)$ is *anonymized*, so numerical automaton indices—1, 2, ..., N —are not explicitly enumerated and are instead modeled using symbolic indices— i_1, i_2, \dots, i_N . Frequently, the number of symbolic indices needed to represent equivalent states is significantly smaller than the number of numerical indices. For a given state $\mathbf{x} \in Q^N$, the set of corresponding states $\mathbf{X} \subseteq Q^N$ that are equivalent modulo indices is obtained by substituting any numerical index i of all local variables $v[i] \in V_L(i)$ with a symbolic index j with type $[N]$.

Definition 3. Two states $\mathbf{x}, \mathbf{x}' \in Q^N$ of \mathcal{A}^N , are equivalent modulo indices if there exists a bijection $\pi : [N] \rightarrow [N]$ such that for each $v[i] \in V(i)$, $\mathbf{x}.v[i] = \mathbf{x}'.v[\pi(i)]$. For a state $\mathbf{x} \in Q^N$ of \mathcal{A}^N , the set of states $\varepsilon(\mathbf{x})$ that is equivalent modulo indices to \mathbf{x} is: $\varepsilon(\mathbf{x}) \triangleq \{\mathbf{x}' \in Q^N \mid \mathbf{x} \text{ and } \mathbf{x}' \text{ are equivalent modulo indices}\}$.

We note this is the same type of definition as the existence of an automorphism used in [1–3]. A state is equivalent modulo indices to itself by picking the bijection π to be the identity mapping. For a formula ϕ , we will overload π and write $\pi(\phi)$, which modifies ϕ by applying π to each index variable $i \in \text{ivars}(\phi)$. The anonymized representation takes this idea a step further by utilizing symbolic names for process indices along with counters, and a formula representing the valuations of any global variables.

Definition 4. For a fixed $N \in \mathbb{N}$ and a template $\mathcal{A}(N, i)$, consider the automaton network \mathcal{A}^N (Definition 2). An anonymized state S of \mathcal{A}^N is a tuple $\langle \text{Classes}, G \rangle$, where:

- (a) Each anonymized class $C \in \text{Classes}$ is a tuple $C \triangleq \langle \text{Count}, I, \text{Form} \rangle$, where:
 - (i) Form is a quantifier-free formula over the variables $V_L(i_1) \cup \dots \cup V_L(i_{C.I})$, where i_1, \dots, i_I are I distinct symbolic index variables.
 - (ii) I is a natural number called the class's rank, which is equal to the number of distinct symbolic index variables appearing in Form : $I = |\text{ivars}(\text{Form})|$.
 - (iii) Count is a natural number called the class's count, and satisfies $N \geq \text{Count} \geq |I|$. The count is the number of automata of class C .

Additionally, the sum of all the class counts in S equals N : $N = \sum_{C \in S.\text{Classes}} C.\text{Count}$, where $C.\text{Count}$ is the count of class C .

(b) G is a quantifier-free formula over the global variables V_G .

For an anonymized class C , requirement (iii) of Definition 4 that $C.\text{Count} \geq |C.I|$ means the number of automata satisfying Form is at least the number of distinct index variables appearing in Form . We use the $(.)$ notation to refer to particular elements of tuples. For example, $C.\text{Count}$ refers to the count of anonymized class C , $C.\text{Form}$ refers to its formula, etc. When the number of index variables $C.I$ is clear from context, we drop it from the C tuple and write $\langle \text{Count}, \text{Form} \rangle$. Two anonymized classes C_1 and C_2 over the same symbolic indices ($\text{ivars}(C_1.\text{Form}) = \text{ivars}(C_2.\text{Form})$) are equivalent and write $C_1 \equiv C_2$ iff they have the same class formulas and class counts:⁴

Definition 5. Two classes C_1 and C_2 are equivalent, $C_1 \equiv C_2$ iff $C_1.\text{Count} = C_2.\text{Count} \wedge C_1.\text{Form} \Leftrightarrow C_2.\text{Form}$.

Here, equivalence between the class formulas is a semantic and not syntactic notion, and means the formula $C_1.\text{Form} \Leftrightarrow C_2.\text{Form}$ is valid. We say two anonymized states S_1 and S_2 are equivalent and write $S_1 \equiv S_2$ iff they have the same state counts, the classes in their sets of classes are equivalent, and their global formulas are equivalent.

Definition 6. Two anonymized states S_1 and S_2 are equivalent, $S_1 \equiv S_2$, iff $\forall C_1 \in S_1.\text{Classes} \exists C_2 \in S_2.\text{Classes} C_1 \equiv C_2 \wedge G_1 \equiv G_2$.

We make the following assumption about the format of the class formulas.

Assumption 1. For an anonymized state S , for each class $C \in \text{Classes}$, the class formula $C.\text{Form}$ is in conjunctive normal form (CNF), and for each index $i \in \{i_1, \dots, i_{C.I}\}$, $C.\text{Form}$ contains an equality $q[i] = l$ for some location $l \in L$.

For example, Equation 1 (arising from Example 1) satisfies this assumption. This assumption ensures that each class has a control location specified to determine the transitions and trajectories that may be possible (recall Definition 1). Under Assumption 1, the interpretation of an anonymized state S corresponds to a set of states of Q^N , which we write as $\llbracket S \rrbracket$ and define formally next. Since the class formulas of S are over the variables of automata with symbolic indices, the interpretation instantiates the symbolic indices with specific elements of $[N]$, which yields the set of states that are equivalent modulo indices. First, we define a notion of consistent partitions of these indices.

⁴ It is possible for classes with different ranks to represent the same states. For example, consider $S_1 = \langle \{ \langle 2, 2, q[i_1] = \text{base} \wedge q[i_2] = \text{base} \rangle \}, \text{last} = \perp \rangle$ and $S_2 = \langle \{ \langle 2, 1, q[i_1] = \text{base} \rangle \}, \text{last} = \perp \rangle$, which both represent there are two automata with location base and last is \perp , i.e., $\llbracket S_1 \rrbracket = \llbracket S_2 \rrbracket$. However, we allow classes of different ranks because classes of different ranks may not be expressible. For example, there is no way to express the following using rank 1 classes: $\langle \{ \langle 2, 2, q[i_1] = \text{base} \wedge q[i_2] = \text{base} \wedge x[i_1] \geq x[i_2] \rangle \}, \text{last} = \perp \rangle$, which expresses that there are two automata in base with one's position at least as large as the other's.

Definition 7. For an anonymized state $S = \left\langle \left\{ \underbrace{\langle \text{Count}_1, \mathbb{I}_1, \text{Form}_1 \rangle}_{C_1}, \dots, \underbrace{\langle \text{Count}_k, \mathbb{I}_k, \text{Form}_k \rangle}_{C_k} \right\}, G \right\rangle$,

we instantiate the set of symbolic indices $\{i_1, \dots, i_{\mathbb{I}_k}\}$ with all possible values in $[N]$ as follows. A consistent partition of $[N]$,

$$P = \left\{ \underbrace{\{P_1^1, \dots, P_1^{\mathbb{I}_1}\}}_{P_1}, \dots, \underbrace{\{P_k^1, \dots, P_k^{\mathbb{I}_k}\}}_{P_k} \right\}$$

is a partition of $[N]$, such that, for any $P_j \subseteq P$, (a) $|P_j| = \text{Count}_j$ and (b) P_j is partitioned into \mathbb{I}_j sets $P_j^1, \dots, P_j^{\mathbb{I}_j}$ (and we recall that \mathbb{I}_j is the rank of C_j).

For a consistent partition P , we note that (a) $\sum_{P_j \in P} |P_j| = N$, since P partitions $[N]$, and (b) $\text{Count}_j \geq \mathbb{I}_j$ (by Definition 4, (iii)). For example, consider the anonymized state (arsing from Example 1, Figure 1) with count three and rank two:

$$\langle \{ \langle 3, 2, q[i_1] = \text{base} \wedge q[i_2] = \text{base} \wedge x[i_2] \geq x[i_1] + L_S \rangle \}, \text{last} = i_1 \rangle. \quad (1)$$

One consistent partition is: $P = \{P_1^1, P_1^2\}$ where $P_1^1 = \{1\}$ and $P_1^2 = \{2, 3\}$. The set $\{\{1, 2, 3\}\}$ is *not* a consistent partition since it is partitioned into one set, but $\mathbb{I} = 2$, and Definition 8 requires each $P_j \in P$ be partitioned into \mathbb{I}_j partitions. For an anonymized state S , the set of consistent partitions $\text{ConsPart}(S)$ are all consistent partitions of $[N]$. Continuing Example 1 for S , $\text{ConsPart}(S)$ is $\{\{\{1\}, \{2, 3\}\}, \{\{2\}, \{1, 3\}\}, \{\{3\}, \{1, 2\}\}, \{\{1, 2\}, \{3\}\}, \{\{1, 3\}, \{2\}\}, \text{and } \{\{2, 3\}, \{1\}\}\}$. All these partitions define the full set of states $\llbracket S \rrbracket$ that the anonymized state S represents. This is equivalent to all the states equivalent modulo indices to the states $\llbracket S_P \rrbracket$ for a particular consistent partition P .

Definition 8. For an anonymized state S and a consistent partition $P \in \text{ConsPart}(S)$, the set of states of network \mathcal{A}^N represented by S corresponding to P are:

$$\llbracket S_P \rrbracket \triangleq \{ \mathbf{x} \in Q^N \mid \mathbf{x} \models G \wedge \text{Form}_1(P_1) \wedge \dots \wedge \text{Form}_k(P_k) \}, \quad (2)$$

where each $\text{Form}_j(P_j) \triangleq \forall i_j^1 \in P_j^1, \dots, i_j^{\mathbb{I}_j} \in P_j^{\mathbb{I}_j} : \text{Form}_j(i_j^1, \dots, i_j^{\mathbb{I}_j})$. The set of states of network \mathcal{A}^N represented by S with all consistent partitions is:

$$\llbracket S \rrbracket \triangleq \bigcup_{P \in \text{ConsPart}(S)} \llbracket S_P \rrbracket. \quad (3)$$

We have written $\text{Form}_j(i_j^1, \dots, i_j^{\mathbb{I}_j})$ to highlight that Form_j is over \mathbb{I}_j symbolic index variables. Note that $\text{Form}_j(P_j)$ is equivalent to a finite-length conjunction since each $P_j^{\mathbb{I}_j}$ is a finite set. The next lemma states that this definition of interpretations of anonymized states yields the same set of states as equivalence modulo identifiers.

Lemma 1. For an anonymized state S , for any $\mathbf{x} \in \llbracket S \rrbracket$, for any $\mathbf{x}' \in \varepsilon(\mathbf{x})$, $\mathbf{x}' \in \llbracket S \rrbracket$.


```

1  function areach( $\mathcal{A}(\mathcal{N}, i)$ , Init( $i$ ), N)
   AnonReach  $\leftarrow \emptyset$ 
3  Frontier  $\leftarrow \{ \{ \langle N, \text{Init}(i) \downarrow V_L(i) \rangle \}, \text{Init}(i) \downarrow V_G(i) \} \}$  // create initial anonymized state
   while Frontier  $\neq \emptyset$  // repeat until no new states are added to the frontier
5     FrontierNew  $\leftarrow \emptyset$  // initialize next frontier
     AnonReach  $\leftarrow \text{AnonReach} \cup \text{Frontier}$  // add frontier to reachable states
7     // compute successors of each anonymized state in the frontier
     foreach anonymized state S in Frontier
9         FrontierNew  $\leftarrow \text{Frontier}_{\text{New}} \cup \text{discPost}(S)$  // Figure 4
         FrontierNew  $\leftarrow \text{Frontier}_{\text{New}} \cup \text{contPost}(S)$  // Figure 5
11        FrontierNew  $\leftarrow \text{mergeAndDrop}(\text{Frontier}_{\text{New}}, \text{AnonReach})$  // Figure 3
         Frontier  $\leftarrow \text{Frontier}_{\text{New}}$ 
13  return AnonReach

```

Fig. 2. On-the-fly anonymized reachability algorithm. The inputs are an automaton template $\mathcal{A}(\mathcal{N}, i)$, an initial condition $\text{Init}(i)$, and a constant natural number N . The anonymized reachable states AnonReach are computed as a fixed-point starting from the anonymized initial states.

Continuing Example 1 with the consistent partition $P = \{\{1\}, \{2, 3\}\}$, the states represented by S_P are:

$$\begin{aligned}
\llbracket S_P \rrbracket &= \{ \mathbf{x} \in Q^3 \mid \mathbf{x} \models \forall i_1^1 \in P_1^1, i_1^2 \in P_1^2 : q[i_1] = \text{base} \wedge q[i_2] = \text{base} \wedge \\
&\quad x[i_2] \geq x[i_1] + L_S \wedge \text{last} = i_1 \} \\
&= \{ \mathbf{x} \in Q^3 \mid \mathbf{x} \models (q[1] = \text{base} \wedge q[2] = \text{base} \wedge q[3] = \text{base} \wedge \\
&\quad x[2] \geq x[1] + L_S \wedge x[3] \geq x[1] + L_S \wedge \text{last} = 1) \}.
\end{aligned}$$

Applying Lemma 1, $\llbracket S \rrbracket = \varepsilon(\llbracket S_P \rrbracket)$.

4 Anonymized Reachability of Hybrid Automata Networks

Next we describe an on-the-fly algorithm for overapproximating the reachable states of a network \mathcal{A}^N using anonymized states. We note that the CNF requirement (Assumption 1) is not restrictive: if a new class is created during the execution of the algorithm that contains disjunctions, it is split into multiple classes with CNF formulas. We use projections extensively, so recall the \downarrow notation introduced in Section 2.1.

Pseudocode for the reachability algorithm, areach appears in Figure 2. The algorithm operates on frontiers of reachable states represented by the set Frontier, which is initialized (line 3) to a singleton set with one class that has count equal to N and class formula equal to $\text{Init}(i) \downarrow V_L(i)$, which is $\text{Init}(i)$ projected onto the local variables. The global formula is initialized with $\text{Init}(i) \downarrow V_G(i)$, which is $\text{Init}(i)$ projected onto the global variables. The set of reachable anonymized states computed so far is the set AnonReach . Next (line 4), we remove an anonymized state S from Frontier, compute anonymized post-states from S , and continue until no new anonymized states

```

1  function mergeAndDrop(FrontierNew, AnonReach)
   foreach S in FrontierNew
3     if S  $\in \text{AnonReach}$  then FrontierNew = FrontierNew  $\setminus \{S\}$ 
     else
5         foreach distinct pair of anonymized classes  $\langle C_1, C_2 \rangle$  in S.Classes
             if  $\neg(C_1.\text{Form} \equiv C_2.\text{Form})$  is UNSAT then
7                 C1.Count  $\leftarrow C_1.\text{Count} + C_2.\text{Count}$  // if equivalent, sum counts
                 S.Classes  $\leftarrow S.\text{Classes} \setminus \{C_2\}$  // if equivalent, drop equivalent class
9  return FrontierNew

```

Fig. 3. mergeAndDrop combines classes with equivalent class formulas and sums their counts.

```

1  function discPost(S)
   StatesNew ← ∅
3  VS ← V'(i1) ∪ ... ∪ V'(ic,I)
   foreach anonymized class C in S.Classes
5     foreach symbolic index i in VS
       foreach transition t in Trans(i)
9         CNew.Form ← (C.Form ∧ S.G ∧ grd(t, i) ∧ eff(t, i)) ↓ V'(i) // compute new class
           // substitute primed variables with unprimed variables
10        CNew.Form ← Substitute(CNew.Form, V(i)', V(i))
           // project onto global variables for global constraint
11        SNew.G ← CNew.Form ↓ VG(i)
           // project onto local variables for local constraint
13        ⟨CNew.Count, CNew.Form⟩ ← ⟨1, CNew.Form ↓ VL(i)⟩
           SNew.Classes ← S.Classes \ {C} // remove pre-state class from post-state classes
           // add pre-state class to post-state classes if its count is at least its rank
15        if C.Count > C.I then SNew.Classes ← S.Classes ∪ {⟨C.Count - 1, C.Form⟩}
           // otherwise, pre-state class no longer exists (count less than rank)
17        else SNew.Classes ← S.Classes ∪ {⟨C.Count - 1, C.Form ↓ VS \ V(i)⟩}
19        SNew.Classes ← SNew.Classes ∪ {CNew} // add new class to new anonymized state
21  StatesNew ← StatesNew ∪ {SNew}
   return StatesNew

```

Fig. 4. `discPost` computes the post-states of an anonymized state S due to discrete transitions for an automaton with index i and states satisfying C 's formulas.

are added to `Frontier`. Anonymized post-states are added to the frontier using the set `FrontierNew` (line 5). Computing successors (post states)—the states reachable from S in one step—is composed of two parts: (a) computing the discrete successors corresponding to transitions (line 9), and (b) computing the continuous successors corresponding to trajectories (line 10).

Subroutine for Merging Classes. We first describe a subroutine, `mergeAndDrop` (Figure 3). It takes a set of anonymized states `FrontierNew` and returns a set of anonymized state that is guaranteed to both (a) not have any equivalent classes (lines 7 through 8) and (b) be new (not already represented in `AnonReach`) (line 3). Invariant 1 states that no two class formulas in any reachable anonymized state are equivalent, and Invariant 2 states that no two anonymized states in `AnonReach` are equivalent (Definition 6).

Invariant 1. For any $S \in \text{AnonReach}$, $C_1, C_2 \in S.\text{Classes}$, $C_1.\text{Form} \neq C_2.\text{Form}$.

Invariant 2. For any distinct $S_1, S_2 \in \text{AnonReach}$, $S_1 \neq S_2$.

Discrete Successors. The function `discPost` (Figure 4) computes the discrete successors from an anonymized state S in the frontier (Figure 2, line 9). The post-states `StatesNew` are added to the frontier `FrontierNew`. First, we iterate over each class C in `S.Classes` (line 4), and then we iterate over each index variable i in the set of index variables in the class formula, $\{i_1, \dots, i_{c,I}\}$ (line 5). Next, we iterate over the (syntactic) transitions `Trans(i)` of $\mathcal{A}(\mathcal{N}, i)$ (line 6). For a transition $t \in \text{Trans}(i)$ and an anonymized class C , line 7 computes the subsequent class from C under the transition t , made by the automaton with index i . This computation can be carried out using quantifier elimination procedures over the types of the variables appearing in the guard and effect of the transition t , and then syntactically unpriming all primed variables (representing successors) following quantifier elimination using `Substitute` (line 9). This step is an overapproximation, since it is computing the successors of each class regardless of the number of automata with states satisfying the anonymized class formula

```

1  function contPost(S)
   Vs ← VG
   // formula used to encode semantics of trajectories for all automata in the network
   postFormula ← te > 0 ∧ S.G
5  foreach anonymized class C in S.Classes // iterate over each class in pre-state
   Vs ← Vs ∪ V'(i1) ∪ ... ∪ V'(iC,I)
7   postFormula ← postFormula ∧ C.Form // encode pre-state class formula
   // determine the locations any automata may be in each class (recall Assumption 1)
9   foreach location m in L
      foreach i in {i1, ..., iC,I} // iterate over all indices (ranks)
11      if C.Form  $\not\models$  (q[i] = m) is UNSAT then // use location m if automaton i is in m
         // add the trajectory semantics overapproximating the post-states
13         postFormula ← postFormula ∧ inv(m, i) ∧ X(i) ∈ flow(m, X(i), te)
   postFormula ← postFormula ↓ Vs
15   postFormula ← Substitute(postFormula, V(i)', V(i))
   SNew ← RemapClasses(S, postFormula) // Figure 6
17  return SNew

```

Fig. 5. contPost function that computes the continuous successors from an anonymized state S.

Form, and just presuming *there is some* automaton with variable valuations satisfying Form. The post-state anonymized state S_{New} is constructed using the classes of the anonymized state S of the current iteration along with the new anonymized class, C_{New} (lines 14 through 19). First, the classes for S_{New} are set to be the anonymized classes of S, without the anonymized class of the current iteration, C (line 14). Next, if the class count of C is larger than its rank, then it is added to the classes of the post-state, with its count reduced by one to indicate some automaton has left the set of states satisfying the corresponding class formula (line 16). On the other hand, if the class count is equal or less than its rank, then the pre-state's anonymized class C would no longer satisfy the requirements of Definition 4, (iii), so its class formula is projected onto the variables of all automata except those of automaton i , the one making a transition (line 18). However, this may result in two classes with equivalent formulas, since the algorithm has not yet detected if any other classes had the same formula and assumed the new class C_{New} had a count of one, which is why we use mergeAndDrop (Figure 2, line 11).

Lemma 2. (Anonymized Discrete Successor Soundness) *For an anonymized state S, for any corresponding concretized state $\mathbf{x} \in \llbracket S \rrbracket$, if $\mathbf{x} \rightarrow_{\mathcal{D}^N} \mathbf{x}'$, then $\mathbf{x}' \in \llbracket \text{discPost}(S) \rrbracket$.*

Continuous Successors. An overapproximation of continuous successors are computed using contPost—shown in Figure 5—called from symreach (Figure 2, line 10). For an anonymized state S in the frontier, contPost computes an overapproximation of the post-states from S owing to the individual trajectories of all automata in the network for up to the most amount of time that can elapse before any invariant is violated. The anonymized state specifies a location $m \in L$ for each automaton in the network (recall Assumption 1), and each location m specifies a trajectory statement, so trajectories are defined for each automaton in the network. Each new anonymized state $S_{\text{New}} \in \text{States}_{\text{New}}$ computed corresponds to the trajectory semantics updating the continuous variables of *all* automata in the network \mathcal{A}^N . The variable *postFormula* encodes the trajectory semantics of all automata in the network \mathcal{A}^N (line 4), which is initially the constraint $t_e > 0$, indicating that some positive real amount of time t_e will elapse. However, for an anonymized state S, for distinct anonymized classes C_1, C_2 in S.Classes, the symbolic indices appearing in the formulas may be equal, i.e., $\exists i \in \text{ivars}(C_1)$ and $\exists j \in \text{ivars}(C_2)$ such that $i = j$. Since *postFormula* encodes the states of all automata in the network, the symbolic index variables appearing in any

```

1  function RemapClasses(S, postFormula)
   SNew.Classes = {}
3  foreach anonymized class C in S.Classes
   // project postFormula onto variables of indices in each pre-state class
5   Vs ← V(i1) ∪ ... ∪ V(ic,1)
   // create new class with post-state formula and copy pre-state count
7   (CNew.Count, CNew.Form) ← (C.Count, postFormula ↓ Vs)
   SNew.Classes ← SNew.Classes ∪ CNew // add new post-state class to post-state's classes
9  SNew.N ← S.N
   return SNew

```

Fig. 6. RemapClasses recreates variables in *postFormula* using their pre-state indices, class counts, and ranks to create the anonymized post-state S_{New} . It first projects onto the variables with indices of each class in the pre-state and then uses the pre-state count to ensure class counts remain constant over trajectories.

class formula of any anonymized class must be distinct. Rather than performing these tedious syntactic manipulations, we assume that for an anonymized state S , for distinct classes C_1, C_2 in $S.\text{Classes}$, $\forall i \in \text{ivars}(C_1), \forall j \in \text{ivars}(C_2)$, we have $i \neq j$.⁵

Each anonymized class formula $C.\text{Form}$ of an anonymized state S specifies the location(s) the automata are in (recall Assumption 1), so the first step is to determine the dynamics that will modify each class formula. This is accomplished by first determining the appropriate flow-rate conditions to use for each class in $S.\text{Classes}$, which can be detected by finding which Form imply the location variable $q[i]$ is in some location $m \in L$. If the control location of automaton i is found to be equal to location m , then the trajectory statement of location m is used to define the semantics of the time-evolution of i 's continuous variables (line 13). The semantics of trajectories result in *all* the automata's continuous variables evolving over time t_e , so the formula encoding the trajectory statements of all automata is conjuncted (line 13). The post-states are computed by projecting onto the primed variables of all classes, and then renaming primed variables with their unprimed counterparts (line 15).⁶ We call RemapClasses with the pre-state S and *postFormula*, which encodes the post-state constraints, to recreate classes from subformulas of *postFormula* (Figure 6 called at line 16). This is done to ensure the class counts are constant when computing post-states due to trajectories.

Lemma 3. (*Anonymized Continuous Successor Soundness*) For an anonymized state S , for any corresponding concretized state $\mathbf{x} \in \llbracket S \rrbracket$, if $\mathbf{x} \rightarrow_{\mathcal{T}_N} \mathbf{x}'$, then $\mathbf{x}' \in \llbracket \text{contPost}(S) \rrbracket$.

The next invariant states that the sum of all the class counts Count equals N . It follows from the definitions of discPost and contPost , since discPost always decreases class counts by the same amount it increases them—so the sum remains invariant—and contPost does not change class counts, only class formulas. Additionally, mergeAndDrop changes class counts, but their sum remains the same since it removes any duplicate classes after adding their counts (Figure 3, lines 7 through 8).

Invariant 3. For any $S \in \text{AnonReach}$, $N = \sum_{C \in S.\text{Classes}} C.\text{Count}$.

Theorem 1 states partial correctness of the reachability algorithm, namely soundness, that the concretization of the set of anonymized reachable states AnonReach contains the set of reachable states for network \mathcal{A}^N , and follows from Lemmas 2 and 3. The

⁵ This is a tedious, but trivial invariant that we maintain in our implementation in *Passel*, so we make this assumption for clarity of presentation only.

⁶ This may result in a DNF formula, and if so, each conjunctive clause is added as a new anonymized state by iterating over the conjunctive clauses so all class formulas are CNF.

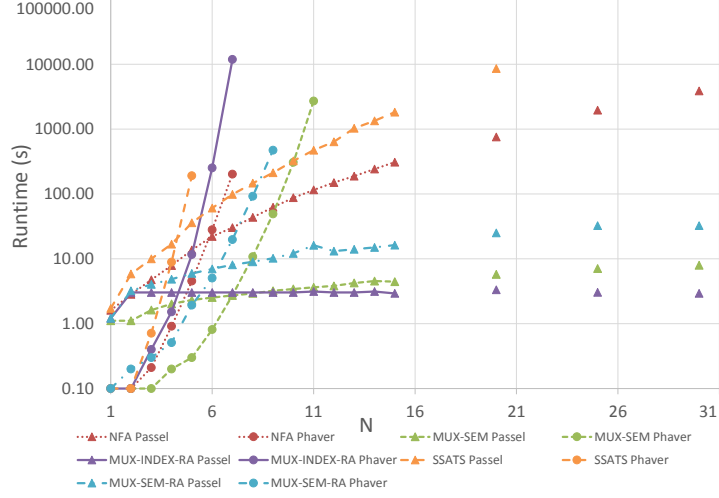


Fig. 7. Runtime comparison of PHAVer and *Passel*'s anonymized reachability. Vertical axis is logarithmic and has units of seconds, and horizontal axis is number of automata, N .

approximation comes from two sources. First, index-typed variables are abstracted to be equal or not equal to some index only. Second, the rectangular dynamics are overapproximated.

Theorem 1. (*Soundness*) For a fixed $N \in \mathbb{N}$, for the network \mathcal{A}^N composed of N instantiations of the template $\mathcal{A}(N, i)$, the anonymized reachable states AnonReach computed by areach overapproximate the reachable states of \mathcal{A}^N : $\text{Reach}(\mathcal{A}^N) \subseteq \llbracket \text{AnonReach} \rrbracket$.

5 Experimental Results

The anonymized reachability algorithm has been implemented in the *Passel* verification tool [20–22]. The current implementation of *Passel* uses the SMT solver Z3 [23] for proving validity, checking satisfiability, and performing quantifier elimination. *Passel* is written in C# and uses the managed .NET API to version 4.3.2 of Z3 (from the latest unstable branch of the Z3 repository as of April 2014). *Passel* proves validity of a formula ϕ by checking unsatisfiability of $\neg\phi$. The variables $V(i)$ used in defining $\mathcal{A}(N, i)$ are specified to the SMT solver. Each local variable $v[i] \in V_L(i)$ is modeled as an uninterpreted function $v : [N] \rightarrow \text{type}(v)$. *Passel* automatically generates and asserts trivial data-type lemmas that the SMT solver requires. The experiments were conducted in an Ubuntu 12.04 VMWare virtual machine with 4 GB RAM allocated running *Passel* through Mono, executed on a modern laptop with a quad-core Intel i7 processor running Windows 8 with 16 GB RAM physically available. For comparison purposes, we evaluated *Passel*, PHAVer (version 0.38), and SpaceEx (version 0.9.8b). We do not present experimental results for SpaceEx, as the only scenario—out of the PHAVer, LeGuernic-Girard [LGG], and STC scenarios—that can compute the reachable states of systems with rectangular differential inclusion dynamics ($\dot{x} \in [a, b]$ for real constants $a \leq b$) adequately is the PHAVer scenario, so the results are equivalent.

Figures 7 and 8 show, respectively, a runtime and memory usage comparison between PHAVer and *Passel* for several examples as a function of N , the number of au-

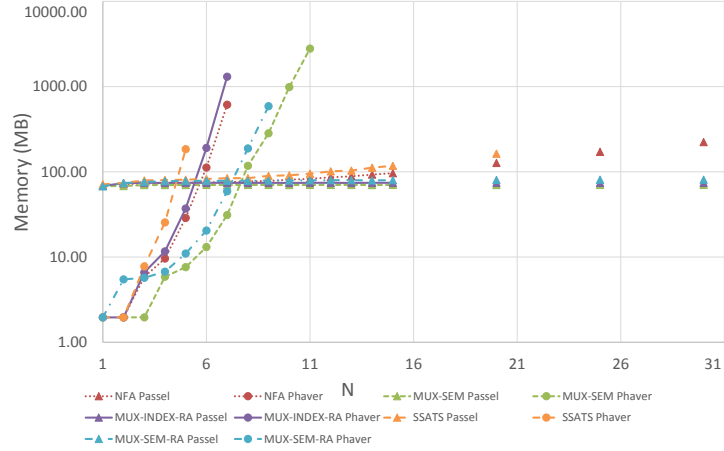


Fig. 8. Memory usage comparison of PHAVer and *Passel*'s anonymized reachability. Vertical axis scale is logarithmic and has units of megabytes, and horizontal axis is number of automata, N .

tomata in the finite instantiation of the network.⁷ The examples include the simplified SATS model (from Figure 1), several timed mutual exclusion algorithms, several purely discrete examples, and all properties were safety properties (invariants), such as safe separation in SATS, mutual exclusion, etc. Comparing all the examples, the anonymized reachability method implemented in *Passel* allows us to compute the reachable states of networks composed of many more automata than PHAVer, which runs out of memory on all examples for $N \leq 11$. The experimental results indicate that the primary advantage is reduced memory growth. Even for networks of tens of automata, *Passel* never uses more than a few hundred megabytes of memory as shown in Figure 8. The runtime required by *Passel* could be reduced by performing some operations more efficiently in the implementation—particularly the checks to determine if a new anonymized state representation is actually new or not—which we plan to implement for future work.

6 Summary

In this paper, we present an on-the-fly forward reachability algorithm that computes an anonymized representation of the reachable states for hybrid automata networks consisting of N instantiations of a template $\mathcal{A}(N, i)$. The anonymized representation avoids generating all permutations of automata indices and states. We showed it to be effective at computing the reachable states of networks with tens of automata for several examples, with significantly lower memory usage than PHAVer.

References

1. E. M. Clarke, R. Enders, T. Filkorn, and S. Jha, "Exploiting symmetry in temporal logic model checking," *Formal Methods in System Design*, vol. 9, pp. 77–104, 1996.
2. C. N. Ip and D. L. Dill, "Better verification through symmetry," *Formal Methods in System Design*, vol. 9, pp. 41–75, 1996.
3. E. A. Emerson and A. P. Sistla, "Symmetry and model checking," *Formal Methods in System Design*, vol. 9, no. 1-2, pp. 105–131, 1996.

⁷ *Passel* and the examples may be downloaded from: <https://publish.illinois.edu/passel-tool/>.

4. C. N. Ip and D. L. Dill, "Verifying systems with replicated components in Mur ϕ ," *Formal Methods in System Design*, vol. 14, no. 3, May 1999.
5. V. Braberman, D. Garbervetsky, and A. Olivero, "Improving the verification of timed systems using influence information," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. LNCS, J.-P. Katoen and P. Stevens, Eds. Springer, 2002, vol. 2280, pp. 21–36.
6. G. Behrmann, P. Bouyer, E. Fleury, and K. G. Larsen, "Static guard analysis in timed automata verification," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. LNCS, H. Garavel and J. Hatcliff, Eds. Springer, 2003, vol. 2619, pp. 254–270.
7. M. Hendriks, G. Behrmann, K. G. Larsen, P. Niebert, and F. W. Vaandrager, "Adding symmetry reduction to UPPAAL," in *Formal Modeling and Analysis of Timed Systems (FORMATS '03)*, ser. LNCS, K. G. Larsen and P. Niebert, Eds., no. 2791. Springer-Verlag, 2004, pp. 46–59.
8. E. Emerson and T. Wahl, "Dynamic symmetry reduction," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. LNCS, N. Halbwachs and L. Zuck, Eds. Springer, 2005, vol. 3440, pp. 382–396.
9. W. D. Obal, M. McQuinn, and W. Sanders, "Detecting and exploiting symmetry in discrete-state Markov models," *Reliability, IEEE Transactions on*, vol. 56, no. 4, pp. 643–654, Dec. 2007.
10. T. Wahl, N. Blanc, and E. Emerson, "SVISS: Symbolic verification of symmetric systems," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. LNCS, C. Ramakrishnan and J. Rehof, Eds. Springer, 2008, vol. 4963, pp. 459–462.
11. G. Basler, M. Mazzucchi, T. Wahl, and D. Kroening, "Symbolic counter abstraction for concurrent software," in *Computer Aided Verification*, ser. LNCS, A. Bouajjani and O. Maler, Eds. Springer, 2009, vol. 5643, pp. 64–78.
12. J. Sun, Y. Liu, J. S. Dong, Y. Liu, L. Shi, and E. André, "Modeling and verifying hierarchical real-time systems using stateful timed csp," *ACM Trans. Softw. Eng. Methodol.*, vol. 22, no. 1, pp. 1–29, Mar. 2013.
13. Y. Si, J. Sun, Y. Liu, and T. Wang, "Improving model checking stateful timed csp with non-zenoness through clock-symmetry reduction," in *Formal Methods and Software Engineering*, ser. LNCS, L. Groves and J. Sun, Eds. Springer, 2013, vol. 8144, pp. 182–198.
14. D. L. Dill, "The mur ϕ verification system," in *Proceedings of the 8th International Conference on Computer Aided Verification*, ser. CAV '96. London, UK, UK: Springer-Verlag, 1996, pp. 390–393.
15. J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi, "UPPAAL: A tool suite for automatic verification of real-time systems," in *Hybrid Systems III*, ser. LNCS, R. Alur, T. Henzinger, and E. Sontag, Eds. Springer, 1996, vol. 1066, pp. 232–243.
16. J. Sun, Y. Liu, J. Dong, and J. Pang, "PAT: Towards flexible verification under fairness," in *Computer Aided Verification*, ser. LNCS, A. Bouajjani and O. Maler, Eds. Springer, 2009, vol. 5643, pp. 709–714.
17. M. Hendriks, "Model checking timed automata: Techniques and applications," Ph.D. dissertation, University of Nijmegen, The Netherlands, 2006.
18. C. Herrera, B. Westphal, S. Feo-Arenis, M. Muñiz, and A. Podelski, "Reducing quasi-equal clocks in networks of timed automata," in *Formal Modeling and Analysis of Timed Systems*, ser. LNCS, M. Jurdzinski and D. Nickovic, Eds. Springer, 2012, vol. 7595, pp. 155–170.
19. S. Bogomolov, C. Herrera, M. Muñiz, B. Westphal, and A. Podelski, "Quasi-dependent variables in hybrid automata," in *17th International Conference on Hybrid Systems: Computation and Control*, 2014.
20. 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)*, ser. LNCS. Springer, June 2012, vol. 7273.
21. —, "Invariant synthesis for verification of parameterized cyber-physical systems with applications to aerospace systems," in *Proceedings of the AIAA Infotech at Aerospace Conference (AIAA Infotech 2013)*, Boston, MA, Aug. 2013.
22. T. T. Johnson, "Uniform verification of safety for parameterized networks of hybrid automata," Ph.D. dissertation, University of Illinois at Urbana-Champaign, Urbana, IL 61801, 2013.
23. L. De Moura and N. Björner, "Z3: An efficient SMT solver," in *Proc. of 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS '08/ETAPS '08. Springer-Verlag, 2008, pp. 337–340.
24. 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," NASA, Tech. Rep. NASA/TP-2006-214512, L-19215, Oct. 2006.
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.

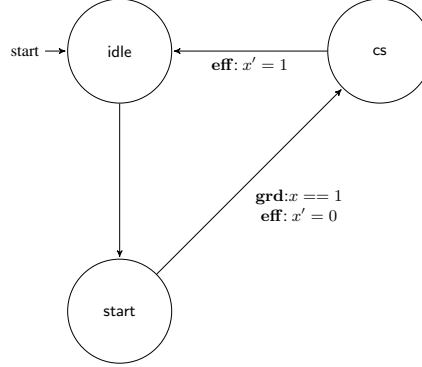


Fig. 9. MUX-SEM mutual exclusion algorithm for automaton $\mathcal{A}(i)$ for illustrating the anonymized state-space representation.

A Appendix: Additional Case Study Details

A.1 SATS Example from Example 1 and Figure 1

The SSATS simplified Small Aircraft Transportation System example is the most challenging example we evaluate. PHAVer was able to compute the reachable states of SSATS up to $N = 5$ using around 350 MB (as shown in Figure 8) and requiring a few minutes (Figure 7). In comparison, *Passel* was able to compute the anonymized reachable states of SSATS for up to $N = 20$ using only about 200 MB memory, although this took about 2.5 hours to complete.

A.2 MUX-SEM Mutual Exclusion Algorithm

For illustrating the anonymized representation of the state-space, we use the MUX-SEM mutual exclusion example shown graphically in Figure 9 and as a *Passel* specification in Figure 10. Note that this example does not have timing information, but has simple enough reachable states to be used for illustration purposes. MUX-SEM has one local variable $q[i]$ with type $\text{loc} \triangleq \{\text{idle}, \text{start}, \text{cs}\}$ and one global variable x of Boolean \mathbb{B} type. For $N = 3$, the product of the types is $\text{loc}^3 \times \mathbb{B}$ which has $2 |\text{loc}^3| = 54$ elements, which is the number of elements in the state-space of \mathcal{A}^3 . For \mathcal{A}^3 , the reachable states


```

2      variable name='q[i]' type='L'      // location local variable
      variable name='x' type='boolean' // global mutex variable

4      location name='idle'
      location name='start'
6      location name='cs'

8      transition from='idle' to='start'
      transition from='start' to='cs'
10     grd: x = 1
      eff: x' = 0
12     transition from='cs' to='idle'
      eff: x' = 0
14
16     property: forall i, j (i != j and q[i] = cs) implies (q[j] != cs)
      initially: forall i (q[i] = idle and x = 1)

```

Fig. 10. *Passel* input file specifying automaton template $\mathcal{A}(i)$ for mutual exclusion algorithm MUX-SEM.

are encoded as the DNF formula:

$$(q[1] = \text{idle} \wedge q[2] = \text{idle} \wedge q[3] = \text{idle} \wedge x = 1) \vee \quad (4)$$

$$(q[1] = \text{start} \wedge q[2] = \text{idle} \wedge q[3] = \text{idle} \wedge x = 1) \vee \quad (5)$$

$$(q[1] = \text{idle} \wedge q[2] = \text{start} \wedge q[3] = \text{idle} \wedge x = 1) \vee \quad (6)$$

$$(q[1] = \text{idle} \wedge q[2] = \text{idle} \wedge q[3] = \text{start} \wedge x = 1) \vee \quad (7)$$

$$(q[1] = \text{start} \wedge q[2] = \text{start} \wedge q[3] = \text{idle} \wedge x = 1) \vee (q[1] = \text{start} \wedge q[2] = \text{idle} \wedge q[3] = \text{start} \wedge x = 1) \vee$$

$$(q[1] = \text{idle} \wedge q[2] = \text{start} \wedge q[3] = \text{start} \wedge x = 1) \vee (q[1] = \text{start} \wedge q[2] = \text{start} \wedge q[3] = \text{start} \wedge x = 1) \vee$$

$$(q[1] = \text{cs} \wedge q[2] = \text{idle} \wedge q[3] = \text{idle} \wedge x = 0) \vee (q[1] = \text{idle} \wedge q[2] = \text{cs} \wedge q[3] = \text{idle} \wedge x = 0) \vee$$

$$(q[1] = \text{idle} \wedge q[2] = \text{idle} \wedge q[3] = \text{cs} \wedge x = 0) \vee (q[1] = \text{cs} \wedge q[2] = \text{start} \wedge q[3] = \text{idle} \wedge x = 0) \vee$$

$$(q[1] = \text{start} \wedge q[2] = \text{cs} \wedge q[3] = \text{idle} \wedge x = 0) \vee (q[1] = \text{start} \wedge q[2] = \text{idle} \wedge q[3] = \text{cs} \wedge x = 0) \vee$$

$$(q[1] = \text{cs} \wedge q[2] = \text{start} \wedge q[3] = \text{start} \wedge x = 0) \vee (q[1] = \text{start} \wedge q[2] = \text{cs} \wedge q[3] = \text{start} \wedge x = 0) \vee$$

$$(q[1] = \text{start} \wedge q[2] = \text{start} \wedge q[3] = \text{cs} \wedge x = 0) \vee (q[1] = \text{cs} \wedge q[2] = \text{idle} \wedge q[3] = \text{start} \wedge x = 0) \vee$$

$$(q[1] = \text{idle} \wedge q[2] = \text{cs} \wedge q[3] = \text{start} \wedge x = 0) \vee (q[1] = \text{idle} \wedge q[2] = \text{start} \wedge q[3] = \text{cs} \wedge x = 0). \quad (8)$$

A.3 MUX-SEM Anonymized Reachable States and Transition Graph

Figure 13 shows the anonymized reachable states and transition graph for MUX-SEM (see Figures 9 and 10) for $N = 3$, and Figure 14 shows the anonymized reachable states and transition graph for $N = 4$. The comparison of Figures 13 and 14 illustrates the crux of the benefit of the anonymized reachability representation and algorithm. First, observe that no permutations are explicitly enumerated, unlike as was required in the typical representation shown in Equations 4 through 8. Second, observe that while the number of anonymized states increases between $N = 3$ to $N = 4$, the number of distinct anonymized classes remains equivalent. This illustrates that for computing the anonymized reachable states for $N = 3$ and $N = 4$ requires simply counting the additional number of automata with states satisfying each of the anonymized classes'

formulas. This result coincides with related notions of bounding the diameter of the reachability graph. In the implementation in *Passel*, all of these data structures are implemented as hash sets, so the number of formulas does not increase between computing the reachable states of $N = 3$ to larger N . Finally, by inspecting the anonymized reachable states in Figures 13 and 14, we may conclude that MUX-SEM satisfies the mutual exclusion property:

$$\forall i, j \in [N] : (i \neq j \wedge q[i] = \text{cs}) \implies q[j] \neq \text{cs}.$$

This is because, for any anonymized state $S \in \text{AnonReach}$, any class $C \in S.\text{Classes}$ where the class formula $q[i] = \text{cs}$ is satisfied has at most count 1, meaning there is at most one automaton in the critical section cs .

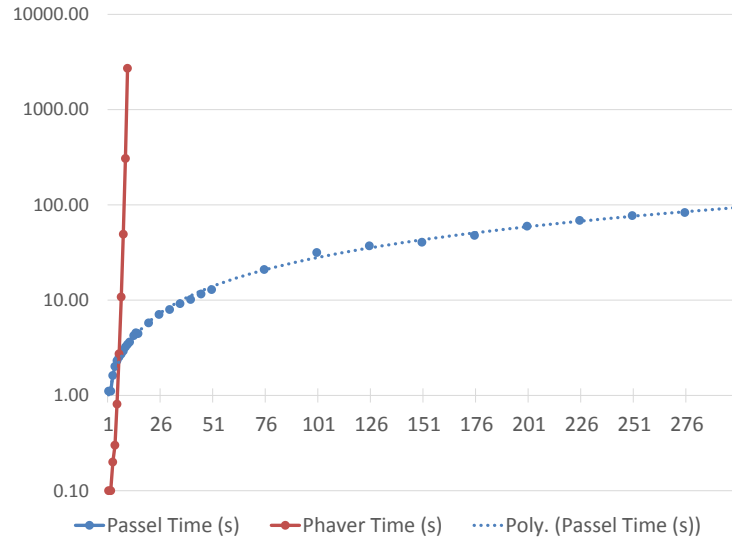


Fig. 11. Anonymized reachability runtime comparison of PHAVer and *Passel* for MUX-SEM. The vertical axis scale is logarithmic in seconds (s). The horizontal axis is the number of automata N . This illustrates scaling to hundreds of automata.

For $N = 11$, PHAVer runs out of memory, so comparisons beyond this value are not possible. As shown in Figure 8, for $N = 10$, PHAVer uses over 2.5 GB memory and completes in about 45 minutes, while *Passel* uses more than order of magnitude less memory at about 70 MB and nearly three orders of magnitude less runtime at about 3.5 seconds.⁸ Because of the anonymized representation of the state-space, *Passel* is able to compute the reachable states of $N = 30$ in under ten seconds (Figure 7) using about 70 MB memory (Figure 8). As shown in Figures 11 and 12, *Passel* is able to easily scale to hundreds of automata for MUX-SEM with modest runtime and memory usage.

⁸ Note that Z3 has some nondeterministic heuristics built-in that cause some of the memory fluctuations seen in the *Passel* results.

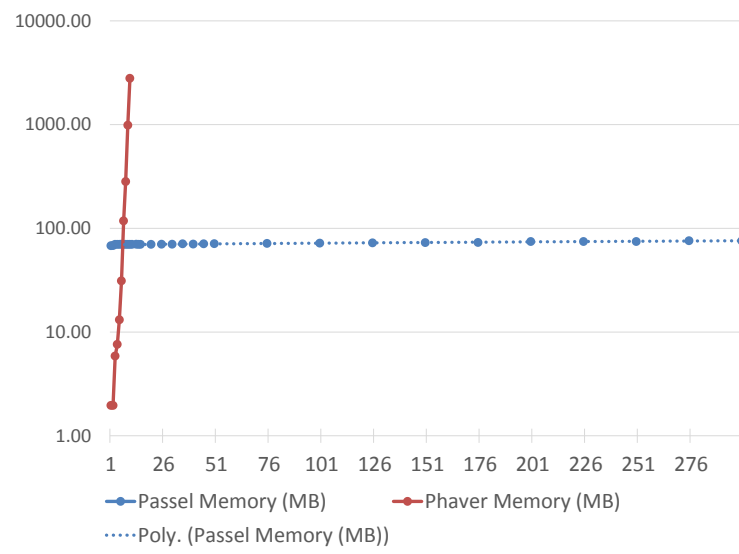


Fig. 12. Anonymized reachability memory usage comparison of PHAVer and *Passel* for MUX-SEM. The vertical axis scale is logarithmic in megabytes (MB). The horizontal axis is the number of automata N . This illustrates scaling to hundreds of automata.

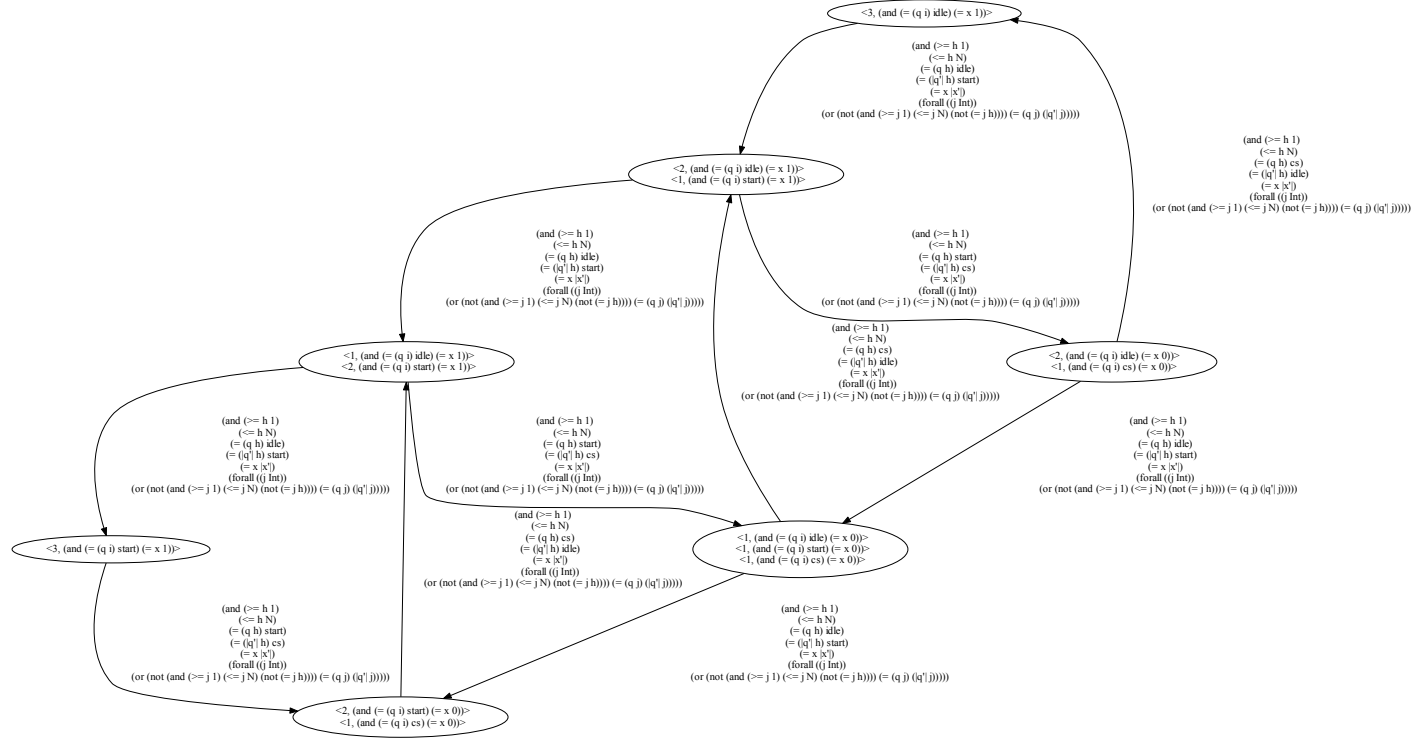


Fig. 13. Graphical view of the MUX-SEM mutual exclusion algorithm reachable states and transition graph for $N = 3$. This graphical representation is computed automatically by *Passel* after constructing the reachable states and transition graph on-the-fly using the method described in Section 4.

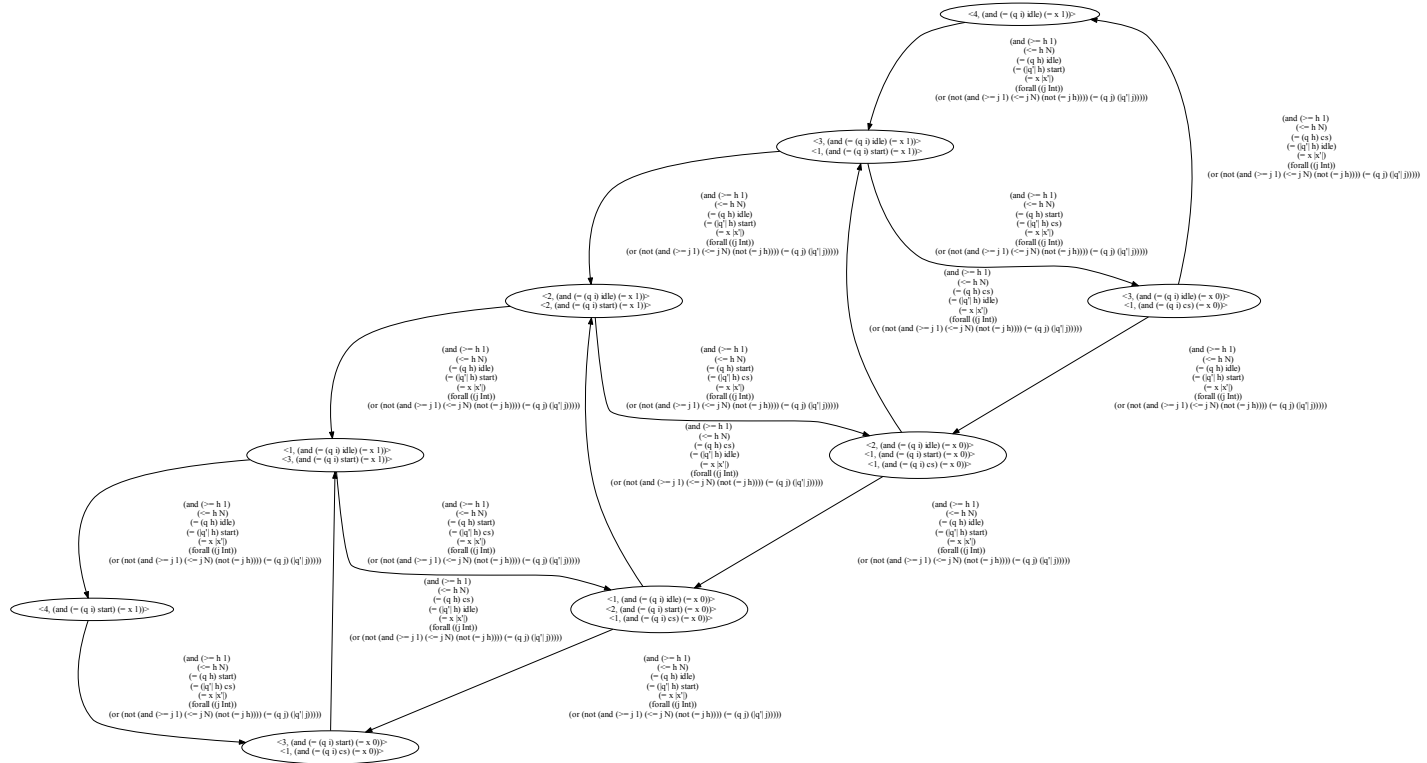


Fig. 14. Graphical view of the MUX-SEM mutual exclusion algorithm reachable states and transition graph for $N = 4$. This graphical representation is computed automatically by *Passel* after constructing the reachable states and transition graph on-the-fly using the method described in Section 4.

```

parameter name='lb' type='real' value = 1.0 // minimum rate
2 parameter name='ub' type='real' value = 2.0 // maximum rate
parameter name='B' type='real' value = 5.0 // guard constant
4
automaton name='MUX-INDEX-RECT'
6   variable name='q[i]' type='L' // location local variable
   variable name='x[i]' type='real' // continuous local variable
8   variable name='g' type='index' // global lock variable

10  location name='rem'
    frate: x[i]_dot >= lb and x[i]_dot <= ub
12  location name='try'
    frate: x[i]_dot >= lb and x[i]_dot <= ub
14  location name='cs'
    frate: x[i]_dot >= lb and x[i]_dot <= ub
16
    transition from='rem' to='try'
18    grd: g = 1 and x[i] >= B
    eff: g' = i and x[i]' = 0.0
20  transition from='try' to='cs'
    grd: g = i and x[i] >= 2*B
22    eff: x[i]' = 0
    transition from='cs' to='rem'
24    grd: x[i] >= 3*B
    eff: x[i]' = 0
26 property: forall i, j (i != j and q[i] = cs) implies (q[j] != cs)
initially: forall i (q[i] = rem and x[i] = 0 and g = 1)

```

Fig. 15. *Passel* input file specifying automaton template $\mathcal{A}(i)$ for mutual exclusion algorithm MUX-INDEX-RECT.

A.4 MUX-INDEX-RECT Mutual Exclusion Algorithm: Example with Anonymized Reachable State-Space Cardinality Independent of N

This section describes a simple timed mutual exclusion algorithm that has an anonymized reach set that is *independent of* N . The hybrid automaton template $\mathcal{A}(\mathcal{N}, i)$ specifying the MUX-INDEX-RECT example appears in Figure 15 and graphically in Figure 16. Specifically, the number of anonymized classes in *ReachForms* does not increase as a function of N . Additionally, for any $S \in \text{AnonReach}$, the sum of the class counts of S is 1, N , or $N-1$. This is in contrast to the MUX-SEM example described previously, which has some S with counts in $\{1, \dots, N\}$, so their runtimes and memory usages increase as a function of N . Due to this state-space size independence from N , our experiments have been successful for computing the reachable states for compositions of millions of automata.

For MUX-INDEX-RECT, PHAVer runs out of memory for $N = 8$. As shown in Figures 7 and 8, for $N = 7$, PHAVer uses over 1.3 GB memory and completes in over 3 hours, while *Passel* uses over an order of magnitude less memory at about 70 MB and nearly four orders of magnitude less runtime at about three seconds. Because of the anonymized representation of the state-space, *Passel* is able to compute the reachable states of $N = 30$ in a few seconds using about 70 MB memory, and *Passel* is able to easily scale to thousands of automata for MUX-INDEX-RECT. This is because the number of elements in the anonymized state-space representation does not grow as a function of N .

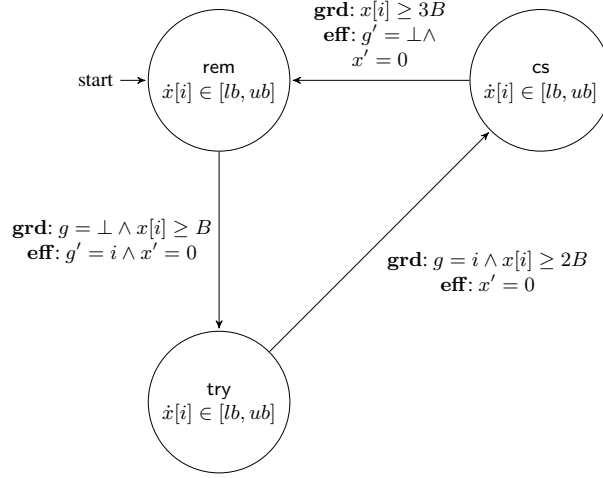


Fig. 16. MUX-INDEX-RECT mutual exclusion algorithm of Figure 15.

A.5 Nondeterministic Finite-State Automaton.

The next example specifies a simple nondeterministic finite-state automaton example with 5 states and 10 transitions. This artificial example is created purely to demonstrate the strength the anonymized state representation. For the NFA example, PHAVer is only able to compute the reachable states up to $N = 6$ before running out of memory due to its representation of all the permutations of reachable states. Even at $N = 6$, PHAVer uses about 600 MB memory and required over 3 minutes to compute the reachable states. While \mathcal{A}^6 only has $5^6 = 15625$ states, PHAVer utilizes an inefficient explicit-state representation. In comparison, the anonymized reachability method implemented in *Passel* computed the reachable states for the same example in an order of magnitude less time (about 20 seconds) and used about an order of magnitude less memory (about 75 MB). Furthermore, *Passel* was able to compute the set of reachable states up to $N = 30$ in a little over an hour, while using about 220 MB memory. The memory usage for the NFA example shown in Figure 8 illustrates the strength of *Passel*'s anonymized reachability method. While *Passel* initially uses much more memory than PHAVer—in part due to loading a variety of libraries, including the API to Z3—its scaling as a function of N is far superior. This is highlighted by *Passel* using about a third of the memory at $N = 30$ of 220 MB compared to PHAVer's usage of over 600 MB memory at $N = 6$, even though the problem size in terms of N is 5 times larger.

A.6 MUX-SEM-RA Mutual Exclusion

The MUX-SEM-RA mutual exclusion example shown in Figure 17 is just like the MUX-SEM purely discrete mutual exclusion example, except it includes a single continuous local variable for each automaton $\mathcal{A}(i)$. This example illustrates the additional memory and runtime requirements between a discrete and hybrid automaton, as the

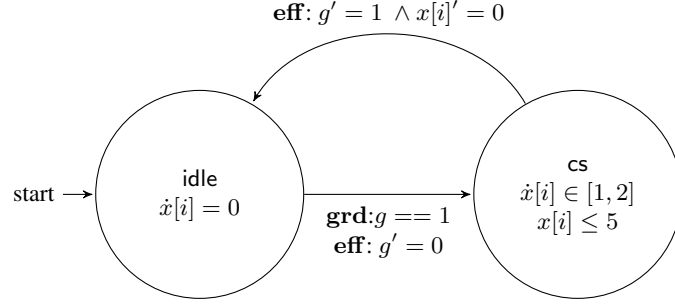


Fig. 17. MUX-SEM-RA mutual exclusion algorithm for automaton $\mathcal{A}(i)$ for illustrating the computation of continuous successors in the anonymized state-space representation.

formulas required to represent the continuous state variables are more complex. As we can see from Figure 7, at $N = 6$, PHAVer requires approximately an order of magnitude more runtime to compute the reachable states of MUX-SEM compared to MUX-SEM-RA, at slightly less than one second and around eight seconds, respectively. In comparison, for $N = 6$, *Passel* requires about four seconds to compute the reach set of MUX-SEM and around eight seconds to compute the reach set of MUX-SEM-RA, a growth of a factor of two, compared to PHAVer's order of magnitude growth. The memory comparison in Figure 8 is even better for *Passel*, even at $N = 30$ is using under 100 MB memory for MUX-SEM-RA, while PHAVer ran out of memory at $N = 9$, where it used about 700 MB memory.