A new Abstraction-Refinement based Verifier for Modular Linear Hybrid Automata and its Implementation

Hao Ren\(^1\) (ren@iastate.edu), Jing Huang\(^2\) (freescaler@live.com), Shengbing Jiang\(^3\) (shengbing.jiang@gm.com) and Ratnesh Kumar\(^1\), *Fellow, IEEE* (rkumar@iastate.edu)

\(^1\) Department of Electrical and Computer Engineering, Iowa State University, Ames, IA, 50014 USA
\(^2\) Technology Solutions Organization, Freescale Semiconductor, Austin, TX, 78735 USA
\(^3\) Vehicle Health Management Group, General Motors, Warren, MI, 48090 USA

Abstract—A concurrent linear hybrid automaton is composed of a set of linear hybrid automata and is used to model linear hybrid systems. Each component's behavior exhibits both discrete and continuous dynamics. We have developed LhaVrf, a symbolic verifier for the reachability verification of concurrent linear hybrid automata. The implementation is based on the algorithm proposed in S. Jiang’s papers. In his paper, reachability problem of linear hybrid automata was first reduced to one for linear transition systems, whose reachability analysis was then performed by using counterexample fragment based specification relaxation. S. Jiang proposed in another paper further enhancement in efficiency in context of concurrent systems, where for each counterexample fragment, a minimal conflicting constraints set was identified (that makes the fragment invalid), and used for specification relaxation. We adopted the above key ideas in the implementation of LhaVrf, with added features such as it automatically composes the concurrent subsystem models. Whenever the reachability to an unsafe state is satisfied, the output provides a concrete counterexample with values assigned to the variables. The LhaVrf is illustrated via an application to the Fischer mutual exclusion protocol.

I. INTRODUCTION

The reachability problem of the hybrid systems arises typically in the safety critical applications such as air traffic management system, aircraft and automobile control, medical equipment control, etc. We present a tool, LhaVrf, for the reachability verification of concurrent linear hybrid systems, modeled as concurrent linear hybrid automata. A concurrent linear hybrid automaton is a composition of individual linear hybrid automaton, which is a mathematical description of a hybrid system subject to certain linear constraints. A hybrid system includes both continuous and discrete dynamics, and in a linear hybrid system the flow dynamics, flow invariants, transition guards and jump conditions are all constrained by linear predicates.

Normally, the reachability analysis requires the computation of the set of reachable states, and in the presence of continuous dynamics, this is typically undecidable. However, semi-decision algorithms can be developed for hybrid automata by employing ordinary model checking over their finite-state abstractions, together with their iterative refinements based on invalid counterexamples [7]. Literature on correctness verification of hybrid systems is vast, and for a survey refer to [8], [9], [6], [10] and references therein.

[13] lists a number of tools for the formal verification of hybrid systems such as HyTech, CheckMate, PHAVer, etc. Our technique is based on [1] and [2] and is unique in translating a hybrid automaton into a transition system that preserves the discrete transition structure, possesses no continuous dynamics, and also preserves reachability of discrete states. The extension of the translation approach to general hybrid systems with inputs and outputs was later presented in [4].

After translating a hybrid automaton into a transition system, a finite state abstraction is then obtained by simply discarding all the transition guards. A counterexample for this abstraction may not be a counterexample of the concrete linear transition system, which can be easily checked by solving a linear programming problem as shown in [1]. The standard practice then is to use the entire invalid counterexample to refine the abstraction for eliminating the counterexample. Instead, [1] proposed a more efficient approach where a minimal invalid counterexample fragment (a subpath of the counterexample path) is identified and used to relax the specification to let it not reject the invalid fragment, thereby eliminate all paths that contain the fragment, including the counterexample path. (The relaxed specification is simply the disjunction of the current specification with an encoding of the counterexample fragment.) An advantage of specification relaxation (as opposed to standard abstract model refinement) is that there is no abstract model blow-up, and further the discrete transition structure is preserved so that the counterexample paths can always be intuitively interpreted.

In the setting of concurrent linear hybrid systems as in [2], the individual linear hybrid automata models are first composed, and next the composed linear hybrid automaton is abstracted, as in [1], into its linear transition system, preserving the discrete transition graph as well as the reachability (of discrete states), yet possessing no discrete dynamics. [2] then proposed another novel innovation to enhance efficiency: Compute for each invalid counterexample fragment its minimal conflicting constraints set, and use it to relax the specification so as to eliminate all paths that contain this set of conditions, including the counterexample path.

We developed a prototype implementation of the concur-
rent linear hybrid automaton verifier based on the approach [1] and [2] described above. $F^\#$ was used as the programming language. The symbolic model checker NuSMV was called inside the program for abstraction verification and counterexample generation. The counterexample or its fragment validity problem was converted into the linear programming problem and solved using the tools in Microsoft Solver Foundation. An incremental linear constraint solving algorithm like Cassowary [5] was used to compute the minimal conflict constraints set for inclusion into the relaxed satisfaction.

The rest of the paper is organized as follows. Section 2 gives some notations and preliminaries. Section 3 gives the implementation details of the tool that we developed. The Fischer mutual exclusion protocol is used as a running example to demonstrate the modeling, abstraction, safety verification and the use of the tool. The result of the tool’s application to it is provided in Section 4. Conclusions are given in Section 5.

II. NOTATION AND PRELIMINARY

A. Linear Hybrid Automaton (LHA)

Let $V = \{v_1, v_2, \ldots, v_n\}$ be a set of real-valued variables and $\vec{v} = (v_1, v_2, \ldots, v_n)$ be their vector representation. A convex linear predicate over $V$ is a finite boolean conjunction of linear inequalities over $V$. For a linear predicate $\phi$ and a valuation $\vec{a}$ over $\vec{v}$, we write $\phi[\vec{v} := \vec{a}]$ for the truth value obtained by evaluating $\phi$ with the constant $a_i$ replacing in $\phi$ all occurrences of variable $v_i$, for each $i \in \{1, \ldots, n\}$. Every linear predicate $\phi$ over $\vec{v}$ defines a set $[\phi] \subseteq \mathbb{R}^n$ of valuations such that $[\phi] = \{\vec{a} \in \mathbb{R}^n | \phi[\vec{v} := \vec{a}] = \text{true}\}$.

**Definition 1:** A linear hybrid automaton (LHA) is a tuple $A = (L, V, \Sigma, E, \text{Init}, \text{flow}, \text{inv}, \text{guard}, \text{jump})$ consisting of the following components:

- **Locations** $L$ is a finite set of locations.
- **Variables** $V = \{v_1, v_2, \ldots, v_n\}$ is a finite set of real-valued variables. $\dot{v}_i$ denotes the first derivative of $v_i$.
- **Events** $\Sigma$ is a finite set of events.
- **Discrete transitions** $E \subseteq L \times \Sigma \times L$ is the set of discrete transitions.
- **Initial state** $\text{Init} = (l_0, \vec{a}_0)$ is the initial state with $l_0 \in L$ and $\vec{a}_0 = (a_{01}, a_{02}, \ldots, a_{0n}) \in \mathbb{R}^n$.
- **Flow conditions** The flow function $\text{flow}$ assigns each location $l \in L$ a convex linear predicate $\text{flow}(l)$ over $\vec{v}$ that constrains the rates at which the values of variables change.
- **Invariant** The invariant function $\text{inv}$ assigns each location $l \in L$ a convex linear predicate $\text{inv}(l)$ over $\vec{v}$ that constrains the values of variables. It is assumed that $\vec{a}_0 \in \text{inv}(l_0)$.
- **Guard conditions** The guard function $\text{guard}$ assigns each transition $e = (l, \sigma, l') \in E$ a convex linear predicate $\text{guard}(e)$ over $\vec{v}$ such that the transition $e$ is enabled at a state $(l, \vec{a})$ only if $\vec{a} \in \text{guard}(e)$.
- **Jump conditions** The jump function $\text{jump}$ assigns each transition $e = (l, \sigma, l') \in E$ a convex linear predicate $\text{jump}(e)$ over $\vec{v}$ and $\vec{v}'$ such that if the transition $e$ is taken place from a state $(l, \vec{a})$ to a state $(l', \vec{a}')$, then $\vec{a}$ and $\vec{a}'$ must satisfy $\text{jump}(e)[\vec{v} := \vec{a}, \vec{v}' := \vec{a}']$.

A run in $A$ is a (finite or infinite) sequence $r = (l_0, \vec{a}_0) (l_1, \vec{a}_1) \ldots (l_i, \vec{a}_i) \ldots$ such that there exists a sequence of events $\sigma_0 \sigma_1 \ldots \sigma_i$ satisfying following properties:

- $\forall i \geq 0, (l_i, \sigma_i, l_{i+1}) \in E$.
- There exists a sequence of non-negative real numbers $t_0 \sigma_1 t_1 \ldots$ and a sequence of functions $\dot{x}_0, \dot{x}_1, \ldots$ such that $\forall i \geq 0, \dot{x}_i(t) \in \text{flow}(l_i) \land \dot{x}_i(t) \in \text{guard}(l_i) \land \forall i \geq 0, \text{jump}(l_i) \land \forall i \geq 0, \text{jump}(l_{i+1})$.

**Example 1:** The Fischer mutual exclusion protocol is used to guarantee mutual exclusion for shared resources in a concurrent system consisting of a number of processes. Each process $i$ is assumed to have a local clock modeled by the variable $v_i$. The shared variable $n$ is used to coordinate the access to the critical section. The LHA model $A^i$ for the $i$th process is shown in Figure 1. There are two positive real-valued parameters $D_1$ and $D_2$ in the model. $D_1$ represents the upper bound on the time that each process could take in changing the shared variable to its own number, and $D_2$ represents the lower bound on the time that each process must wait before it can check the variable value again.

![Fig. 1. LHA model for process_i in the Fischer protocol](image)

The LHA model for the $i$th process is given by, $A^i = (L^i, V^i, \Sigma^i, E^i, \text{Init}^i, \text{flow}^i, \text{inv}^i, \text{guard}^i, \text{jump}^i)$,

- $L^i = \{l^i, R^i, C^i, A^i\}$
- $V^i = \{v_i, n\}$
- $\Sigma^i = \emptyset$
- $E^i = \{\text{Init}^i, \text{flow}^i, \text{inv}^i, \text{guard}^i, \text{jump}^i\}$
- $\text{Init}^i = (l^i, (0, 0))$
- The predicates for invariants, flows, guards and jumps are clearly shown in Figure 1 (where a forward-slash is used to separate a guard from a jump on each edge).

B. Linear Transition System (LTS) and Translation of LHA into LTS

**Definition 2:** A linear transition system (LTS) is a tuple $S = (L, V, \Sigma, E, \text{Init}, P)$, where $L, V, \Sigma, E$ and $\text{Init}$ are
both sides of the inequality.

A run in $S$ is a finite (or infinite) sequence $r = (l_0, a_0)(l_1, a_1)\ldots(l_i, a_i)$ such that there exists a sequence of $\sigma_0, \sigma_1, \ldots, \sigma_i$, $\forall e \in (l, \sigma, l') \in E$ and $p(r(l_0, \sigma_0, l_1)) = true$. The transition trace associated with the run $r$ in an LHA $A$ (or an LTS $S$) is $(l_0, a_0)(l_1, a_1)\ldots(l_i, a_i, l_{i+1}) \ldots$. A location $l'$ is reachable in $A$ (or $S$) if there exists a run in $A$ (or $S$) with a transition trace $(l_0, a_0)(l_1, a_1, l_2)\ldots(l_i, a_i, l_{i+1})$. The set of transition traces associated with all finite-length runs in $A$ (or $S$) is called the language of $A$ (or $S$) and is denoted by $L(A)$ (or $L(S)$). $L(A)$ (or $L(S)$) is prefix closed.

In the following we discuss the translation from LHA to LTS, in which the vector $\vec{v}$ represents the variable values just after the system executes a transition and enters the current location, $\vec{u}$ represents the variable values just before the system executes a transition and enters the current location, and $t_{pre}$ represents the time that the system spends at the preceding location before entering the current location.

Given an LHA $A = (L, V, \Sigma, E, Init, flow, inv, guard, jump)$, we can construct the following LTS $S = (L, W, \Sigma, E, Init, P)$, which is called the LTS reduction of $A$, where

- $W = V \cup U \cup \{t_{pre}\}$ with $U = \{v_1, \ldots, v_n\}$,
- $Init = (l_0, \vec{0})$, with $\vec{0} = (0, \ldots, 0, a_0, \ldots, a_n)$,
- $\forall e = (l, \sigma, l') \in E$, $p(e) = flow(l)|\vec{v} - (\vec{u} - e) / t_{pre} \wedge inv(l)|\vec{v} - \vec{u} / guard(e)|\vec{v} - \vec{u} / jump(e)|\vec{v} - \vec{u} / t_{pre} ≥ 0$.

In order to handle the case of $t_{pre} = 0$, every inequality in $flow(l)|\vec{v} - (\vec{u} - e) / t_{pre}$ is modified by multiplying $t_{pre}$ at both sides of the inequality.

**Example 2:** The LTS model $S_A = (L^i, \Sigma^i, E^i, Init^i, P^i)$ derived from the LHA model $A^i$ for process $i$ in the Fischer protocol is shown in Figure 2, where

- $L^i = \{I^i, R^i, C^i, A^i\}$,
- $W = \{v^i, u^i, n^i, w^i, t_{pre}\}$,
- $\Sigma^i = \emptyset$,
- $E^i = \{(I^i, I^i), (I^i, R^i), (R^i, R^i), (R^i, C^i), (C^i, C^i), (C^i, I^i), (C^i, A^i), (A^i, A^i), (A^i, I^i)\}$,
- $Init^i_S = (l_0, 0, 0, 0, 0)$.

For each edge,

$p^i((I^i, R^i)) = (n^u = 0) \land (v^i = 0) \land (0 ≤ v^i ≤ D_i) \land (t_{pre} ≥ 0)$

$p^i((R^i, C^i)) = (u' - v = t_{pre}) \land (0 ≤ u' ≤ D_i) \land (n^u = i) \land (v^i = 0) \land (v ≥ 0) \land (t_{pre} ≥ 0)$

$p^i((C^i, A^i)) = (u' - v = t_{pre}) \land (0 ≤ u') \land (n^u = i) \land (u^i ≥ D_2) \land (t_{pre} ≥ 0)$

$p^i((A^i, I^i)) = (n^u = 0) \land (t_{pre} ≥ 0)$

$p^i((C^i, I^i)) = (u' - v = t_{pre}) \land (0 ≤ u') \land (n^u = i) \land (u^i ≥ D_2) \land (t_{pre} ≥ 0)$.

The following result was derived in [1].

**Theorem 1:** $L(A) = L(S_A)$.

Theorem 1 implies that the LTS preserves the behaviors over discrete states, and as a result it also preserves the reachability property. For the reachability analysis of LTS, we have the following algorithm from [1]. We need the following notions to describe the algorithm.

**Definition 3:** For an LTS $S$, a counterexample $ce = e_0\ldots e_n$ is a sequence of consecutive edges of the LTS. A counterexample fragment $f = e_i\ldots e_k$, with $0 ≤ i < k ≤ n$, is a consecutive subsequence of edges of $ce$. $f$ is invalid if and only if $L^k_{e_i} = p(e_j)$ is not satisfiable. $f$ is a minimal invalid fragment if no invalid fragment of smaller length exists. For an invalid fragment $f$, a specification is relaxed by disjuncting it with the formula $¬Bad U (\wedge_{i=0}^{k-1} X^i e_{i+1})$ that holds true until the consecutive sequence of edges of the fragment is encountered (where $Bad$ is an atomic proposition that holds at unsafe states, ”$U$” denotes the "until" operator, and "$X^i"$ denotes the "next" operator).

**Algorithm 1:** Reachability analysis of LTS

1) Construct the initial abstract model by discarding all the transition guards in the LTS, and set the initial specification as: $G¬Bad$, where "$G$" denotes the "globally" operator.

2) Model check the abstract model against the specification. If the specification is satisfied, then terminate with the output of "The system satisfies the specification." Otherwise, go to next step with the counterexample provided by the model checker.

3) Validate the counterexample. If the counterexample is valid in the original LTS, then terminate and output, "The system does not satisfy the specification.", along with the concrete counterexample obtained during the validation. Otherwise go to next step.

4) Identify a fragment (a minimal invalid subpath of the counterexample path) of the counterexample.

5) Relax the specification (see Definition 3) so all paths containing the counterexample fragment no longer violate the specification, and go back to step 2.

**C. Concurrent LHA**

The definition of the composition of LHAs given below is a refined version of the one in [2] to make it more compact.

**Definition 4:** The synchronous composition of $A^i = (L^i, V^i, \Sigma^i, E^i, Init^i, flow^i, inv^i, guard^i, jump^i)$, $i =
1, \ldots, k \) is given by, \( A = \bigoplus_{i=1}^{k} A^i \models (L, V, \Sigma, E, \text{Init}, \text{flow}, \text{inv}, \text{guard}, \text{jump}) \), where

- \( L := \times_{i=1}^{k} L^i \),
- \( V := \times_{i=1}^{k} V^i \),
- \( \Sigma := \times_{i=1}^{k} \Sigma^i \),
- \( E := \{ e | e = (l^1, \ldots, l^k), (\sigma^1, \ldots, \sigma^k), (l^1', \ldots, l^k') \in E; i = 1, \ldots, k \} \), where \( E = E^i \cup \{ (l^i, e, l^i') | l^i \in L^i \} \),
- \( \text{Init} := ((l^0_1, \ldots, l^0_k), \vec{a}_0) \), where \( \vec{a}_0 \) is the initial assignment over \( V \), and \( \text{Init}^i = (l^0_i, a^0_i) \). It is assumed that \( a_0 \in [\text{inv}(l_0)] \).

∀ \in (l^1, \ldots, l^k) \in L, \text{inv}(l) := \bigwedge_{i=1}^{k} \text{inv}^i(l^i).\]

∀ \in (l^1, \ldots, l^k) \in L, let \( f(l, v_j) = \{ i | \text{flow}^i(l^i(v_j)) \neq \text{null} \} \), then \( \text{flow}(l) := \bigwedge_{v_j \in V} f(l, v_j) \), where

\[ f(l, v_j) = \left\{ \begin{array}{ll}
\lambda_i \in E(l, v_j) \text{flow}^i(l^i(v_j)) & F(l, v_j) \neq \emptyset \\
\left( \vec{v}_j = 0 \right) & F(l, v_j) = \emptyset
\end{array} \right. \]

∀ \in (l^1, \ldots, l^k), (\sigma^1, \ldots, \sigma^k), (l^1', \ldots, l^k') \in E, \quad \text{guard}(e) := \bigwedge_{i=1}^{k} \text{guard}^i(e)^i \land \text{syn}(e), \quad \text{where} \quad \text{syn}(e) : \quad E \to \{0, 1\} \quad \text{is used to capture additional user-specified enabling constraints such as at most one subsystem can make a transition in a system transition.}

∀ \in (l^1, \ldots, l^k), (\sigma^1, \ldots, \sigma^k), (l^1', \ldots, l^k') \in E, \quad \text{let} \quad J(e, v_j) = \{ i | \text{jump}^i(e(v_j)) \neq \text{null} \}, \quad \text{then} \quad \text{jump}(e) := \bigwedge_{v_j \in V} \text{jump}(e(v_j)) \land \bigwedge_{v_j \in V} e(v_j) \neq \emptyset.

D. Reachability analysis of concurrent LHA

Once the concurrent LHA model \( A = \bigoplus_{i=1}^{k} A^i \) is obtained, we compute its corresponding LTS \( S_A = (L, W, \Sigma, E, \text{Init}, P) \) as described above, and apply Algorithm 1 for reachability analysis, with the steps 4 and 5 modified as below.

4) Identify a fragment of the counterexample. For the counterexample fragment identified, determine the minimal conflicting constraints set (MCCS) that itself is invalid, where a MCCS is a minimal invalid subset of the set of constraints, that are associated with the counterexample fragment. Recall that for a fragment \( f = e_1 \ldots e_k \), the associated set of constraints consists of the conjunct \( \bigwedge_{j=m}^{k} \text{pref}(e_j) \).

5) Relax the specification by disjuncting it with the encoding of the MCCS (by replacing \( e_{i+j} \) in the formula \( \lnot \text{Bad} U \langle \lambda_j \gamma^k \rangle \text{Bad} j \)) introduced in Definition 3 with \( \bigwedge_{k \in K} j \bigwedge_{k'} \text{pref}(e_{i+j}) \), where \( K_{i+j} \) is the index set of subsystems that participate in the \( (i+j) \)th step of the fragment and contribute to the MCCS), so all paths containing the MCCS no longer violate the specification, and go back to step 2.

III. IMPLEMENTATION OF LHAVRF

A. Architecture

The tool LHAVRF is implemented in the programming language F#. The architecture consists of six modules, and the data flow among them occurs along the arrows, as shown in Figure 3. Each of the modules is introduced separately in the following subsections.

B. Input processing

The Input processing module accepts a series of txt files as its inputs. Each txt file MDL4LHA_i.txt describes a component LHA \( A^i \) with the following syntax:

1) "l e l'" is used to denote a transition relation \((l, e, l')\).
2) "s : constraint" is used to denote a predicate at \( s \).

If \( s \) represents a location, then \( \text{constraint} \) is \( \text{flow} \) or \( \text{inv} \) for that location; if \( s \) represents an edge, then \( \text{constraint} \) is \( \text{guard} \) or \( \text{jump} \) for that edge. In \( \text{inv} \) and \( \text{guard} \), the variable is represented by \( v_i \); in \( \text{flow} \) and \( \text{jump} \) the variable is represented by \( v_j \).

3) The rest of the lines contain:
   a) "l" is used to designate the initial location \( l \).
   b) "!!v_i a_i." is used to designate the initial value \( v_i = a_i \).
   c) "@v_1 \ldots v_j ...." is used to enlist all the variables.

A separate line is added to specify the set of unsafe locations of the concurrent LHA with the format:

\[ \# \text{bad}_1, \ldots, \text{bad}_n \]; where \( \text{bad}_i \) is a location of the concurrent LHA.

The Input processing module translates the syntactical lines in each input file via lexing and parsing and passes the data containing the description information of the LHA to the Data storage module.

C. Data storage

In the Data storage module, each location or transition is assigned a unique id composed by its subsystem index and its serial number in the subsystem. \( F\# \) provides a dictionary data type, in which each predicate is coupled with the location or transition’s id. The unsafe locations are also stored here coupled with their own ids.

D. Model building

Model building module collects all the data from the Data storage module and computes the LTS model of the concurrent LHA. For every possible concurrent edge, the module collects the relevant predicates in the concurrent LHA model and converts them into the edge-predicate for the
LTS model. Then, the Model building module calls the Path analyzing module for validating the edge, and only when the edge is valid, it adds the edge to the LTS model.

Model building module also collects the unsafe locations and encodes them into the initial LTL specification \(G \rightarrow \text{Bad} \). Then it passes the abstract model and the LTL specification to NuSMV as LTS2SMV.txt.

E. NuSMV

NuSMV module is a well known symbolic model checker. Once called by the Model building module, it checks the abstract model in the LTS2SMV.txt file against its specification. If it is satisfied, the entire program terminates with the output stating that the system is safe. Otherwise, a counterexample is generated and written into SMV2LTS.txt and passed on to the Path analyzing module.

F. Path analyzing (LP solver)

Path analyzing module accepts edge guards from the Model building module and returns whether or not those are satisfiable. It also accepts a counterexample from the NuSMV module, and gathers the predicates of the counterexample edges from the Data storage module. The LP solver inside the module, that utilizes the tools in the Microsoft Solver Foundation, a .NET solution for mathematical optimization and modeling, accepts the predicate constraints, solves for satisfiability, and returns \textit{valid} (and the valid assignments to the variables) or \textit{invalid}. If the counterexample from SMV2LTS.txt is found valid, the entire program terminates with the output stating that the system is unsafe and reports the concrete counterexample and the valid assignments. Otherwise a fragment and its MCCS are generated using an iterative linear constraint solving algorithm, working in a "bottom-up" fashion (details can be found in [2]). The resulting MCCS is sent to the Formula relaxing module.

G. Formula refining

Given a MCCS received from the Path analyzing module, the Formula refining module relaxes the current formula by disjuncting it with the encoding of the MCCS. The relaxed formula is then sent to the Model building module to start a new round of model checking.

IV. AN ILLUSTRATIVE EXAMPLE

In the Fischer protocol case, assume there are two processes, and that at most one process can make a location transition at any given time. A snapshot of the input file MDL4LHA_i.txt for \( \text{proc}^i \) (\( i = 1, 2 \)) is shown in Figure 4, in which the locations (1-tuples) and edges (2-tuples): \( I^1, (I^1, R^1), R^2, (R^2, C^0), C^1, (C^1, A^0), A^1, (A^1, I^1), (A^1, I^2) \) are numbered as 1, 2, 3, 4, 5, 6, 7, 8, 9 respectively.

The LhaVrf first reads the model-input files then automatically translates each input file into its LHA model \( A^i \) and stores it in the Data storage module. Next, all the LHA \( A^i \) are composed to form the concurrent LHA \( A^t \), and converted to the LTS \( S \), from which the abstract model as shown in Figure 5 is extracted.

For the specification, \( \text{proc}^1 \) and \( \text{proc}^2 \) are not allowed to be in the access state at the same time. More precisely, we have the following LTL specification: "\( G \rightarrow \text{Bad} \)" where \( \text{Bad} = A_1 \land A_2 \), representing that both \( \text{proc}^1 \) and \( \text{proc}^2 \) are in the access state at the same time.

Note in the Fischer protocol case, any edge can be identified by a unique pair of locations. Therefore, we can express each counterexample/fragment as a sequence of locations instead of edges for simplicity. When \( D_1 = 4, D_2 = 3 \), NuSMV generates a counterexample \( \text{ce} = (I^1, I^2) (R^1, I^2)(R^2, I^2)(R^1, C^0)(R^1, A^2)(C^1, A^2)(A^1, A^2) \). It is verified as a concrete counterexample, which means when \( D_1 = 4, D_2 = 3 \), the mutual exclusion is not guaranteed.

On the other hand when \( D_1 = 2, D_2 = 3 \), NuSMV first generates a counterexample \( \text{ce} = (I^1, I^2)(R^1, I^2)(C^1, I^2) (C^1, R^2)(C^1, C^2)(C^1, A^2)(A^1, A^2) \). This counterexample is invalid. Then the counterexample fragment is computed as \( f = (R^1, I^2)(C^1, I^2)(C^1, R^2) \). In the implementation, each variable is tagged with its time step to denote its value at that particular time step. For instance, \( n^3_5 \) is the value of \( n^3 \) at the 3rd step. Corresponding to \( f \), a minimal conflicting constraints set is identified as follows:

\[
\begin{align*}
(n^3_5 &= 1) & \text{from } \text{jump}^3((R^1, C^1)) \\
(n^4_3 &= 0) & \text{from } \text{guard}^4((I^2, R^2)) \\
(n^4_4 &= n^5_5) & \text{from } \text{flow}^4(C^1) \text{ and } \text{flow}^5(I^2)
\end{align*}
\]

The corresponding subsystem transitions are found to be
\((R^1_3, C^1_3)(I^2_3, R^2_3)\). They are encoded to the formula as:

\[ f_{\text{MCCS}} = \neg\text{Bad} U ((R^1 \times C^1) \times (I^2 \times R^2)) \]

The verifier repeats the execution loop as above for twelve iterations, finally showing that the bad state is not reachable.

The invalidity of the counterexample shown above can be attributed to the variable \(n\), and the same holds for most of the other counterexamples we obtained. By observation, we know that \(n\) is a discrete finite valued variable with its value updated only on transitions. Thus, instead of treating \(n\) as a flow variable, we can treat it as a discrete state, with a finite set of locations ranging over the values \(n\) can take, with its transitions guarded by constraints that represent the edges in the LHA \(A\) that update \(n\). Such a model of \(n\) for the two-process case is shown in Figure 6.

For example, one guard condition in Figure 6 is:

\[ \sigma(0, 1) = ((R^1, I^2, (C^1, I^2)) \lor ((R^1, R^2, (C^1, R^2))) \lor ((R^1, C^2, (C^1, C^2)) \lor ((R^1, A^2, (C^1, A^2))) \]

Then the predicates over \(n\) can be discarded from each \(A^i\). The concurrent model then is obtained by composing \(\tilde{A}_i, i = 1, \ldots, k\), and the model for \(n\). Figure 7 shows the corresponding abstract model for the two-process case. A counterexample generated from this new model is given in [2] as: \((I^1, I^2, 0)(I^1, R^2, 0)(R^1, R^2, 0)(R^1, C^2, 2)(R^1, A^2, 2)(C^1, A^2, 1)(A^1, A^2, 1)\).

Since \(n\) is no longer a flow variable, all the counterexamples related to the discrete variable \(n\) are eliminated. Hence the number of iterations decreases dramatically, and the length of the relaxed LTL specification reduces as a consequence. As a trade-off, the size of the abstract model gets larger (compare Figure 6 versus Figure 4). Nevertheless, the overall efficiency of the algorithm is much enhanced.

Remark 1: If a concurrent linear hybrid system contains a variable that has finite domain, so it gets updated only along transitions (and not within the locations), then we can model the behavior of the variable as another transition system, and compose it with the other subsystems. This corresponds to an "in-built" refinement of the model that automatically removes many of the invalid counterexample paths of the original model, and expedites the verification process.

V. Conclusion

We presented the implementation of LhaVrf, a symbolic verifier for the reachability verification of concurrent linear hybrid automata. It employs the concepts of counterexample fragment and minimal conflicting constraints set based specification relaxation algorithm from [1] and [2]. LhaVrf accepts a set of input files containing the constituent linear hybrid automata written in an easy-to-specify textual format. Upon termination, the verifier outputs a concrete counterexample when the specification of avoiding reaching the unsafe states is violated, or reports that safety is not violated. Since the reachability problem for linear hybrid systems is undecidable, there is no apriori guarantee of termination (as is the case with any hybrid system verifier).

REFERENCES