Software Engineering Group
Department of Computer Science
Nanjing University
http://seg.nju.edu.cn

Technical Report No. NJU-SEG-2011- IC-003

IIS-Guided DFS for Efficient Bounded Reachability Analysis of Linear Hybrid Automata

Lei Bu, Yang Yang, Xuandong Li

Postprint Version. Originally Published in:
Haifa Verification Conference 2011
LNCS 7261, pp. 35–49, 2011

IIS-Guided DFS for Efficient Bounded Reachability Analysis of Linear Hybrid Automata

Lei Bu, Yang Yang, and Xuandong Li

State Key Laboratory for Novel Software Technology,
Nanjing University, Nanjing, Jiangsu, P.R. China 210093
{bulei,lxd}@nju.edu.cn, yangyang@seg.nju.edu.cn

Abstract. In the authors' previous work, we proposed a linear programming (LP) based approach to check the reachability specification along one abstract path in a linear hybrid automaton (LHA) at a time by translating the reachability problem into the satisfiability problem of a linear constraint set. Then a depth-first-search (DFS) is deployed on the graph structure of the LHA to check all the paths with length in the threshold to answer the question of bounded reachability.

In this DFS-style bounded model checking (BMC) algorithm, once a path is found to be infeasible by the underlying LP solver, a backtracking on the graph structure will be conducted. Clearly, the efficiency of the algorithm depends on the accuracy of the backtracking. If the DFS can backtrack to the most reasonable location, the state space need to search and verify can be reduced significantly.

Fortunately, once a linear constraint set is judged to be unsatisfiable, the irreducible infeasible set (IIS) technique can be deployed on the unsatisfiable constraint set to give a quick analysis and find a small set of constraints which makes the whole program unsatisfiable. In this paper, we adopt this technique into our DFS-style BMC of LHA to locate the nodes and transitions which make the path under verification infeasible to guide the backtracking and answer the bounded reachability of LHA more efficiently.

1 Introduction

Hybrid automata [1] are well studied formal models for hybrid systems with both discrete and continuous state changes. However, the analysis of hybrid automata is very difficult. Even for the simple class of *linear hybrid automata (LHA)*, the reachability problem is undecidable [1–4]. The state-of-the-art symbolic model checking techniques for LHA try to compute the transitive closure of the state space of the system by geometric computation which is very expensive and not guaranteed to terminate. Several tools are designed and implemented in this style, like HYTECH [10] and its improvement PHAVer [11] but they do not scale well to the size of practical problems.

In recent years, bounded model checking (BMC) [5] has been presented as an alternative technique for BDD-based symbolic model checking, whose basic idea is to encode the next-state relation of a system as a propositional formula, and unroll this formula to some integer k, using SAT/SMT idea to search for a counterexample in the model executions whose length is bounded by k. These technique have been used to answer the reachability problem of LHA also. But, as these techniques require to encode the state space of LHA in threshold firstly, when the system size or the given step

K. Eder, J. Lourenço, and O. Shehory (Eds.): HVC 2011, LNCS 7261, pp. 35–49, 2012.

threshold is large, the object problem could be very huge, which greatly restricts the size of the problem that can be solved [6, 7].

Both symbolic model checking and bounded model checking are facing the complete state space or the partly complete space under the threshold at one time which is always too large and complex for the solver to handle. In order to control the complexity of the verification of LHA, we proposed a linear programming (LP) based approach[8] to develop an efficient path-oriented reachability checker to check one abstract path in the graph structure of a LHA at a time to find whether there exists a behavior of the LHA along this abstract path and satisfy the given reachability specification. In such a manner, the length of the path and the size of the automaton being checked can be large enough to handle problems of practical interest. As a straightforward extension, all the abstract paths with length shorter than or equal to the threshold in the graph structure can be enumerated and checked one by one by depth-first-search(DFS) traversing to answer the question of bounded reachability analysis of the LHA.

The above DFS based BMC has shown good performance and scalability in our previous studies[9, 14]. Nevertheless, it has a lot of space to optimize:

- The simple DFS algorithm checks each path ρ in the given length threshold for the reachability by solving the corresponding linear program. Although the checking of a single path is very efficient, if the number of candidate path is large, it will still be time consuming. However, suppose we are checking whether location v is reachable in bound k, and v is not contained in ρ at all, we can simply falsify ρ for the reachability to save computation time.
- Once a path ρ is found to be infeasible, the DFS algorithm will only remove the last location in the path and backtrack to the location preceding the last one to search for the next candidate in a recursive manner. This backtracking method does not use any information of the infeasible path. If the infeasible cone in the linear constraint set related to ρ can be extracted, then the DFS procedure can backtrack to the exact place that makes the ongoing path infeasible. Then, the state space needed to search and verify can be pruned significantly.

Based on the above directions, we optimize our DFS-style BMC algorithm in the following ways:

- Only when the last location of the current visiting path ρ is contained in the reachability specification, the DFS procedure will call the underlying decision procedure to check the feasibility of ρ. Otherwise, the DFS will just go on traversing on the graph structure to reduce the time overhead.
- Once a linear constraint set is judged to be unsatisfiable, the irreducible infeasible set (IIS) technique[12] can be deployed to give quick analysis of the program and find a small set of constraints which makes the whole program unsatisfiable. We deploy this technique into our DFS-style BMC of LHA to locate the nodes and transitions which cause the path under verification infeasible to guide the backtracking and answer the bounded reachability of LHA more efficiently.

2 Linear Hybrid Automata and Reachability Verification

This section gives the formal definition of linear hybrid automata and presents the review of the path-oriented reachability analysis and bounded reachability analysis techniques that were proposed in our previous works[8, 9].

2.1 Linear Hybrid Automata

The linear hybrid automata (LHA) considered in this paper are defined in[13], which is a variation of the definition given in [1]. The flow conditions of variables in a linear hybrid automaton considered here can be given as a range of values for their derivatives.

Definition 1. An LHA H is a tuple $H = (X, \Sigma, V, V^0, E, \alpha, \beta, \gamma)$, where

- X is a finite set of real-valued variables; Σ is a finite set of event labels; V is a finite set of *locations*; $V^0 \subseteq V$ is a set of *initial locations*.
- E is a *transition relation* whose elements are of the form $(v, \sigma, \phi, \psi, v')$, where v, v' are in V, $\sigma \in \Sigma$ is a label, ϕ is a set of *transition guards* of the form $a \leq \sum_{i=0}^{l} c_i x_i \leq b$, and ψ is a set of *reset actions* of the form $x := c$ where $x_i \in X$, $x \in X$, a, b, c and c_i are real numbers (a, b may be ∞).
- α is a labeling function which maps each location in V to a *location invariant* which is a set of *variable constraints* of the form $a \leq \sum_{i=0}^{l} c_i x_i \leq b$ where $x_i \in X$, a, b and c_i are real numbers (a, b may be ∞).
- β is a labeling function which maps each location in V to a set of *flow conditions* which are of the form $\dot{x} \in [a, b]$ where $x \in X$, and a, b are real numbers ($a \leq b$). For any $v \in V$, for any $x \in X$, there is one and only one flow condition $\dot{x} \in [a, b] \in \beta(v)$.
- γ is a labeling function which maps each location in V^0 to a set of *initial conditions* which are of the form $x = a$ where $x \in X$ and a is a real number. For any $v \in V^0$, for any $x \in X$, there is at most one initial condition definition $x = a \in \gamma(v)$. □

Path and Behavior. We use the sequences of locations to represent the evolution of an LHA from location to location. For an LHA $H = (X, \Sigma, V, V^0, E, \alpha, \beta, \gamma)$, a *path segment* is a sequence of locations of the form $\langle v_0 \rangle \xrightarrow[\sigma_0]{(\phi_0,\psi_0)} \langle v_1 \rangle \xrightarrow[\sigma_1]{(\phi_1,\psi_1)} \cdots \xrightarrow[\sigma_{n-1}]{(\phi_{n-1},\psi_{n-1})} \langle v_n \rangle$, which satisfies $(v_i, \sigma_i, \phi_i, \psi_i, v_{i+1}) \in E$ for each i ($0 \leq i < n$). A *path* in H is a path segment starting at an initial location in V^0.

For a path in H of the form $\langle v_0 \rangle \xrightarrow[\sigma_0]{(\phi_0,\psi_0)} \langle v_1 \rangle \xrightarrow[\sigma_1]{(\phi_1,\psi_1)} \cdots \xrightarrow[\sigma_{n-1}]{(\phi_{n-1},\psi_{n-1})} \langle v_n \rangle$, by assigning each location v_i with a time delay stamp δ_i we get a *timed sequence* of the form $\begin{pmatrix} v_0 \\ \delta_0 \end{pmatrix} \xrightarrow[\sigma_0]{(\phi_0,\psi_0)} \begin{pmatrix} v_1 \\ \delta_1 \end{pmatrix} \xrightarrow[\sigma_1]{(\phi_1,\psi_1)} \cdots \xrightarrow[\sigma_{n-1}]{(\phi_{n-1},\psi_{n-1})} \begin{pmatrix} v_n \\ \delta_n \end{pmatrix}$ where δ_i ($0 \leq i \leq n$) is a nonnegative real number, which represents a behavior of H such that the system starts at v_0, stays there for δ_0 time units, then jumps to v_1 and stays at v_1 for δ_1 time units, and so on.

The behavior of an LHA can be described informally as follows. The automaton starts at one of the initial locations with some variables initialized to their initial values. As time progresses, the values of all variables change continuously according to the

flow condition associated with the current location. At any time, the system can change its current location from v to v' provided that there is a transition $(v, \sigma, \phi, \psi, v')$ from v to v' whose all transition guards in ϕ are satisfied by the current value of the variables. With a location change by a transition $(v, \sigma, \phi, \psi, v')$, some variables are reset to the new value accordingly to the reset actions in ψ. Transitions are assumed to be instantaneous.

Let $H = (X, \Sigma, V, V^0, E, \alpha, \beta, \gamma)$ be an LHA. Given a timed sequence ω of the form $\left\langle \begin{matrix} v_0 \\ \delta_0 \end{matrix} \right\rangle \xrightarrow[\sigma_0]{(\phi_0,\psi_0)} \left\langle \begin{matrix} v_1 \\ \delta_1 \end{matrix} \right\rangle \xrightarrow[\sigma_1]{(\phi_1,\psi_1)} \cdots \xrightarrow[\sigma_{n-1}]{(\phi_{n-1},\psi_{n-1})} \left\langle \begin{matrix} v_n \\ \delta_n \end{matrix} \right\rangle$, let $\zeta_i(x)$ represents the value of x ($x \in X$) when the automaton has stayed at v_i for delay δ_i along with ω ($0 \le i \le n$), and $\lambda_i(x)$ represents the value of x at the time the automaton reaches v_i along with ω. It follows that $\lambda_0(x) = a$ if $x = a \in \gamma(v_0)$, and $\lambda_{i+1}(x) = \begin{cases} d & \text{if } x := d \in \psi_i \\ \zeta_i(x) & \text{otherwise} \end{cases}$ $(0 \le i < n)$.

Definition 2. For an LHA $H = (X, \Sigma, V, V^0, E, \alpha, \beta, \gamma)$, a timed sequence of the form $\left\langle \begin{matrix} v_0 \\ \delta_0 \end{matrix} \right\rangle \xrightarrow[\sigma_0]{(\phi_0,\psi_0)} \left\langle \begin{matrix} v_1 \\ \delta_1 \end{matrix} \right\rangle \xrightarrow[\sigma_1]{(\phi_1,\psi_1)} \cdots \xrightarrow[\sigma_{n-1}]{(\phi_{n-1},\psi_{n-1})} \left\langle \begin{matrix} v_n \\ \delta_n \end{matrix} \right\rangle$ represents a behavior of H if and only if the following condition is satisfied:

- $\langle v_0 \rangle \xrightarrow[\sigma_0]{(\phi_0,\psi_0)} \langle v_1 \rangle \xrightarrow[\sigma_1]{(\phi_1,\psi_1)} \cdots \xrightarrow[\sigma_{n-1}]{(\phi_{n-1},\psi_{n-1})} \langle v_n \rangle$ is a path;
- $\delta_1, \delta_2, \ldots, \delta_n$ ensure that each variable $x \in X$ evolves according to its flow condition in each location v_i $(0 \le i \le n)$, i.e. $u_i\delta_i \le \zeta_i(x) - \lambda_i(x) \le u_i'\delta_i$ where $\dot{x} \in [u_i, u_i'] \in \beta(v_i)$;
- all the transition guards in ϕ_i $(1 \le i \le n-1)$ are satisfied, i.e. for each transition guard $a \le c_0x_0 + c_1x_1 + \cdots + c_lx_l \le b$ in ϕ_i, $a \le c_0\zeta_i(x_0) + c_1\zeta_i(x_1) + \cdots + c_l\zeta_i(x_l) \le b$;
- the location invariant of each location v_i $(1 \le i \le n)$ is satisfied, i.e.
  - at the time the automaton leaves v_i, each variable constraint $a \le c_0x_0 + c_1x_1 + \cdots + c_lx_l \le b$ in $\alpha(v_i)$ $(0 \le i \le n)$ is satisfied, i.e.$a \le c_0\zeta_i(x_0) + c_1\zeta_i(x_1) + \cdots + c_l\zeta_i(x_l) \le b$, and
  - at the time the automaton reaches v_i, each variable constraint $a \le c_0x_0 + c_1x_1 + \cdots + c_lx_l \le b$ in $\alpha(v_i)$ $(0 \le i \le n)$ is satisfied, i.e.$a \le c_0\lambda_i(x_0) + c_1\lambda_i(x_1) + \cdots + c_l\lambda_i(x_l) \le b$. □

Definition 3. For an LHA $H = (X, \Sigma, V, V^0, E, \alpha, \beta, \gamma)$, if a timed sequence of the form $\left\langle \begin{matrix} v_0 \\ \delta_0 \end{matrix} \right\rangle \xrightarrow[\sigma_0]{(\phi_0,\psi_0)} \left\langle \begin{matrix} v_1 \\ \delta_1 \end{matrix} \right\rangle \xrightarrow[\sigma_1]{(\phi_1,\psi_1)} \cdots \xrightarrow[\sigma_{n-1}]{(\phi_{n-1},\psi_{n-1})} \left\langle \begin{matrix} v_n \\ \delta_n \end{matrix} \right\rangle$ is a behavior of H, we say path $\rho = \langle v_0 \rangle \xrightarrow[\sigma_0]{(\phi_0,\psi_0)} \langle v_1 \rangle \xrightarrow[\sigma_1]{(\phi_1,\psi_1)} \cdots \xrightarrow[\sigma_{n-1}]{(\phi_{n-1},\psi_{n-1})} \langle v_n \rangle$ is *feasible*, and location v_n is *reachable* along ρ. □

2.2 Reachability Specification and Verification

Reachability Specification. For an LHA $H = (X, \Sigma, V, V^0, E, \alpha, \beta, \gamma)$, a *reachability specification*, denoted as $\mathcal{R}(v, \varphi)$, consists of a location v in H and a set φ of variable constraints of the form $a \le c_0x_0 + c_1x_1 + \cdots + c_lx_l \le b$ where $x_i \in X$ for any i $(0 \le i \le l)$, a, b and c_i $(0 \le i \le l)$ are real numbers.

Definition 4. Let $H = (X, \Sigma, V, V^0, E, \alpha, \beta, \gamma)$ be an LHA, and $\mathcal{R}(v, \varphi)$ be a reachability specification. A behavior of H of the form $\left\langle \begin{matrix} v_0 \\ \delta_0 \end{matrix} \right\rangle \xrightarrow[\sigma_0]{(\phi_0,\psi_0)} \left\langle \begin{matrix} v_1 \\ \delta_1 \end{matrix} \right\rangle \xrightarrow[\sigma_1]{(\phi_1,\psi_1)} \cdots \xrightarrow[\sigma_{n-1}]{(\phi_{n-1},\psi_{n-1})} \left\langle \begin{matrix} v_n \\ \delta_n \end{matrix} \right\rangle$ *satisfies* $\mathcal{R}(v, \varphi)$ if and only if $v_n = v$ and each constraint in φ is satisfied when the automaton

has stayed in v_n for delay δ_n, i.e. for each variable constraint $a \leq c_0x_0+c_1x_1+\cdots+c_lx_l \leq b$ in φ, $a \leq c_0\zeta_n(x_0) + c_1\zeta_n(x_1) + \cdots + c_m\zeta_n(x_l) \leq b$ where $\zeta_n(x_k)$ $(0 \leq k \leq l)$ represents the value of x_k when the automaton has stayed at v_n for the delay δ_n. *H satisfies $\mathcal{R}(v, \varphi)$* if and only if there is a behavior of H which satisfies $\mathcal{R}(v, \varphi)$. □

Definition 5. Given LHA $H = (X, \Sigma, V, V^0, E, \alpha, \beta, \gamma)$, and reachability specification $\mathcal{R}(v, \varphi)$, by introducing a new sink location $v_{\mathcal{R}}$ and a sink transition $e_{\mathcal{R}}$ into H, which results a new LHA $H_{\mathcal{R}}$. The satisfiability of $\mathcal{R}(v, \varphi)$ on LHA H are equivalent to the reachability of $v_{\mathcal{R}}$ in $H_{\mathcal{R}}$ iff $\alpha(v_{\mathcal{R}}) = \emptyset$ and $e_{\mathcal{R}} = (v, \sigma, \phi, \psi, v_{\mathcal{R}})$, where $v = v \in \mathcal{R}(v, \varphi)$, $\phi = \mathcal{R}(v, \varphi)$, $\psi = \emptyset$. □

Based on the above definition, without loss of generality, in the following paragraph, we will only discuss the reachability problem of a given location in the LHA, which covers the verification of the reachability specification $\mathcal{R}(v, \varphi)$.

According to Definition 2, the reachability verification problem of location v_n along the path ρ can be translated into the satisfiability problem of a set of constraints on variables δ_i and $\zeta_i(\boldsymbol{x})$ where $(0 \leq i \leq n)$. If we use notation $\Theta(\rho, v_n)$ to represent this set of linear constraints, we can check whether ρ reaches location v_n by checking whether $\Theta(\rho, v_n)$ has a solution, which can be solved by linear programming (LP) efficiently.

Bounded Reachability Verification. The bounded reachability analysis is to look for a system trajectory in a given threshold which can satisfy the given specification. Last paragraph gives a technique to verify the reachability of an abstract path in the LHA. Based on that, we proposed a bounded reachability verification method in[9] to traverse the system structure directly by DFS and check all the potential paths one by one until a feasible path to the reachability target is found or the given threshold is reached.

The pseudocode for this algorithm is shown in Table.1. The main function is Verify(H, v, *bound*) where H is the LHA, v is the reachability target location and *bound* is the value of the threshold. This function traverses the graph structure by calling function TRAVERSE(*stack*) recursively, where the input parameter *stack* is the stack which contains the current visiting path. When the function finds a path which satisfies the specification, it returns 1, then the upper caller will be informed and the DFS will be terminated. If the return value is 0, the caller will remove the last location from the ongoing path and keep on traversing. Because whenever a new location is added into the ongoing path, the algorithm will check the feasibility of it, we call this algorithm the "Eager"-DFS based bounded reachability analysis algorithm.

Instead of encoding the whole problem space to a group of formulas like SAT-style solver, which suffers the state space explosion a lot when dealing with big problems, this plain DFS style approach only needs to keep the discrete structure and current visiting path in memory, and check each potential path one by one, which makes it possible to solve big problems as long as enough time is given. The case studies given in[9] give a demonstration of this approach which also supports our belief of this argument.

3 Pruning Algorithm For DFS Optimization

The DFS-based algorithm for bounded reachability analysis of LHA reviewed in the last section gives an intuitive method to traverse and check one path at a time[9]. As

Table 1. Eager-DFS Based Bounded Reachability Analysis of LHA

VERIFY (H, v, *bound*)
1. **for** each location $v_I \in V^0$:
2. **begin**
3. new stack s;
4. s.push(v_I);
5. int res=TRAVERSE(s);
6. if (res==1) **return** true;
7. s.pop(v_I);
8. **end**
9. **return** false;

TRAVERSE (*stack s*)
1. Get the ongoing path $\rho = \langle v_0 \rangle \xrightarrow[\sigma_0]{(\phi_0,\psi_0)} \langle v_1 \rangle \xrightarrow[\sigma_1]{(\phi_1,\psi_1)} \cdots \xrightarrow[\sigma_{n-1}]{(\phi_{n-1},\psi_{n-1})} \langle v_n \rangle$ from stack s;
2. check the feasibility of ρ;
3. **if** (infeasible) **return** 0;
4. **if** ((feasible)&&(v_n == v)) **return** 1;
5. **if** (($s.depth$ == *bound*)||(v_n doesn't have any successive location)) **return** 0;
6. **for** each successive location *sloc* of v_n
7. **begin**
8. s.push(*sloc*);
9. boolean res=TRAVERSE(s);
10. if (res==1) **return** res;
11. s.pop(*sloc*);
12. **end**
13. **return** 0;

the number of paths under given bound are finite, this algorithm is guaranteed to terminate. Furthermore, the algorithm only checks the reachability of one path, therefore the memory usage will not blow up quickly without control.

3.1 Target Location-Guided Lazy-DFS

Basically, the above algorithm makes a tradeoff between space and time to handle problems with large size. Using our DFS-style BMC algorithm, we can solve a problem with practical size given enough time. As a result, we have a stable ground for the control of memory usage. Now let's turn our direction to time control, which means we want to give an algorithm to traverse the bounded behavior tree of a LHA more efficiently.

The current Eager-DFS algorithm checks all the paths under the given threshold. When the threshold is large and the graph structure is complex, there could be numerous candidate paths to check, which could consume a considerably large amount of computation time. Take the LHA in Fig.1 for example, suppose we want to check whether v_6 is reachable within bound 7, the related bounded behavior tree of this automaton is shown in Fig.2, which has 37 candidate paths, for example, $\langle v_1 \rangle \xrightarrow[e_1]{} \langle v_2 \rangle$ and so on. This means the DFS procedure could call the underlying LP solver 37 times in

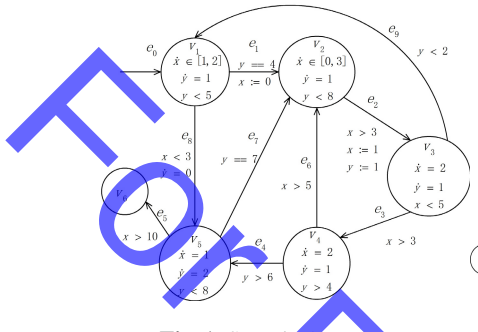


Fig. 1. Sample Automaton

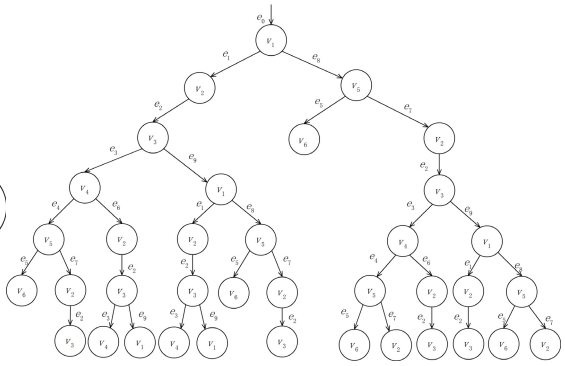


Fig. 2. Behavior Tree With Bound 7

the worst case. If the size of bound is larger, the number of path segments need to check could blow up quickly, which will be the main bottleneck for the entire bounded verification. Therefore, if there is a method to decrease the number of paths need to check, then the efficiency of the above algorithm can be improved for sure.

By investigating the 37 paths, we can find that most of the paths are not even related with the reachability specification. As we are checking whether location v_6 is reachable in bound 7, path segments like $\langle v_1\rangle \xrightarrow[e_1]{} \langle v_2\rangle$, $\langle v_1\rangle \xrightarrow[e_1]{} \langle v_2\rangle \xrightarrow[e_2]{} \langle v_3\rangle$ don't have location v_6 involved, therefore these paths can not satisfy the specification for sure and it could be a waste of time to check their feasibilities by calling a LP solver.

Based on this intuitive idea, we give the first straightforward optimization as follows: When checking whether location v is reachable along a path ρ, if the location v is not contained in ρ, the investigation of the feasibility of ρ will be postponed until v is traversed. Comparing with the Eager-DFS algorithm presented in the last section which checks the feasibility of the ongoing path whenever a new location is traversed, in this optimization, the calling of the LP solver will only be conducted once the new traversed location is specification related. Therefore, we name this algorithm as "Lazy"-DFS.

The pseudocode for the function TRAVERSE in Lazy-DFS algorithm is shown below in Table.2. The main difference between this algorithm and the algorithm in Table.1 is the checking of the feasibility of the ongoing path is moved into the branch with $v_n == v$. It is clear to see that only when the last location of the current visiting path ρ is the target location, the DFS procedure will call the underlying decision procedure to translate the feasibility of ρ into a linear constraint set and verify it by LP. Otherwise, the DFS will just go on traversing on the graph structure. Thus, the number of paths need to be checked can be reduced significantly to raise the efficiency. Again. let's take the automaton given in Fig.1 for example. Under Lazy-DFS, there are only 5 paths need to call the underlying decision procedure to check, e.g., $\langle v_1\rangle \xrightarrow[e_1]{} \langle v_2\rangle \xrightarrow[e_2]{} \langle v_3\rangle \xrightarrow[e_3]{} \langle v_4\rangle \xrightarrow[e_4]{} \langle v_5\rangle \xrightarrow[e_5]{} \langle v_6\rangle$, $\langle v_1\rangle \xrightarrow[e_8]{} \langle v_5\rangle \xrightarrow[e_5]{} \langle v_6\rangle$ and so on.

Table 2. Lazy-DFS Based on Target Location-Guided Checking

TRAVERSE (*stack s*)
1 . Get the ongoing path $\rho = \langle v_0\rangle \xrightarrow[\sigma_0]{(\phi_0,\psi_0)} \langle v_1\rangle \xrightarrow[\sigma_1]{(\phi_1,\psi_1)} \ldots \xrightarrow[\sigma_{n-1}]{(\phi_{n-1},\psi_{n-1})} \langle v_n\rangle$ from stack s;
2 . **if**(v_n == v)
3. **begin**
4 . check the feasibility of ρ;
5 . **if** (feasible) **return** 1 **else return** 0;
6. **end**
7 . **if** (($s.depth$ == $bound$)||(v_n doesn't have any successive location)) **return** 0;
8 . **for** each successive location *sloc* of v_n
9. **begin**
10. s.push(*sloc*);
11. int res=TRAVERSE(s);
12. if (res==1) **return** res;
13. s.pop(*sloc*);
14. **end**
15. **return** 0;

3.2 IIS-Based Infeasible Constraint Locating and Backtracking

In general, by only checking the paths which are specification related, the times of calling the underlying LP solver can be reduced greatly. But it is not always the case. Still take the automaton given in Fig.1 for example. Based on the Lazy-DFS algorithm given in Table.2, when checking the automaton according to target location v_6, the first path that the algorithm will call the underlying decision procedure is: $\rho = \langle v_1\rangle \xrightarrow[e_1]{} \langle v_2\rangle \xrightarrow[e_2]{} \langle v_3\rangle \xrightarrow[e_3]{} \langle v_4\rangle \xrightarrow[e_4]{} \langle v_5\rangle \xrightarrow[e_5]{} \langle v_6\rangle$. Suppose it is proved that ρ is not feasible by calling the underlying LP solver, the algorithm will pop out the last location v_6 from the stack and visit the next branch from v_5. Suppose the path segment $\rho'' = \langle v_1\rangle \xrightarrow[e_1]{} \langle v_2\rangle \xrightarrow[e_2]{} \langle v_3\rangle \xrightarrow[e_3]{} \langle v_4\rangle$ is already infeasible, as ρ'' is not related with the reachability target, under Lazy-DFS the feasibility of ρ'' will not be checked at all. But, if the algorithm deployed is the Eager-DFS which checks all the path segments as shown in Table.1, the feasibility of ρ'' will be checked right after location v_4 is added into the ongoing path. Once ρ'' is proved to be infeasible, then a backtracking will be conducted immediately, which means the subtree starting from location v_4 with prefix as ρ'' will not be traversed in Eager-DFS at all. But, in the Lazy-DFS algorithm, this subtree will still be traversed. So, is there a method which can reduce the times of solving LP problems as proposed by Lazy-DFS and also backtrack to the exact place where infeasibility happened to prune the behavior tree as Eager-DFS? The answer is yes!

Now, let's come back to the automaton given in Fig.1, in location v_3 we have $\dot{x} = 2$, $\dot{y} = 1$ and $x < 5$. According to definition 2, in the related constraint set $\mathbb{R}$, there are accordingly constrains: $\delta_{v_3} > 0$, $\zeta_{v_3}(x) - \lambda_{v_3}(x) = 2\delta_{v_3}$, $\zeta_{v_3}(y) - \lambda_{v_3}(y) = \delta_{v_3}$, $\zeta_{v_3}(x) < 5$, where $\lambda_{v_3}(x) = \lambda_{v_3}(y) = 1$ as x and y are reset to 1 on transition e_2. On transition e_3,

there is guard $x > 3$. In location v_4, there is invariant $y > 4$. Therefore, we also have constraints $\zeta_{v_3}(x) > 3$ and $\zeta_{v_3}(y) > 4$ in $\mathbb{R}$. If we name this set of constraints as $\mathbb{R}_{\rho'}$, clearly it is unsatisfiable. As $\zeta_{v_3}(x) < 5$, $\zeta_{v_3}(x) > 3$, and $\zeta_{v_3}(x) - 1 = 2\delta_{v_3}$, we can get $1 < \delta_{v_3} < 2$. Because $\zeta_{v_3}(y) - 1 = \delta_{v_3}$, we can get $2 < \zeta_{v_3}(y) < 3$, which contradicts with $\zeta_{v_3}(y) > 4$. As these constraints are generated according to the invariants and guards from transition e_2, e_3, and location v_3, v_4, this implies the path segment $\langle v_2\rangle \xrightarrow[e_2]{} \langle v_3\rangle \xrightarrow[e_3]{} \langle v_4\rangle$ is infeasible, but $\langle v_2\rangle \xrightarrow[e_2]{} \langle v_3\rangle$ is an feasible one. Therefore, if the DFS algorithm is clever enough, it will backtrack to the location v_3 and traverse the next branch $\langle v_3\rangle \xrightarrow[e_9]{} \langle v_1\rangle$. So, now the problem is how to locate such a backtracking point?

The answer is the irreducible infeasible set (IIS) technique[12]. Generally speaking, a set of linear constraints $\mathbb{R}$ is said to be satisfiable, if there exists a valuation of all the variables which makes all the constraints in $\mathbb{R}$ to be true. Otherwise, $\mathbb{R}$ is unsatisfiable. If $\mathbb{R}$ is unsatisfiable, then IIS of $\mathbb{R}$ is a subset $\mathbb{R}' \subseteq \mathbb{R}$ that $\mathbb{R}'$ is unsatisfiable and for any $\mathbb{R}'' \subset \mathbb{R}'$, $\mathbb{R}''$ is satisfiable.

Intuitively speaking, the IIS of a linear constraint set is an unsatisfiable set of constraints that becomes satisfiable if any constraint is removed. Fortunately, quoted from[12], the algorithm to locate the IIS from a unsatisfiable set is "simple, relatively efficient and easily incorporation into standard LP solvers". Actually many software packages are available which supports the efficient analysis of a linear constraint set and locating of the minimal IIS, such as MINOS[16], IBM CPLEX[17] and LINDO[18]. Therefore, given an infeasible path ρ, we can simply analyze the constraint set $\mathbb{R}$ generated according to this path to locate the IIS $\mathbb{R}'$. Now, if there is a mapping function to map each constraint $\nabla \in \mathbb{R}'$ to the original elements in the path, we can manipulate the structure of the bounded depth behavior tree more efficiently.

Definition 6. Given LHA $H = (X, \Sigma, V, V^0, E, \alpha, \beta, \gamma)$, path $\rho = \langle v_0\rangle \xrightarrow[\sigma_0]{(\phi_0,\psi_0)} \langle v_1\rangle \xrightarrow[\sigma_1]{(\phi_1,\psi_1)} \ldots \xrightarrow[\sigma_{n-1}]{(\phi_{n-1},\psi_{n-1})} \langle v_n\rangle$, and linear constraint set $\mathbb{R}$ which is generated according to the feasibility of ρ. For constraint $\nabla \in \mathbb{R}$, the stem location set $\mathbb{V}_\nabla$ of ∇ in ρ is defined as follows:

- if ∇ is generated according to the time duration on location v_i $(0 \le i \le n)$, $\delta_{v_i} \ge 0$, $v_i \in \mathbb{V}_\nabla$;
- if ∇ is generated according to the transition guard in ϕ_i on transition e_i $(0 \le i \le n-1)$, $v_{i+1} \in \mathbb{V}_\nabla$. If $i > 0$, $v_{i-1} \in \mathbb{V}_\nabla$ as well;
- if ∇ is generated according to the reset action in ψ_i on transition e_i $(0 \le i \le n-1)$, $v_{i+1} \in \mathbb{V}_\nabla$.If $i > 0$, $v_{i-1} \in \mathbb{V}_\nabla$ as well;
- if ∇ is generated according to the flow conditions in β_{v_i} location v_i $(0 \le i \le n)$, $v_i \in \mathbb{V}_\nabla$. If $i > 0$, $v_{i+1} \in \mathbb{V}_\nabla$ as well;
- if ∇ is generated according to the invariants in α_{v_i} in location v_i $(0 \le i \le n)$;
  - if ∇ is generated according to $\zeta_i(\boldsymbol{x})$, $v_i \in \mathbb{V}_\nabla$
  - if ∇ is generated according to $\lambda_i(\boldsymbol{x})$, $v_i \in \mathbb{V}_\nabla$, if $i > 0$, $v_{i-1} \in \mathbb{V}_\nabla$ as well. □

Definition 7. Given LHA $H = (X, \Sigma, V, V^0, E, \alpha, \beta, \gamma)$, path $\rho = \langle v_0\rangle \xrightarrow[\sigma_0]{(\phi_0,\psi_0)} \langle v_1\rangle \xrightarrow[\sigma_1]{(\phi_1,\psi_1)} \ldots \xrightarrow[\sigma_{n-1}]{(\phi_{n-1},\psi_{n-1})} \langle v_n\rangle$ and linear constraint set $\mathbb{R}$ which is generated according to the feasibility

of ρ. For a set $\mathbb{R}' = \{\nabla_1, \nabla_2, \ldots, \nabla_m\} \subseteq \mathbb{R}$, the stem location set of $\mathbb{R}'$ is $\mathbb{V}_{\mathbb{R}'} = \mathbb{V}_{\nabla_1} \cup \mathbb{V}_{\nabla_2} \cup \cdots \cup \mathbb{V}_{\nabla_m}$. □

Basically speaking, given a path ρ and the linear constraint $\mathbb{R}_\rho$, the above two definitions mark each constraints ϕ in $\mathbb{R}_\rho$ according to a location v in ρ. This means, ϕ will not be added into $\mathbb{R}_\rho$ until ρ travels to location v. Now, let's review the constraint set $\mathbb{R}_{\rho'}$ given in the beginning of this section again. $\mathbb{R}_{\rho'} = \{\delta_{v_3} > 0, \zeta_{v_3}(x) - \lambda_{v_3}(x) = 2\delta_{v_3}, \zeta_{v_3}(y) - \lambda_{v_3}(y) = \delta_{v_3}, \lambda_{v_3}(x) = 1, \lambda_{v_3}(y) = 1, \zeta_{v_3}(x) < 5, \zeta_{v_3}(x) > 3, \zeta_{v_3}(y) > 3\}$. Clearly the stem location set of $\mathbb{R}_{\rho'}$ is $\mathbb{V} = \{v_2, v_3, v_4\}$.

Suppose $\mathbb{R}_{\rho'}$ is the only IIS in the constraint set of path $\rho = \langle v_1\rangle \underset{e_1}{\longrightarrow} \langle v_2\rangle \underset{e_2}{\longrightarrow} \langle v_3\rangle \underset{e_3}{\longrightarrow} \langle v_4\rangle \underset{e_4}{\longrightarrow} \langle v_5\rangle \underset{e_5}{\longrightarrow} \langle v_6\rangle$, because if any constraint in $\mathbb{R}_{\rho'}$ is removed, the new constraint set is satisfiable. Then, it clearly implies that the path segment before reaching the location with the biggest index in the stem location set of $\mathbb{R}_{\rho'}$, which is v_4 in ρ, is feasible. So the sub tree starting from v_4 after $\langle v_1\rangle \underset{e_1}{\longrightarrow} \langle v_2\rangle \underset{e_2}{\longrightarrow} \langle v_3\rangle$ doesn't need to be traversed.

Furthermore, as mentioned above, the constraint set that IIS technique located can be mapped back to a path segments ρ' in the path ρ. This path segment can be saved as a guideline for the future traversing, once a new traversed path ρ'' contains an exact path segment as ρ', we can simply falsify ρ'' for verification without call the underlying decision procedure, since the syntax elements in ρ' has already been proved to be infeasible in ρ, the occurrence of ρ' in ρ'' will just be translated into the same set of unsatisfiable constraints with just variable name changed.

Based on the above discussion, the optimized function TRAVERSE (*stack*) is given below in Table.3. A new function IIS (*stack*) is introduced in Table.3 as well. This function finds the IIS in the constraint set according to the ongoing path $\langle v_0\rangle \underset{\sigma_0}{\overset{(\phi_0,\psi_0)}{\longrightarrow}} \langle v_1\rangle \underset{\sigma_1}{\overset{(\phi_1,\psi_1)}{\longrightarrow}} \ldots \underset{\sigma_{m-1}}{\overset{(\phi_{m-1},\psi_{m-1})}{\longrightarrow}} \langle v_m\rangle$ at first, then locates the stem location set from the IIS constraint set. By locating the node v_k in the set with the largest index, this function will inform the upper caller to backtrack to location v_{k-1} by indicating the distance between v_{k-1} and v_m.

Furthermore, once a path segment ρ' is located in the IIS, ρ' will be added into a global vector Ω as "bad examples". Then in the Traverse function, once a path is found to be specification related, the algorithm will check whether this path contains any "bad example". If any of the bad examples is hit, the Traverse function will directly return the backtracking step to the upper caller[1].

Based on the algorithm given in Table.3, once a path ρ is proved to be infeasible, an IIS based method will be called to locate the path segment which makes ρ infeasible. Then the DFS algorithm can backtrack to the right position to prune the bounded behavior tree efficiently. Besides that, the path segment will be saved to falsify the other new generated paths under checking to save the computation time.

[1] Generally speaking, the mapping with bad examples can be preformed once a new location is added to the path, but the matching will be time consuming if the size of example set is huge, so we decide to be lazy again to postpone the matching until the target is found.

Table 3. IIS-DFS Based On Infeasible Path Segment Localization

TRAVERSE (*stack s*)

1 . Get the ongoing path $\rho = \langle v_0\rangle \xrightarrow[\sigma_0]{(\phi_0,\psi_0)} \langle v_1\rangle \xrightarrow[\sigma_1]{(\phi_1,\psi_1)} \cdots \xrightarrow[\sigma_{n-1}]{(\phi_{n-1},\psi_{n-1})} \langle v_n\rangle$ from stack s;
2 . **if**(v_n == v)
3. **begin**
4 . **if** ($\exists\omega \in \Omega$ && ω is a path segment in ρ)
5 . locate ω in ρ as $\rho_\omega = \langle v_i\rangle \xrightarrow[\sigma_i]{(\phi_i,\psi_i)} \langle v_{i+1}\rangle \xrightarrow[\sigma_{i+1}]{(\phi_{i+1},\psi_{i+1})} \cdots \xrightarrow[\sigma_{j-1}]{(\phi_{j-1},\psi_{j-1})} \langle v_j\rangle$;
6 . **return** j-n;
7 . check the feasibility of ρ;
8 . **if** (unfeasible)
9 . **return** IIS(s);
10. **end**
11 . **if** (($s.depth == bound$)||(v_n doesn't have any successive location)) **return** 0;
12 . **for** each successive location *sloc* of v_n
13. **begin**
14. s.push(*sloc*);
15. int res=TRAVERSE(s);
16. if (res==1) **return** res;
17. s.pop(*sloc*);
18. if (res < 0) **return** res+1;
19. **end**
20. **return** 0;

IIS (*stack s*)

1 . Get the ongoing path $\rho = \langle v_0\rangle \xrightarrow[\sigma_0]{(\phi_0,\psi_0)} \langle v_1\rangle \xrightarrow[\sigma_1]{(\phi_1,\psi_1)} \cdots \xrightarrow[\sigma_{m-1}]{(\phi_{m-1},\psi_{m-1})} \langle v_m\rangle$ from stack s;
2 . Locate the stem location set $\mathbb{V}_\rho$ and the accordingly path segment ρ' of ρ;
3 . Ω.add(ρ');
4 . Get the location v_k in $\mathbb{V}_\rho$ with the largest index;
5 . return k-m;

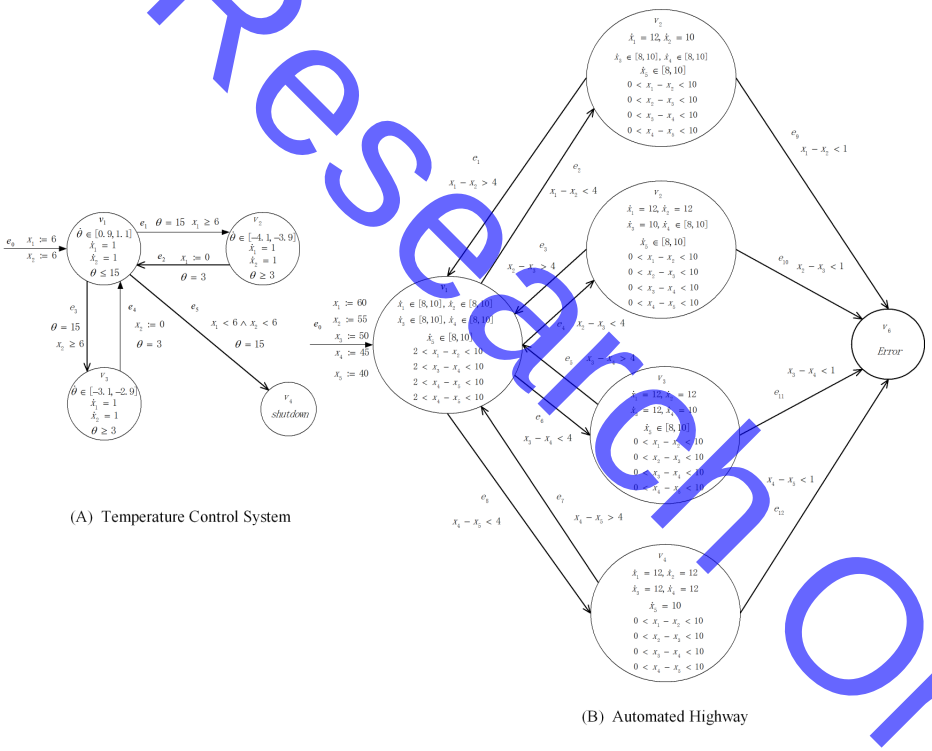
4 Case Studies

In order to evaluate the performance of the optimization methods presented in this paper, we upgrade our bounded reachability checker for LHA: BACH[9, 14] to a new version BACH 3 (`http://seg.nju.edu.cn/BACH/`). BACH 3 shares the graphical LHA Editor with BACH. As the LP solver underlying BACH is OR-objects[15] which does not support the functionality of IIS analysis. BACH 3 calls the IBM CPLEX[17] instead, which gives a nice support of IIS analysis.

In the experiments, we evaluate the performance of BACH 3 under three different settings according to the underlying DFS algorithm, which are Eager-DFS, Lazy-DFS and IIS-DFS respectively. The experiments are conducted on a DELL workstation (Intel Core2 Quad CPU 2.4GHz, 4GB RAM).

As the comparisons between Eager-DFS and other related tools are already reported in[9], in this section, we focus on the comparison between the three different DFS algorithms to show the performance of the optimization methods presented in this paper.

We use three benchmarks in the experiments. The first LHA is the sample automaton given in Fig.1 in this paper. The second one is the temperature control system used in our previous case studies in[9]. The third automaton is the automated highway example introduced in[19] with 5 cars included. These automata are shown in Fig.3. For the sample automaton, we are checking whether location v_6 is reachable. In the other two automata, the reachability of the target location under checking is v_4 for temperature control system, which stands for that no rod is available in the nuclear reactor; and v_6 for the automated highway which stands for that a car collision will happen.



(A) Temperature Control System

(B) Automated Highway

Fig. 3. Experimental Automata

We conduct these three DFS algorithms on all of these three automata. The time limit is set as 2 hours in the experiments. The performance data for each benchmark are shown in Table.4,5,and 6 respectively. In these tables we show the total time spent for each problem w.r.t different bound size. Furthermore, in order to show the performance of the optimization techniques in decreasing the number of paths to check, we also collect and report the times BACH 3 calls the underlying LP solver -CPLEX, each call

means a unique path is transformed into an LP constraint set and solved by CPLEX. To demonstrate these data more intuitively, we also show the plotted graphs in Fig.4.

Table 4. Performance Data On The Sample Automaton In 2 Hours

Tech. / *Bound*	Eager-DFS		Lazy-DFS		IIS-DFS	
	Total Time (Sec.)	Call CPLEX	Total Time (Sec.)	Call CPLEX	Total Time (Sec.)	Call CPLEX
10	0.252	51	0.063	17	0.046	2
20	7.111	431	1.487	853	0.124	2
30	98.354	3223	44.610	46037	0.343	2
40	1036.987	23743	2784.989	2544981	1.322	2
50	N/A	N/A	N/A	N/A	10.729	2
70	N/A	N/A	N/A	N/A	1040.308	2

Table 5. Performance Data On The Temperature Control System Benchmark In 2 Hours

Tech. / *Bound*	Eager-DFS		Lazy-DFS		IIS-DFS	
	Total Time (Sec.)	Call CPLEX	Total Time (Sec.)	Call CPLEX	Total Time (Sec.)	Call CPLEX
5	0.062	16	0.022	3	0.075	3
15	1.595	636	0.58	127	0.595	23
25	46.004	20476	9.669	4095	1.77	43
35	2256.519	655356	386.743	131071	19.867	63
40	N/A	N/A	3389.555	1048575	141.046	75
50	N/A	N/A	N/A	N/A	6470.308	95

Table 6. Performance Data On The Automated Highway System Benchmark In 2 Hours

Tech. / *Bound*	Eager-DFS		Lazy-DFS		IIS-DFS	
	Total Time (Sec.)	Call CPLEX	Total Time (Sec.)	Call CPLEX	Total Time (Sec.)	Call CPLEX
5	1.115	61	0.113	20	0.383	20
10	27.877	2045	1.328	340	0.656	20
15	822.996	65533	79.611	21844	1.647	20
20	N/A	N/A	1689.658	349524	28.225	20
25	N/A	N/A	N/A	N/A	1242.241	20

We can see that with any of the optimizations deployed, the size of the problem that can be solved are increased significantly and the performance for the same question are clearly optimized. Furthermore, IIS-DFS outperforms Lazy-DFS substantially. Take the automated highway system as example, when bound is set as 15, it cost Eager-DFS 822.9 seconds to check 65533 paths. By using Lazy-DFS, the verification time is decreased to 79.6 seconds by only checking 21844 paths. Finally, when use IIS-DFS, the verification is finished in only 1.6 seconds, and only 20 paths are verified.

The reason is that in our DFS schema, each time a candidate path is found, the algorithm will call the underlying LP solver to reason the feasibility of the path. When the size of path and/or the number of candidate paths is large, the reasoning by LP will be very time consuming. By using optimization techniques presented in this paper, the number of paths need to check is reduced significantly, thus, it is possible to solve problem more quickly and to solve larger problems. In detail:

– By introducing Lazy-DFS, the number of candidate paths to check can be reduced in most of the cases, that's the reason that Lazy-DFS outperforms Eager-DFS.

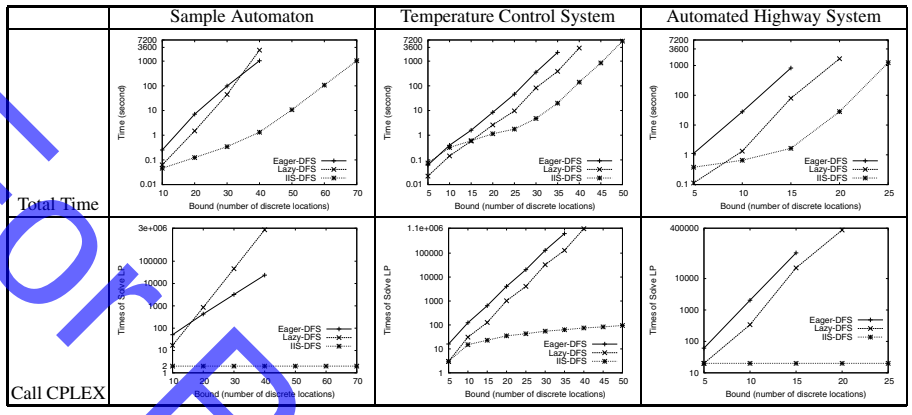


Fig. 4. Performance of Bounded Reachability Analysis in 2 Hours

- By introducing IIS-DFS, when a path is infeasible, IIS can locate the exact path segment where the infeasibility happens to guide the backtracking. Besides this, the infeasible path segment can be saved as a "bad example" that if any future candidate path has a same path segments as the "bad example", the candidate path can be falsified for the feasibility reasoning without call the underlying LP solver. That's the reason that IIS-DFS outperforms Lazy-DFS in almost all the experiments.
- Indeed, if a candidate path can be matched with a "bad example", then it can be falsified directly without call the underlying LP solver to save computation time. Nevertheless, when the size of the candidate path set is huge, the comparison between each of the candidate path and the "bad example" set will also be very time consuming, that's the reason that as shown in our data, the total time spent is not proportional to the times of calling CPLEX.

5 Conclusion

The bounded reachability analysis of hybrid automata is difficult. Even for the simple class of linear hybrid automata (LHA), the state-of-the-art tools can only analyze systems with few continuous variables, control nodes and small bound.

In this paper, we present an algorithm to check the bounded reachability of LHA in a DFS manner. Only the abstract path related with the reachability specification will be analyzed by the underlying LP solver. If the path is judged to be infeasible, the IIS technique will be deployed on the infeasible path to locate the path segment which makes this path infeasible to guide the backtracking of the DFS.

We implement the optimization techniques presented in this paper into BACH which is a bounded reachability checker for LHA. The experiments on BACH greatly strengthen our belief that with the help of the optimization methods presented in this paper, the size of the problem that BACH can solve is increased substantially while the time for solving the same problem is reduced significantly as well.

Acknowledgement. The authors are supported by the National 863 High-Tech Programme of China (No.2011AA010103), the National Natural Science Foundation of China (No.90818022, No.61100036) and by the Jiangsu Province Research Foundation (BK2011558).

References

1. Henzinger, T.A.: The theory of hybrid automata. In: Proceedings of LICS 1996, pp. 278–292. IEEE Computer Society (1996)
2. Henzinger, T.A., Kopke, P.W., Puri, A., Varaiya, P.: What's Decidable About Hybrid Automata? Journal of Computer and System Sciences 57, 94–124 (1998)
3. Henzinger, T.A., Ho, P.-H., Wong-Toi, H.: Algorithmic Analysis of Nonlinear Hybrid Systems. IEEE Transactions on Automatic Control, 540–554 (1998)
4. Alur, R., Courcoubetis, C., Halbwachs, N., Henzinger, T.A., Ho, P.-H., Nicollin, X., Olivero, A., Sifakis, J., Yovine, S.: The algorithmic analysis of hybrid systems. Theoretical Computer Science 138, 3–34 (1995)
5. Biere, A., Cimatti, A., Clarke, E., Strichman, O., Zhu, Y.: Bounded Model Checking. In: Advance in Computers, vol. 58, pp. 118–149. Academic Press (2003)
6. Fränzle, M., Herde, C., Ratschan, S., Schubert, T., Teige, T.: Efficient solving of large non-linear arithmetic constraint systems with complex boolean structure. Journal on Satisfiability, Boolean Modeling and Computation 1, 209–236 (2007)
7. Audemard, G., Bozzano, M., Cimatti, A., Sebastiani, R.: Verifying Industrial Hybrid Systems with MathSAT. In: Proceedings of BMC 2004, ENTCS, vol. 119(2), pp. 17–32. Elsevier Science (2005)
8. Li, X., Jha, S.K., Bu, L.: Towards an Efficient Path-Oriented Tool for Bounded Reachability Analysis of Linear Hybrid Systems using Linear Programming. In: Proceedings of BMC 2006, ENTCS, vol. 174(3), pp. 57–70. Elsevier Science, 07 (2006)
9. Bu, L., Li, Y., Wang, L., Li, X.: BACH: Bounded Reachability Checker for Linear Hybrid Automata. In: Proceedings of the 8th International Conference on Formal Methods in Computer Aided Design, pp. 65–68. IEEE Computer Society (2008)
10. Henzinger, T.A., Ho, P.-H., Wong-Toi, H.: HYTECH: a model checker for hybrid systems. Software Tools for Technology Transfer 1, 110–122 (1997)
11. Frehse, G.: PHAVer: Algorithmic Verification of Hybrid Systems Past HyTech. In: Morari, M., Thiele, L. (eds.) HSCC 2005. LNCS, vol. 3414, pp. 258–273. Springer, Heidelberg (2005)
12. Chinneck, J., Dravnieks, E.: Locating minimal infeasible constraint sets in linear programs. ORSA Journal on Computing 3, 157–168 (1991)
13. Bu, L., Li, X.: Path-Oriented Bounded Reachability Analysis of Composed Linear Hybrid Systems. Software Tools Technology Transfer 13(4), 307–317 (2011)
14. Bu, L., Li, Y., Wang, L., Chen, X., Li, X.: BACH 2: Bounded ReachAbility CHecker for Compositional Linear Hybrid Systems. In: Proceedings of the 13th Design Automation & Test in Europe Conference, Dresden, Germany, pp. 1512–1517 (2010)
15. OR-Objects, `http://OpsResearch.com/OR-Objects/index.html`
16. Chinneck, J.: MINOS(IIS): Infeasibility analysis using MINOS. Computers and Operations Research 21(1), 1–9 (1994)
17. CPLEX, `http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/`
18. L. Systems Inc., `http://www.lindo.com/products/api/dllm.html`
19. Jha, S., Krogh, B.H., Weimer, J.E., Clarke, E.M.: Reachability for Linear Hybrid Automata Using Iterative Relaxation Abstraction. In: Bemporad, A., Bicchi, A., Buttazzo, G. (eds.) HSCC 2007. LNCS, vol. 4416, pp. 287–300. Springer, Heidelberg (2007)