Regular Paper

Interval-based Counterexample Analysis for Error Explanation

Takahisa Toda[1,a) Takeru Inoue[2,b)

Received: December 24, 2020, Accepted: July 7, 2021

Abstract: Model checking is an automated reasoning technique for the verification of hardware and software. If there is a fault in a system description, model checkers return, as an explanation of failure, a single execution trace of the system that results in an error state. Counterexamples are useful clues for locating faults, however, there is a big gap between computing counterexamples and locating faults, and the fault localization task is done by a manual inspection of counterexamples, which largely depends on individual expertise and intuition. Effective explanation of the failure is, thus, considered as an important issue. Since a single counterexample returned by model checkers is only one instance of failing executions, it is hard to gain clear perspective on the failure with just one specific case. In this paper we take another approach for error explanation: we generate many counterexamples and then abstract an essence of the failure from them. For example, in the formal verification of network configuration, a range of possible values (naturally identified with integers) to a single variable often makes it easier to understand the essence of the failure. In our experiments, such a range of values (called interval) is simply a set of consecutive IP addresses and can be substantially represented in two end addresses. We formulate the notion of intervals in a general setting. The concept of intervals is not limited to network configuration and it can be considered in an arbitrary system model as long as a variable on which interval is computed substantially takes integers. We present a method for computing the longest interval by combining bounded model checking, BDD, and AllSAT solver. To evaluate our method for the longest interval computation, we conduct experiments with a real network dataset and its randomly modified dataset. We confirm that about 8 millions of counterexamples are generated in 1.61 s and among them, the longest interval of length about 600 millions is reported in less than 0.01 s.

Keywords: bounded model checking, AllSAT solver, BDD, network verification, counterexample, error explanation

1. Introduction

Model checking is an automated reasoning technique that is used to determine whether a concurrent system never behaves against the intention of designers, i.e., *specifications* of the system. When it turns out in model checking process that some erroneous behavior can occur, model checkers show, as an explanation of failure, a single execution trace of the system that results in an error state, which is called a *counterexample*. Counterexamples are useful clues for locating faults in the system description. Thanks to the power of model checking, there are many applications to industry [4] and its practical utility is well-recognized.

Model checking alone is not enough to fix erroneous systems. Although counterexamples are automatically computed by model checkers, locating faults is not immediate. Indeed, there is a big gap between faults in system descriptions and execution traces caused by them. Generally, fault localization is done by a manual inspection of counterexamples, which largely depends on individual expertise and intuition. Doing this task manually is cumbersome and depressing. Due to the inherent hardness, rather than automating or semi-automating this task, we consider it more realistic to concentrate on *error explanation*, which is defined as the task to aid users in moving from a trace of failure to an understanding of the essence of the failure and perhaps, to a correction for the problem [8].

There are some papers that share the same motivation as ours, i.e., error explanation. They are categorized into three approaches as follows. The first approach is to focus on specific domains and largely exploit their knowledge. A typical technique is animations. However, animations do not work well when abstraction of models is excessively done. This is discussed in the context of railway interlocking verification [23] and natural language interpretation is used instead.

The second approach is to examine a single counterexample by transforming it into a more suitable form so that it becomes more informative. A typical transformation is minimization of counterexamples in its length or by excluding irrelevant parts. Another work minimizes the values of variables [9].

The third approach is to generate multiple counterexamples and analyze them with comparison. Almost all papers in this approach that we are aware of are concerned with software verification: they assume the notion of lines in source codes, but hardly use further knowledge, thereby they are distinguished from the domain specific approach. One work of the third approach introduces the notions of successful executions and failing execu-

[1] Graduate School of Informatics and Engineering, the University of Electro-Communications, Chofu, Tokyo 182–8585, Japan
[2] NTT Network Innovation Laboratories, NTT Corporation, Yokosuka, Kanagawa 239–0847, Japan
[a) todat@acm.org
[b) inoue.takeru@lab.ntt.co.jp

tions, exploiting the notion of lines, and presents techniques for comparing the two kinds of executions using set operations to see common and different features [10]. Another work applies Lewis' counterfactual theory of causality to program executions and presents a technique to generate probably-most-similar successful executions and to compare them with a failing execution by introducing a distance between executions and solving a pseudo-Boolean optimization problem [8].

We discuss the three approaches here. The domain specific approach can be effective if domain knowledge is fully utilized; however, it cannot be easily adapted to other domains. The single counterexample approach can be connected to model checkers without major change, which means it requires only adding some post-processing or replacing their counterexample finding algorithms; however without domain knowledge it is hard to know what is informative, and there are few clues to reduce the complexity of counterexamples. Moreover, a single counterexample returned by model checkers is only one instance of failing executions, and which counterexample is picked up from among many others is determined according to which search algorithm is taken. It is thus hard to gain clear perspective on the failure with just one specific case. In this respect, the multiple counterexamples approach is promising because of a large amount of information. The major challenge is how to take advantage of the amount of information. One counterexample is complicated, and many counterexamples are even more so. It is important to abstract an essence of the failure from many counterexamples in some way.

In this paper we take the multiple counterexample approach for error explanation: we generate many counterexamples and then abstract an essence of the failure from them. For example, in the formal verification of network configuration, a range of IP addresses often makes it easier to understand the essence of the failure because addresses in the same range are probably related to a common failure. We formulate such a range of possible values (hereafter called *interval*) naturally identified with integers in a general setting to allow not only network models but also arbitrary models that come from diverse domains as long as a variable on which the interval is computed substantially takes integers. It would be desirable that we could present all such intervals in compact and human-understandable form such as a bit pattern, however this task is more involved.

As a first step to automatize such an analysis, in this paper we concentrate on computing the longest interval, which captures many cases probably caused by a common failure. More concretely, we consider the following computation with user's aid in part.

(1) Perform a bounded model checking and obtain a counterexample π_B.
(2) Manually inspecting π_B, select a specific variable of integer type, v_T, from a system description.
(3) Perform the computation for generating all counterexamples such that all Boolean variables are assigned the same values as π_B but those encoded from v_T need not.
(4) Perform the computation for finding the longest interval of integers that are assigned to v_T in the initial states of the generated counterexamples.

Although our approach needs a user's aid in part, our approach provides a practical and useful way of better understanding the errors that are found in an ordinary model checking.

We present a practical method for computing the longest interval by combining bounded model checking, BDD, and AllSAT solver. Since the number of counterexamples generated in Step 3 is likely to blow up exponentially, our method utilizes a well-accepted space-efficient data structure BDD, seen as a kind of finite automaton, to avoid a combinatorial explosion. The computation in Step 3 is indirectly done by constructing a BDD that accepts counterexamples to be computed, in stead of searching them one by one. The computation in Step 4 is then run over the BDD. We will prove that Step 4 can be done in time proportional to the product of the number of Boolean variables encoded from v_T and the number of nodes in the BDD. This means that once BDDs can be successfully constructed within a realistic amount of time and space, Step 4 can be done quickly. This will be confirmed by experiments using a real network dataset and its randomly modified dataset.

In order to realize not only the longest interval computation but also other analyses based on the same approach, we present a generic computational framework that integrates bounded model checking with advanced functions such as constrained counterexample generation, counterexample databases, and AllSAT solvers.

The paper is organized as follows. Section 2 summarizes basic notions and terminology of model checking and BDDs. Section 3 describes a basic idea of counterexample analysis in the context of the verification of network configuration. Based on this, Section 4 formulates intervals, related notions, and the problem of computing the longest interval in a general setting. Our proposed method for the longest interval computation consists of the preprocessing part and the main part. Section 5 describes the preprocessing part, and Section 6 describes the main part and a computational complexity result. Section 7 presents the whole picture of our computational framework. Section 8 presents experimental results of the longest interval computation. Section 9 concludes the paper.

2. Preliminaries

In this section, we summarize model checking and BDDs, which can be skipped by familiar readers.

Model checking is a method for checking the behavior of systems whose states change over time in a non-deterministic manner [6], [7]. *Kripke structure* models such a system with (S, I, T, l), where S is a set of states; I is a set of initial states with $I \subseteq S$; T is a binary relation over S and $(s, s') \in T$ means that the transition from s to s' is possible in one step; and l is a function that maps each state s to a set of atomic propositions, meaning atomic propositions that evaluate to true in state s.

Model checking is to determine whether the model of a given system satisfies properties of system behaviors (called *specifications*). Specifications are described using temporal logic. Typical temporal operators are X, F, and G, which refer to the "next time", "some time in the future", and "all time in the future", respectively. Among variants of temporal logic, this paper concentrates on *linear temporal logic* (*LTL* for short) because our work uses bounded model checking.

A *path* is an infinite sequence of states such that all adjacent states satisfy the transition relation of a model. It represents one instance of possible system behaviors. The truth values of LTL formulas are determined along paths. This implies that a given specification may happen to be satisfied by one behavior, while falsified by another.

A (universal) *LTL model checking problem* is, given an LTL formula f and a Kripke structure M, to determine whether f holds along π for all paths π that start from initial states in M. LTL model checking is PSPACE-complete and it is hard to compute in general.

Bounded model checking is a practical method for LTL model checking [2]. The basic idea is to limit the range of states so that states to be examined are only those reachable from initial states in a certain number of steps and to determine whether there is an error state reachable, through states in that range, from initial states. If such an error state exists, bounded model checking returns, as a *counterexample* to the specification, a sequence of states representing one instance of failing executions. Since counterexamples are searched just within a limited portion of the vast search space, the completeness of verification is not ensured, although it is, in principle, possible to achieve by incrementally expanding a search range [20], [24]. Actually bounded model checking is widely accepted as a practical method for bug hunting rather than the means of verification.

A *binary decision diagram* (a *BDD* for short) is a graphical representation for Boolean functions [1], [3], [17]. We follow the terminology of the literature [16]. A BDD is an acyclic directed graph with exactly one node of indegree 0, which is called the *root*, and each non-terminal node has a variable index as its label and two children: *LO child* and *HI child*. The arc to a LO child (resp. a HI child) is called a *LO arc* (resp. a *HI arc*). The LO arc (resp. the HI arc) means that the value 0 (resp. 1) is assigned to the variable of the source node. There are two terminal nodes, denoted by $\top$ and $\bot$. Paths from the root to $\top$ correspond to the assignments by which a Boolean function evaluates to true. The variables of a BDD appear in a fixed order along paths from the root to terminal nodes. A BDD is *reduced* in that (1) every pair of equivalent subgraphs is shared and (2) every node whose HI arc and LO arc point to the same node is eliminated. It should be noted that *each node in a BDD is conventionally identified with the subgraph rooted by the node, which also forms a BDD.*

3. Basic Idea

In this section, we describe a basic idea of counterexample analysis in the context of the verification of network configuration, which is an active area attracting a lot of attention of researchers recently [14], [19], [21], [25].

Figure 1 depicts a network with three routers R_1, R_2, R_3 and three end nodes A, B, D, which is a simplified version of the network in the literature [18]. The table below the network shows a routing table. Packets have two fields, *ipdst* and *ipsrc*, with each represented as a bit sequence of length 3 for simplicity. Each row

$$\begin{tikzcd}
A \arrow[r] & R_1 \arrow[rr] \arrow[dr] & & R_2 \arrow[r] & B \\
 & & R_3 \arrow[ur] \arrow[r] & C &
\end{tikzcd}$$

in	ipdst	ipsrc	out
R_1	$10\star$	$01\star$	R_2
R_1	$1\star\star$	$\star\star\star$	R_3
R_2	$10\star$	$\star\star\star$	B
R_3	$\star\star\star$	$1\star\star$	C
R_3	$1\star\star$	$\star\star\star$	R_2

Fig. 1 Network and routing table.

of the table is a packet transfer rule. The column *in* denotes the current location of packets, *ipdst* and *ipsrc* denote patterns of addresses, where $\star$ is a wildcard and either 0 or 1 matches, and *out* denotes the next location of packets to be transferred. For each packet, the rules are scanned in order from the top. The first rule that matches all the conditions specified by *in*, *ipdst*, and *ipsrc* is applied. The packet is then transferred to the location specified by *out*. For example, R_1 transfers incoming packets with the *ipdst* field $10\star$ and the *ipsrc* field $01\star$ to R_2; among the remaining packets, R_1 transfers packets with the *ipdst* field $1\star\star$ to R_3; R_1 drops all the remaining packets.

Network configurations can be modeled using Kripke structure. For example, **Fig. 2** shows how the network of Fig. 1 is described in NuSMV2 model checker's language. States are represented by the two variables: *packet* and *location*, where *packet* holds two bit sequences for the *ipdst* and *ipsrc* fields, which are indicated by *packet.ipdst* and *packet.ipsrc*, and *location* holds one of the network nodes. The *ipsrc* and *ipdst* fields are declared as FRONZENVAR, meaning that these fields retain their initial values throughout state transitions. Word constants presented in this paper begin with 0*u*, followed by one of the characters *b* (binary) and *h* (hexadecimal); next comes an optional decimal integer giving the number of bits, then the character _ and lastly the constant value itself.

Let us now consider the following LTL formula, which is also described in NuSMV2 Language.

```
LTLSPEC
  packet.ipdst2 = 0ub3_100 -> F(location = b)
```

This formula states that if the upper 2 bits of the *ipdst* field matches 10, then the packet some time in the future reaches B. Here *packet.ipdst*2 is defined in Fig. 2 as the bit-wise AND of *packet.ipdst* and 110.

Running NuSMV2 model checker with the network model and the LTL formula above, we may obtain the following counterexample.

```
  -> State: 1.1 <-
    packet.ipsrc = 0ub3_101
    packet.ipdst = 0ub3_101
    location = a
  -> State: 1.2 <-
    location = r1
  -> State: 1.3 <-
    location = r3
  -- Loop starts here
  -> State: 1.4 <-
    location = c
  -> State: 1.5 <-
```

This means that if the $ipdst$ and $ipsrc$ fields of a packet are both 101, then the packet is transferred along $A \to R_1 \to R_3 \to C$ and it remains in C forever. This shows that the LTL formula is really falsified. Indeed, even though the $ipdst$ field of the packet matches 10$\star$, the packet does not reach B forever.

In this way, bounded model checking explains why the LTL formula is falsified by presenting one specific witness. This is informative thanks to the concreteness, however we are often not satisfied with just this because counterexamples are probably lengthy and complicated in practice. We need an error explanation in more abstract form.

```
MODULE packet
FROZENVAR
   ipsrc: unsigned word[3];
   ipdst: unsigned word[3];
DEFINE
   ipsrc1 := ipsrc & 0ub3_100;
   ipsrc2 := ipsrc & 0ub3_110;
   ipdst1 := ipdst & 0ub3_100;
   ipdst2 := ipdst & 0ub3_110;

MODULE main
VAR
   packet: packet;
   location: {a,b,c,r1,r2,r3,drop};
ASSIGN
   init(location) := a;
   next(location) :=
     case
       location = a:
        case
          TRUE: r1;
        esac;
       location = r1:
         case
           (packet.ipdst2 = 0ub3_100) & (packet
               .ipsrc2 = 0ub3_010): r2;
           packet.ipdst1 = 0ub3_100: r3;
           TRUE: drop;
         esac;
       location = r2:
         case
           packet.ipdst2 = 0ub3_100: b;
           TRUE: drop;
         esac;
       location = r3:
         case
           packet.ipsrc1 = 0ub3_100: c;
           packet.ipdst1 = 0ub3_100: r2;
           TRUE: drop;
         esac;
       location = b:
        case
          TRUE: b;
        esac;
       location = c:
        case
          TRUE: c;
        esac;
      TRUE: drop;
     esac;
```

Fig. 2 Network model described in NuSMV2's Language.

To see this, let us continue the network example. In order for the LTL formula to be falsified, there is no necessity for the $ipsrc$ and $ipdst$ fields to be both 101. We want to know other related cases that are probably caused by the same failure. For instance, let us consider what packets are transferred along the same route. Suppose, for simplicity, that we fix the $ipsrc$ field of a packet to 101. All we need to do is consider all possible values for the $ipdst$ field such that the LTL formula is falsified: in other words, among all packets such that the $ipdst$ field matches 10$\star$, consider all possible ones that are transferred along the same route. Inspecting the routing table in Fig. 1 manually, we will realize that they are the bit sequences matching 1 $\star$ $\star$. Such sequences form the interval ranging from 4 to 7 if a bit sequence is seen as the binary representation of an integer. Although it happens to be a single interval, but for other network configurations it would be pairwise disjoint intervals in general. It would be desirable that we could compute all such intervals and present them in a compact form such as a bit pattern, however this task is more involved.

As a first step to automatize such an analysis, in this paper we concentrate on computing the longest interval, which captures many cases probably caused by a common failure. Our proposed framework is applicable to arbitrary models that come from diverse domains as long as a variable on which interval is computed substantially has integers. In this paper, intervals are those indicated by bitmasks, but our framework can handle arbitrary intervals and thus applications are not limited to network only. Learning a bit pattern remains as future work.

4. Formulation

Based on the basic idea presented in the previous section, we formulate our general setting of this paper. The notations and the terminology introduced in this section will also appear in later sections.

Let $M = (S, I, T, l)$ be a Kripke structure. Let V be a set of variables, and suppose that every state in S is represented as an assignment of the variables of V to values. For all $x \in V$, the set of all possible values that can be assigned to x is called the *domain* of x. As in bounded model checking, variable domains are assumed to be finite.

Recall that a path π in M is an infinite sequence of states in S such that for all adjacent states s_i, s_{i+1}, the transition relation holds: $(s_i, s_{i+1}) \in T$. For all $i = 0, 1, \cdots$ and all $u \in V$, let us denote by π_i the i-th state of π, and by $\pi_i(u)$ the value assigned to u in π_i.

Let f be an LTL formula, and suppose that f is falsified by a path such that the first state is an initial state of M and the path has the form uv^ω for finite sequences of states $u = s_0 \cdots s_{l-1}, v = s_l \cdots s_k$. Here, v^ω represents the infinite repetitions of v. Let us fix such a path and denote it by π_B. We will call π_B a *base counterexample* at *bound k*. Since π_B is substantially represented by the first $k+1$ states, it will be sometimes identified with the sequence of the first $k + 1$ states, i.e., uv.

Let us fix a variable v_T of V such that the domain of v_T is totally ordered with respect to order relation $\leq$. Since the domain is finite, it can be identified with a (unsigned) integer domain. Without loss of generality, we can assume that such a domain is an interval, i.e., there is no missing integer between the minimum and the maximum integers in the domain. We will call v_T a *target variable*.

A counterexample π is *constrained* by π_B except v_T if $\pi_i(u) = \pi_{Bi}(u)$ for all $i = 0, 1, \cdots, k$ and all $u \in V\backslash\{v_T\}$. Since a base counterexample is fixed, we will sometimes omit π_B and simply say a *constrained counterexample*. The *longest interval problem* is to compute the longest interval of values, e, such that e is assigned to v_T in the initial state of some counterexample constrained by π_B except v_t. Here, a set of values, J, in a totally ordered domain is an *interval* if $a \leq b \leq c$ for $a, c \in J$ and a value b in the domain, then $b \in J$.

Our framework computes the longest interval problem by separating it into the preprocessing part and the main part. This will be separately explained in the succeeding sections.

5. Preprocessing

In this section, we describe the preprocessing part of the longest interval computation.

In the preprocessing part, our framework computes the set of all constrained counterexamples. Since the number of such counterexamples is likely to blow up exponentially, our method utilizes a well-accepted space-efficient data structure BDD (see Section 2), seen as a kind of finite automaton, to avoid a combinatorial explosion. The computation is indirectly done by constructing a BDD that accepts counterexamples to be computed, in stead of searching them one by one.

We do this computation with AllSAT solver. An AllSAT solver is a program for computing all satisfying assignments to a CNF (, which is a normal form of propositional formulas). Among several types of AllSAT solvers, we select the one integrating SAT solver and BDD [11], [12], [22]. A basic division of roles is that satisfying assignments are enumerated using a SAT solver, while they are added to a BDD. However, these two are not separately done. Actually, the solution enumeration is accelerated by using BDD as if it is a DP table. That is, the search for each subformula (obtained by applying the current partial assignment) is pruned if it turns out to have been done by looking it up in the BDD. Hence, the duplicated search for equivalent subformulas can be largely reduced.

The CNF fed into the AllSAT solver is obtained as follows. As in bounded model checking, encode a Kripke structure $M = (S, I, T, l)$ and an LTL formula f to a CNF ϕ. Suppose that the base counterexample π_B is the one obtained by solving ϕ. Through the encoding, for each state of π_B, the assignments for all variables of V except v_T are mapped to assignments for Boolean variables in ϕ. Hence, apply these truth assignments to ϕ. The resulted CNF, ϕ', is the one such that satisfying assignments to ϕ' exactly correspond to constrained counterexamples to f.

For the main part of the longest interval computation, we make two assumptions and a simplification of BDD. First, we fix a Boolean encoding for the target variable v_T as follows. Here, let us identify π_B with the sequence of states $s_0 s_1 \cdots s_k$, as noted in the previous section. For each state s_i, we introduce a set of Boolean variables, $bits_i(v_T)$, to encode the value assigned to v_T. We use *log encoding*, where Boolean variables correspond to bits of the binary representation of an unsigned integer. The variables of BDD now fall into two groups: variables in $bits_i(v_T)$ $(i = 0, \ldots, k)$ and other variables introduced for auxiliary purpose in the encoding of bounded model checking.

Next, we predetermine a variable ordering of BDD so that x precedes y if one of the following conditions holds:

- $x \in bits_0(v_T)$ and $y \notin bits_0(v_T)$;
- $x, y \in bits_0(v_T)$ and the bit assigned to x is more significant than that of y.

Finally, we simplify the BDD by the projection on $bits_0(v_T)$. Because for the longest interval computation, we are only interested in the values assigned to v_T in an initial state, there is no problem for ignoring other assignments. The projection can be done efficiently thanks to the variable ordering. Indeed, since all the variables in $bits_0(v_T)$ precede the other variables, it is sufficient to traverse the BDD and redirect each arc to $\top$ if the source node has some variable in $bits_0(v_T)$ and the target node has any other variable not in $bits_0(v_t)$.

6. Main Algorithm

In this section, we describe the main part of the longest interval computation and analyze its computational complexity. As depicted in **Fig. 3**, the key is that an interval is represented in BDD as a pair of paths that fork on the way to $\top$, having a certain condition. We will call such paths *forked paths*. The longest interval computation is then reduced to finding the longest interval among all pairs of forked paths.

We first describe the basic idea for efficiently finding the longest interval in a BDD using an example. Suppose that we have a target variable *packet.ipdst* of some network as given in Section 3; By applying the preprocessing, we now have the BDD in Fig. 3, which represents a set of values for *packet.ipdst*. Here, the values for the upper 19 bits are fixed due to the following LTL formula.

```
LTLSPEC
packet.ipdst19 = 0uh32_ab43a000 ->  F (
    location = bbrb_rtr | location = bbra_rtr)
```

The LTL formula states that if a destination is in the range

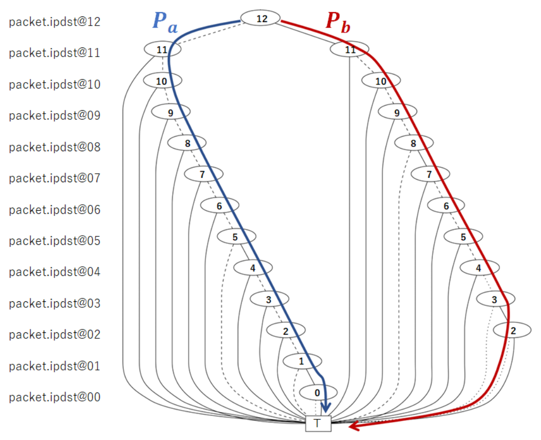

Fig. 3 Forked Paths, which is a pair of the paths marked in red and blue. A path from the root to $\top$ corresponds to a satisfying assignment if the number of complement arcs along the path is even. Solid arcs, dashed arcs, and dotted arcs represent HI arcs, LO arcs, and complement arcs, respectively.

Algorithm 1 The main part of the longest interval computation for a BDD f obtained after the preprocessing.

```
function LongestInterval($f$)
if $f$ is a terminal node or $f.done = true$ holds. then
    return;
end if
LongestInterval($f.lo$);
LongestInterval($f.hi$);
$I_L \leftarrow$ FindForkedPaths($f, L$);
$I_R \leftarrow$ FindForkedPaths($f, R$);
$I_M \leftarrow$ FindForkedPaths($f, M$);
$I \leftarrow$ the longest one among $I_L$, $I_R$, and $I_M$;
$f.dir \leftarrow$ the label of direction to which $I$ is included;
$f.done \leftarrow true$;
end function
```

171.67.160.0 to 171.67.191.255, then the packet some time in the future reaches *bbrb_rtr* or *bbra_rtr*.

For simplicity, the BDD nodes for the upper 19 bits are removed, and the remaining BDD nodes are only those for the lower 13 bits.

The long dashed arc, designated by P_a, and the long solid arc, designated by P_b, in Fig. 3 correspond to 171.67.160.35 and 171.67.177.11, respectively. The important observation here is that not only these addresses but also all addresses between them are accepted in the BDD. Actually, in each non-terminal node in P_a, either
(1) the HI child is a non-terminal node, or
(2) the HI child is $\top$ and the LO child is a non-terminal node.
For any bit sequence, S, of length 13 larger than P_a as integers, there must be a bit position at which S has value 1 while P_a has value 0. Either the case (2) above occurs at the highest one, i, among such positions or S has value 1 at bit 12. The former case means that if the value at bit i is 1, any values for the lower bits, i.e. from bit 0 to bit $i-1$, are allowed, and thus S is accepted in the BDD. For the latter case, one can also check, by a symmetric argument with respect to P_b, whether S is accepted.

It would be easy to see that for any BDD obtained after the preprocessing, if $[a, b]$ is maximal among all intervals such that all integers in $[a, b]$ are accepted in the BDD, then a corresponds to the path, P_a, in the BDD so that in each non-terminal node in P_a, either one of the conditions (1) and (2) above holds. The same observation applies to b and its corresponding path, P_b.

Our algorithm finds such P_a and P_b for each non-terminal node and among all intervals $[a, b]$ formed by such pairs (P_a, P_b) selects the longest one. We will call P_a and P_b the *left branch* and the *right branch*, and P_a and P_b *forked paths*.

The main part of the longest interval computation is shown in Algorithm 1. We summarize notations there. For each non-terminal BDD node f, the variable index is denoted by $f.idx$; the LO child and the HI child are denoted by $f.lo$ and $f.hi$, respectively. For efficient computation, f in addition has the two auxiliary fields: $f.done$ and $f.dir$, where $f.done$ holds a truth value, initially false, showing whether the computation for f is done, and $f.dir$ holds one of the labels L, R, and M, showing in which child of f, the current longest interval is included.

Algorithm 1, in each step of the recursion, computes the forked paths pairs, I_L and I_R, in the BDDs rooted by $f.lo$ and $f.hi$, respectively. We in addition compute one more forked paths pair, I_M, which consists of branches with one in each child, that is, the left branch goes along the LO arc of f and the right branch goes along the HI arc of f.

Algorithm 2 The left branch for I_M with starting node $curr$, where $curr$ is a non-terminal node.

```
if $curr.lo = \bot$ then
    return NIL;
else if $curr.lo = \top$ then
    $Fill(assign, curr, L, 0)$;
    return $assign$;
end if
$Fill(assign, curr, L, 1)$;
$prev \leftarrow$ NIL;
$curr \leftarrow curr.lo$;
while (1) $curr.hi$ is a non-terminal node or (2) $curr.hi = \top$ holds and $curr.lo$ is a non-terminal node. do
    if (1) holds. then
        $Fill(assign, curr, R, 1)$;
        $curr \leftarrow curr.hi$;
    else if (2) holds. then
        $Fill(assign, curr, L, 1)$;
        $prev \leftarrow curr$;
        $curr \leftarrow curr.lo$;
    end if
end while
if (3) $curr.hi = \top$ and $curr.lo = \bot$ then
    $Fill(assign, curr, R, 0)$;
else if (4) $curr.hi = \bot$ then
    if $prev \neq$ NIL then
        $Fill(assign, prev, R, 0)$;
    else
        return NIL;
    end if
end if
return $assign$;
```

The function FindForkedPaths computes I_L by reducing to trivial cases or the computation of I_M for some descendant of $f.lo$. That is, starting from the LO child of f, it repeats to move to the child indicated by the dir field of the current node, $curr$, until dir is M or the indicated child is a terminal node. For the trivial cases,
- if $curr.lo$ is $\bot$, it returns NIL, which means an empty interval;
- if $curr.lo$ is $\top$, it returns the pair of assignments such that one is the assignment with the i-th and later variables all assigned 0, where $i = curr.idx$, and the other is the assignment with the i-th and later variables all assigned 1.

For the computation of I_M, it tries to find the left branch and the right branch for the BDD rooted by $curr$. If it fails to find either branch, it returns NIL; Otherwise, it returns the pair of the left branch and the right branch.

We omit the computation of I_R because it is symmetrical to that of I_L.

For the computation of I_M, the left branch of I_M is computed as shown in Algorithm 2. It goes down BDD while filling an assignment of variables so that the assignment satisfies the necessary and sufficient condition for left branches described previously. For each step, we fill a part of the current assignment by invoking the subroutine *Fill*: given an array, *assign*, a BDD g, a label of direction *dir*, and $value \in \{0, 1\}$, the subroutine *Fill* sets $assign[f.idx]$ to 0 if $dir = L$ and to 1 if $dir = R$, and moreover, for each eliminated node g between f and the child of f indicated by *dir*, it sets $assign[g.idx]$ to *value*.

In the while statement, the algorithm attempts to find the left branch in the BDD pointed to by the LO arc of the starting node. In each step of this loop, there are only four cases (1) – (4) for the children of *curr*, as described in Algorithm 2. (1) and (2) mean that it is possible to move to one of the children so as not to violate the condition for left branches; (3) means that the left branch is found; and (4) means that it is necessary to backtrack to *prev*, because otherwise the condition for the left branch is violated. If (2) holds at least once in the while loop, then *prev.hi* must be $\top$ and the left branch must be found. Otherwise, *prev* is *NIL* and this case means that there is no left branch. The right branch of I_M can be computed in the same way, and hence omitted.

Theorem 1. Let f be a BDD obtained after the preprocessing described in Section 5. The longest interval in f can be computed in $O(nm)$ time, where two extra fields of constant size for each BDD node are assumed. n is the number of Boolean variables encoded from a target variable with log encoding and m is the number of nodes in f.

Proof. If f is $\top$, the longest interval is $[0, 2^n - 1]$. If f is $\bot$, it is an empty set. If f is a non-terminal node, we invoke the function LongestInterval with f, which can be done in $O(nm)$ time because computing FindForkedPaths and comparing I_L, I_M, and I_R are in $O(n)$ time. After that, invoking FindForkedPaths with f and $f.dir$, we obtain the longest interval. The whole computation time is $O(nm)$. □

7. Computational Framework

In this section, we present the whole picture of our computational framework and some remarks for each module. Only major modules are described and implementation details are omitted.

Since our framework utilizes bounded model checking in interactive manner, it includes the modules of an interactive command interface, a Boolean encoder, and a SAT solver as shown in **Fig. 4**. Our framework integrates bounded model checking with advanced features of a constrained counterexample generation and the longest interval computation by adding them as separate modules as shown in Fig. 4. Analysis of generated counterexamples is not limited to the longest interval computation. These major modules of our framework are described in more detail below.

Interactive Command Interface. This module can be taken from bounded model checkers but it is necessary to register additional commands such as those for the constrained counterexample generation and the longest interval computation.

The longest interval computation can be performed through a command-line interface as follows. Issue the command for the bounded model checking. If a counterexample is found, it is displayed as shown in Section 3. This can be repeated in different

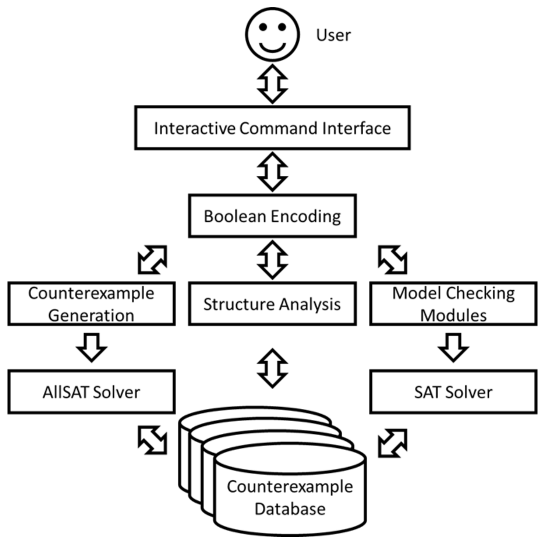

Fig. 4 Major modules of our computational framework.

settings until a desirable counterexample is found. Found counterexamples are associated with unique indices in order to distinguish between them.

Among such counterexamples, manually select one as a base counterexample and an integer variable in it as a target variable. Issue the command for the constrained counterexample generation with parameters including the selected base counterexample and the target variable. This also can be repeated, although unlike bounded model checking, a resulting set of constrained counterexamples (called a *counterexample database*) consumes much memory space in general and it would be hard to repeat many times. It would be convenient to make it possible to discard counterexample databases when needed.

Finally issue the command for the longest interval computation with parameters including a selected counterexample database. As a resulted interval, just two end values of the interval are displayed. As shown in Theorem 1, this computation would not become a computational bottleneck.

Boolean Encoding. This module can be taken from bounded model checkers without modification. Kripke structures and LTL formulas, which are described with some modeling language, are translated into propositional formulas such as CNFs to be able to utilize various solvers and data structures such as SAT solvers, AllSAT solvers, and BDDs. Based on the Boolean encoding, the results of analysis such as a counterexample and an interval of a specified variable are decoded and displayed in a human-readable form.

Bounded Model Checking and SAT Solver. These modules also can be taken from bounded model checkers without modification.

Counterexample Generation and AllSAT Solver. These modules provide functions such as the constrained counterexample generation and the projection of BDDs detailed in Section 5. In the constrained counterexample generation, a CNF fed into AllSAT solver is created using the Boolean encoding module. Like the relation between the model checking module and SAT solver, the constrained counterexample generation module is independent of AllSAT solver, and any AllSAT solver can be utilized as long as it supports a predetermined interface.

Counterexample Databases. This module provides data representations of generated counterexamples, called *counterexample database*, and basic operations over them. Since the number of counterexamples is likely to blow up exponentially, our main algorithm for the longest interval computation utilize BDDs in order to efficiently represent and manipulate counterexamples.

Structure Analysis. This module is in charge of various analyses based on generated counterexamples including the main part of the longest interval computation detailed in Section 6. Although this paper considers only the range of values for the initial state and fixes all the other variables' values, it would be interesting to extend this to be able to allow a non-initial state and fix only some variables' values. It would be also interesting to consider analyses based on efficient queries to BDDs. An example is a membership of a user-specified interval, that is, given a user-specified interval for a target variable and a base counterexample, to determine whether substituting the value of the target variable in the base counterexample with any integer of the interval remains a counterexample.

8. Experiments

In this section we present experimental results of the longest interval computation.

Implementation. We implement the longest interval computation based on constrained counterexample generation on top of NuSMV2.5.4 [5], where CUDD 2.4.1 and MiniSat 2.2.0 are linked. For the generation part, we utilize *BDD solver*, a variant of AllSAT solvers [22]. For counterexample databases, we utilize the same BDD library (CUDD 2.4.1) attached to NuSMV2.5.4.

Environments. All experiments are conducted on a computer with an Intel ® Core TM i7-4600U 2.10 GHz processor and 16 GB of RAM, running Ubuntu 18.04.4 LTS with gcc compiler 5.3.1.

Stanford Network. Our method is evaluated with a real network dataset obtained from the backbone network in Stanford university [*1] and also with a dataset randomly modified from the Stanford network so that each transfer rule regarding IP destinations is changed in a certain probability but all LTL specifications are unchanged.

The Stanford network has been used to evaluate network verification tools [13], [15]. It consists of 16 switches with 58 interfaces in total. Packets are forwarded by the switches based on their header bits; the forwarding decision is made with 88 bits including IP address, TCP port number, and so on. The Stanford backbone network is modeled with 94 binary variables: 88 bits for packet header and 6 bits for switch interfaces. The dataset can be converted into NuSMV models by the bundled script (`nu_smv/nu_smv_generator.py`).

Network verification is usually used to check conformance with operational policies, but the Stanford dataset includes no material that can be used to specify policies. In the experiments, we try to find counterexamples under a hypothetical policy — all packets should reach their destinations without being filtered out inside the network. We randomly choose 764 interface pairs and examine their reachability. The reachability properties are converted into CNF instances with the shortest bounds such that counterexamples are found; the bounds are 2 for internal interfaces, while they are 6 for external ones (note that reachability related to special addresses, e.g., 0.0.0.0, 255.255.255.255, and 224.*.*.*, are ignored, because counterexamples are not found until bound 100).

Experimental Result on Original Stanford Network. Since all LTL formulas are reachability properties regarding IP destinations, we select *packet.ipdst* as a target variable, and for each LTL formula, we obtain a base counterexample by performing bounded model checking.

For all LTL formulas, the maximum number of clauses and variables in CNF fed into AllSAT solver are 71,358,640 and 4,980, respectively. The maximum time for the constrained counterexample generation is 0.06 s, and the maximum time for the main part of the longest interval computation is less than 0.01 s. Picking up several LTL formulas, we inspect constructed BDDs and the Stanford network carefully, and it turns out that all packets such that their destination addresses are in the ranges indicated by LTL formulas to be inspected and the other fields such as source addresses and port numbers are fixed according to a counterexample computed in advance fail to reach specified locations. This is because no constraint is imposed on the Boolean variables corresponding to the inspection ranges in the Stanford network. If a few upper bits are fixed by LTL specification, many counterexamples are reported because all the lower bits are not affected, however this is a bit extreme situation due to the lack of policies.

Experimental Result on Randomly Modified Stanford Network. Considering the previous experimental result, we modify the Stanford network for each LTL formula so that many addresses in the inspection range of the LTL formula are likely to be forwarded to different destinations. More concretely, each forwarding rule regarding IP destinations is changed in probability 0.3 so that some (but not necessarily all) addresses of the inspection range of the LTL formula matches the replaced transfer rule. For example, for the LTL specification given in Section 6, the following rule

```
packet.ipdst28 = 0uh32_4441a870: drop;
```

is changed in probability 0.3 so that the upper 19 bits of the word constant 441a870 given in hex are fixed so as to match the antecedent of the LTL specification and the other bits are unchanged. The resulting rule is as follows.

```
packet.ipdst28 = 0uh32_ab43a870: drop;
```

This means that the range from 68.65.168.112 to 68.65.168.127 is changed to the range from 171.67.168.112 to 171.67.168.127, and hence some packets satisfying the antecedent of the LTL specification are dropped in the resulting rule. In the current experiment we suppose that such a change is caused by a result of mis-configuration.

We evaluate our method with the modified Stanford networks. **Figure 5** shows cummulative running times of all randomly modified stanford networks, where bmc denotes bounded model checking and allsat+interval denotes our method (i.e., the com-

[*1] https://bitbucket.org/peymank/hassel-public/

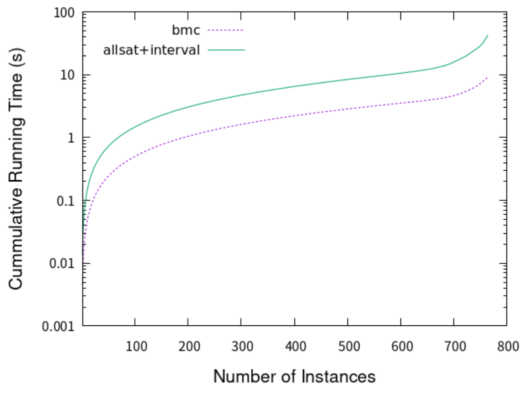

Fig. 5 Cummulative running times of randomly modified stanford networks, where the y-axis is in log-scale.

bination of allsat solving and longest interval computation over BDD). Ranges of values that result in counterexamples are confirmed to be computable with small overheads compared to bounded model checking.

Other results are summarized below. For all LTL formulas, the maximum number of clauses and variables in CNF fed into AllSAT solver are 75,563,312 and 16,361, respectively. The maximum time for the constrained counterexample generation is 1.50 s, and the maximum time for the main part of the longest interval computation is less than 0.01 s. Picking up several LTL formulas and inspecting carefully, we confirm that as intended, not all addresses in the inspection ranges are present in the constructed BDDs. We confirm that 8,248,830 counterexamples are generated in 1.41 s and among them, the longest interval of length 6,160,384 is reported in less than 0.01 s. This case achieves the maximum number of counterexamples obtained after the constrained counterexample generation and also the maximum length of interval among all LTL formulas.

9. Conclusion

In this paper, we studied counterexample analysis in model checking, aiming at an effective explanation of failure. We formulated such a range of possible values (called interval) naturally identified with integers in a general setting to allow not only network models but also arbitrary models that come from diverse domains. We presented a practical method for computing the longest interval problem given a counterexample (called a base counterexample) and a variable (called a target variable) on which the interval is computed. This computation is separated into two parts: the preprocessing part and the main part. In the preprocessing part, constrained counterexamples are generated using AllSAT solver. Since the number of such counterexamples is likely to blow up exponentially, our method utilized a well accepted space-efficient data structure BDD in the AllSAT solving. In the main part, our method searches the longest interval while traversing the BDD obtained after the preprocessing. We proved that the main part can be computed in time proportional to the product of the number of Boolean variables encoded from a target variable and the number of nodes in the BDD. Our method was evaluated with a real network dataset and its randomly modified dataset. We confirmed that about 8 millions of counterexamples were generated in 1.61 s and among them, the longest interval of length about 600 millions was reported in less than 0.01 s. We also presented a generic computational framework that integrated bounded model checking with advanced functions such as constrained counterexample generation, counterexample databases, and AllSAT solvers to allow not only the longest interval computation but also other analyses based on the same approach. It would be desirable that we could compute all intervals and present them in a compact form such as a bit pattern, however this task is more involved, and remains a future work.

Acknowledgments This work is supported by JSPS KAKENHI Grant Number 17K17725.

References

[1] Akers, S.B.: Binary Decision Diagrams, *IEEE Trans. Comput.*, Vol.27, No.6, pp.509–516 (online), DOI: 10.1109/TC.1978.1675141 (1978).

[2] Biere, A., Cimatti, A., Clarke, E.M. and Zhu, Y.: Symbolic Model Checking Without BDDs, *Proc. 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems, TACAS '99*, pp.193–207, Springer-Verlag (1999) (online), available from ⟨http://dl.acm.org/citation.cfm?id=646483.691738⟩.

[3] Bryant, R.E.: Graph-Based Algorithms for Boolean Function Manipulation, *IEEE Trans. Computers*, Vol.C-35, No.8, pp.677–691 (online), DOI: 10.1109/TC.1986.1676819 (1986).

[4] Cimatti, A.: *Modeling and Verification of Parallel Processes*, chapter Industrial Applications of Model Checking, pp.153–168, Springer-Verlag New York, Inc. (2001) (online), available from ⟨http://dl.acm.org/citation.cfm?id=766794.766801⟩.

[5] Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R. and Tacchella, A.: NuSMV 2: An Open-Source Tool for Symbolic Model Checking, *Computer Aided Verification*, Brinksma, E. and Larsen, K.G. (Eds.), pp.359–364, Springer Berlin Heidelberg (2002).

[6] Clarke, E.M., Emerson, E.A. and Sifakis, J.: Model Checking: Algorithmic Verification and Debugging, *Comm. ACM*, Vol.52, No.11, pp.74–84 (online), DOI: 10.1145/1592761.1592781 (2009).

[7] Clarke, Jr., E.M., Grumberg, O. and Peled, D.A.: *Model Checking*, MIT Press (1999).

[8] Groce, A.: Error Explanation with Distance Metrics, *Tools and Algorithms for the Construction and Analysis of Systems*, Jensen, K. and Podelski, A. (Eds.), pp.108–122, Springer Berlin Heidelberg (2004).

[9] Groce, A. and Kroening, D.: Making the Most of BMC Counterexamples, *Proc. 2nd International Workshop on Bounded Model Checking (BMC 2004)*, Vol.119, No.2, pp.67–81 (online), DOI: 10.1016/j.entcs.2004.12.023 (2005).

[10] Groce, A. and Visser, W.: What Went Wrong: Explaining Counterexamples, *Model Checking Software*, Ball, T. and Rajamani, S.K. (Eds.), pp.121–136, Springer Berlin Heidelberg (2003).

[11] Huang, J. and Darwiche, A.: Using DPLL for Efficient OBDD Construction, *Theory and Applications of Satisfiability Testing*, Hoos, H.H. and Mitchell, D.G. (Eds.), pp.157–172, Springer Berlin Heidelberg (2005).

[12] Huang, J. and Darwiche, A.: The Language of Search, *J. Artif. Int. Res.*, Vol.29, No.1, pp.191–219 (2007) (online), available from ⟨http://dl.acm.org/citation.cfm?id=1622606.1622613⟩.

[13] Inoue, T., Chen, R., Mano, T., Mizutani, K., Nagata, H. and Akashi, O.: An Efficient Framework for Data-plane Verification with Geometric Windowing Queries, *IEEE ICNP*, pp.1–10 (2016).

[14] Kazemian, P., Varghese, G. and McKeown, N.: Header Space Analysis: Static Checking for Networks, *Proc. 9th USENIX Conference on Networked Systems Design and Implementation, NSDI'12*, p.9, USENIX Association (2012) (online), available from ⟨http://dl.acm.org/citation.cfm?id=2228298.2228311⟩.

[15] Kazemian, P., Varghese, G. and McKeown, N.: Header Space Analysis: Static Checking for Networks, *USENIX NSDI*, pp.113–126 (2012).

[16] Knuth, D.E.: *The Art of Computer Programming, Volume 4, Fascicle 1: Bitwise Tricks & Techniques; Binary Decision Diagrams*, Addison-Wesley Professional, 12th edition (2009).

[17] Lee, C.Y.: Representation of Switching Circuits by Binary-Decision Programs, *Bell System Technical Journal*, Vol.38, No.4, pp.985–999 (online), DOI: 10.1002/j.1538-7305.1959.tb01585.x (1959).

[18] Lopes, N.P., Bjørner, N., Godefroid, P., Jayaraman, K. and Varghese, G.: Checking Beliefs in Dynamic Networks, *Proc. 12th USENIX Conference on Networked Systems Design and Implementation, NSDI'15*,

pp.499–512, USENIX Association (2015) (online), available from ⟨http://dl.acm.org/citation.cfm?id=2789770.2789805⟩.

[19] McGeer, R.: Verification of switching network properties using satisfiability, *2012 IEEE International Conference on Communications* (*ICC*), pp.6638–6644 (online), DOI: 10.1109/ICC.2012.6364863 (2012).

[20] Prasad, M.R., Biere, A. and Gupta, A.: A Survey of Recent Advances in SAT-based Formal Verification, *Int. J. Softw. Tools Technol. Transf.*, Vol.7, No.2, pp.156–173 (online), DOI: 10.1007/s10009-004-0183-4 (2005).

[21] Qadir, J. and Hasan, O.: Applying Formal Methods to Networking: Theory, Techniques, and Applications, *IEEE Communications Surveys Tutorials*, Vol.17, No.1, pp.256–291 (online), DOI: 10.1109/COMST.2014.2345792 (2015).

[22] Toda, T. and Soh, T.: Implementing Efficient All Solutions SAT Solvers, *J. Exp. Algorithmics*, Vol.21, pp.1.12:1–1.12:44 (online), DOI: 10.1145/2975585 (2016).

[23] van den Berg, L., Strooper, P. and Johnston, W.: An Automated Approach for the Interpretation of Counter-Examples, *Proc. Workshop on Verification and Debugging* (*V&D 2006*), Vol.174, No.4, pp.19–35 (online), DOI: 10.1016/j.entcs.2006.12.027 (2007).

[24] Vizel, Y., Weissenbacher, G. and Malik, S.: Boolean Satisfiability Solvers and Their Applications in Model Checking, *Proc. IEEE*, Vol.103, No.11, pp.2021–2035 (online), DOI: 10.1109/JPROC.2015.2455034 (2015).

[25] Zhang, S., Malik, S. and McGeer, R.: Verification of Computer Switching Networks: An Overview, *Automated Technology for Verification and Analysis*, Chakraborty, S. and Mukund, M. (Eds.), pp.1–16, Springer Berlin Heidelberg (2012).

Takahisa Toda received his B.E. degree in integrated human studies, and his M.E. and his Ph.D. degrees in human and environmental studies from Kyoto University, Japan, in 2004, 2006, and 2012, respectively. He is an associate professor of Graduate School of Informatics and Engineering, the University of Electro-Communications. He was an ERATO Researcher with the Japan Science and Technology Agency, from 2012 to 2014. His research interests are experimental analysis and applications of discrete algorithms. He is a member of ACM, IPSJ, JSAI and IEICE.

Takeru Inoue received his B.E. and M.E. degrees in engineering science and his Ph.D. degree in information science from Kyoto University, Japan, in 1998, 2000, and 2006, respectively. He is a Senior Researcher with Nippon Telegraph and Telephone Corporation (NTT) Laboratories. He was an ERATO Researcher with the Japan Science and Technology Agency, from 2011 to 2013. His research interests widely cover algorithmic approaches in computer networks. He is a member of IEEE and IEICE.