An Automatic Solution to Checking Compatibility between Routing Metrics and Protocols

Chang Liu

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Engineering

Yaling Yang, Chair
Haibo Zeng
Yiwei Thomas Hou

December 2, 2015
Blacksburg, Virginia

Keywords: Routing Metrics, Routing Protocols, Software Verification

An Automatic Solution to Checking Compatibility between Routing Metrics and Protocols

Chang Liu

(ABSTRACT)

Routing metrics are important mechanisms to adjust routing protocols' path selection according to the needs of a network system. However, if a routing metric design does not correctly match a particular routing protocol, the protocol may not be able to find an optimal path; routing loops can be produced as well. Thus, the compatibility between routing metrics and routing protocols is increasingly significant with the widespread deployment of wired and wireless networks. However, it is usually difficult to tell whether a routing metric can be perfectly applied to a particular routing protocol. Manually enumerating all possible test cases is very challenging and often infeasible. Therefore, it is highly desirable to have an automatic solution so that one can avoid putting an incompatible combination of routing metric and protocol into use. In this thesis, the above issue has been addressed by developing two automated checking systems for examining the compatibility between real world routing metric and protocol implementations. The automatic routing protocol checking system assumes that some properties of routing metrics are given and the system's job is to check if a new routing protocol is able to achieve optimal, consistent and loop-free routing when it is combined with metrics that hold the given metric properties. In contrast to the protocol checking system, the automatic routing metric checking system assumes that a routing protocol is given and the checking system needs to verify if a new metric implementation will be able to work with this protocol. Experiments have been conducted to verify the correctness of both protocol and metric checking systems.

Acknowledgments

Some graduates write terse acknowledgments thanking exactly the select few who sped them along their way. Others ramble on, trying to give thanks to everyone who helped throughout the years of toil, only to glance at their dissertation two weeks later and realize they left out key individuals.

You have probably already guessed this is more like the second than the first. This work took me a long time, and I had a lot of help. I would like to express the most gratefulness to my advisor Yaling Yang, who shepherded this work through the lengthy stages of implementation, debugging, experimentation and writing. She held me to the highest standards of quality and accuracy, and any value in my work is thanks to those high standards. I am also grateful for her frank opinions and advice, and for encouraging me when I was frustrated. Without her instruction and brainstorm, I may not be able to overcome so many difficulties and eventually finish this research project.

Life, however, is not all about research, and I was lucky enough to have many good friends to remind me of that.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

In general, a routing system includes two critical components: a routing metric and a routing protocol. The routing metric evaluates path quality based on various network needs and the routing protocol computes the optimal path based on the path quality evaluation from the routing metric. To ensure proper operation, routing protocols usually have some requirements on routing metric designs. If a wrong type of routing metrics is used with a routing protocol, unexpected problems, such as suboptimal paths and routing loops, may be yielded.

To understand the impacts, we studied a number of routing metric implementations. Code block 1.1 shows a typical metric implementation generated by our random metric synthesizer. In Code block 1.1, the `weight` function returns its evaluation of a `path` based on two values `v0` and `v1` assigned to each link constituting this `path`. Besides, `tp[0]` is a predefined tunable parameter subject to $0 \leq tp[0] \leq 1$ while `alpha` is an array of predefined coefficients whose summation equals 1. Obviously, it is difficult to quickly tell whether this implementation works correctly with a specified

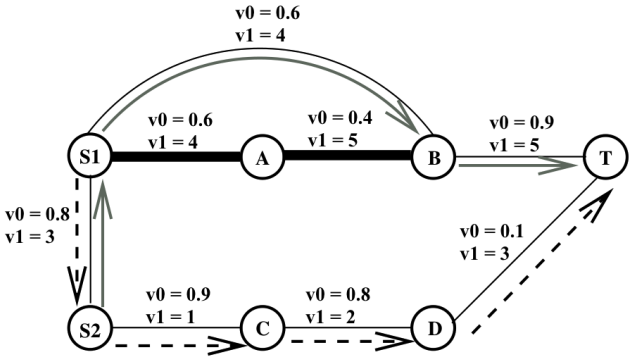


Figure 1.1: Counterexample topology for Code block 1.1.

routing protocol in the real world.

According to our experiments, if this metric is applied to link-state routing protocol, it is possible to create routing loops and suboptimal paths as shown in Figure 1.1. Assume that a link-state routing protocol uses Dijkstra's algorithm for path selection. To calculate the minimum weight path from S_1 to T in Figure 1.1, `tp[0]` and `alpha` are randomly set to 0.8 and $\{0.69, 0.31\}$, respectively. As a result, the routing system wrongfully identifies the "minimum" weight path as $S_1 \to S_2 \to C \to D \to T$ (weight = 0.5234) while the real minimum weight path is actually $S_1 \to B \to T$ (weight = 0.3514). This is because when running Dijkstra's algorithm at node S_1, the minimum weight path calculated from node S_1 to node B is found to be $S_1 \to A \to B$ (weight = 0.2824) rather than $S_1 \to B$ (weight = 0.31). Due to the greedy nature of Dijkstra's algorithm, link $S_1 \to B$ is discarded prematurely from Dijkstra's algorithm's further calculation even if $S_1 \to B \to T$, the real path of minimum weight, has smaller weight than $S_1 \to A \to B \to T$ does.

In addition, with this metric, the link-state routing system also creates routing loops and disconnected network in this example. At S_2, the routing system computes that the minimum weight path to T is $S_2 \to S_1 \to B \to T$. Since node S_1 has the incorrect minimum weight path $S_1 \to S_2 \to C \to D \to T$, when setting up the routing tables

for all nodes in the topology at the very beginning, traffic from S_1 selects S_2 as the forwarding node while traffic from S_2 selects S_1 as the counterpart simultaneously. Therefore, the route reply message for these two traffics will correspondingly modify the routing table for S_1 and S_2, respectively. At last, a routing loop is formed between S_2 and S_1 such that all packets will be forwarded between S_1 and S_2 back and forth recursively. Due to this loop, S_1 and S_2 are virtually disconnected from T even though they are physically connected.

Code block 1.1: A randomly generated routing metric.

```
float weight(const Path &path) {
  float ret(0), max(0), res[value_size(path)] = {0};
  for (std::size_t i(0); i != path.size() - 1; ++i)
    if (path[i].v0 < 0.5)
      res[0] += (path[i + 1].v0 + path[i].v0) * tp[0]);
    else
      res[0] += (path[i + 1].v0 - path[i].v0) * (1 - tp[0]));
  for (std::size_t i(0); i != path.size(); ++i)
    ++item0[path[i].v1];
  for (std::size_t i(0); i != item0.size(); ++i)
    if (item0[i] > max)
      max = item0[i];
  res[1] = max;
  for (std::size_t i(0); i != res.size(); ++i)
    ret += alpha[i] * res[i];
  return ret;
}
```

1.2 Existing methods in identifying incompatibility

Due to the potentially serious impacts of incompatibility between routing metrics and protocols, a few methods have been put forward to solve the compatibility issue. These methods include simulation-based approach, protocol verification approach, and routing metric property analysis. Since large-scale real-world tests are very

costly and therefore unrealistic, simulation-based approach is often used to evaluate metric designs empirically. By running a large number of test cases, a metric designer can tell whether a design is likely to cause trouble for a specified routing protocol. However, even if the metric passes all the tests, still, the designer cannot guarantee that the metric is flawless when it is combined with a protocol in the real world. In other words, a metric design may fail the real-world tests because its designer is unable to consider all possible network topologies during the simulation.

Because of the limitation of simulation-based approach, numerous literatures focused on protocol verification works, who aim at examining an entire routing system design through formal verification. For example, Bharti and Kumar generalized in their survey [1] a series of routing system errors and corresponding verification methods. Due to the code complexity of real routing system implementation, using formal verification techniques to verify the correctness and functionality of an entire system implementation is usually prohibitively complex. Hence, most of the protocol verification works aim at verifying an abstract model of a routing system. Manual effort is needed to build such an abstract model and there is no guarantee that the actual implementation is a faithful realization of the abstract model, rendering these protocol verification methods of limited use in real-world scenarios.

Aiming at finding better methods to verify metric and protocol combination, some theoretical efforts have been made in recent years to study routing metric properties that make routing metrics and protocols compatible (or incompatible). These research efforts have identified and proved rigorously the exact mathematical properties a metric needs to work with popular routing protocols. For example, in [2], Yang stated that whether a link-state routing protocol can satisfy the optimality, consistency and loop-freeness requirements depends on different types of isotonicity and monotonicity properties of its routing metric. Lu and Wu systematically analyzed in [3] the relationship between routing metrics and opportunistic routing protocols, which demands a few essential metric properties so as to make the opportunistic routing protocol designs optimal, loop-free and consistent. In [4], Li et al.

introduced several common geographic routing metric design requirements, lacking of which may possibly produce routing loops and unreachable nodes. Han enumerated in [5] a few typical broadcast tree construction algorithms and their demands on routing metric properties.

While all of these existing theoretical works are very rigorous, manual efforts are still needed to theoretically prove if a given unknown metric holds certain required properties. In addition, if a new routing protocol is designed or some implementation of an unknown routing protocol is given, manual theoretical analysis and proofs similar to the ones in [2], [3], [4] and [5] must also be conducted to figure out the metric properties demanded by the routing protocols. All these manual efforts are nontrivial. Hence, existing theoretical works are still far from making the job of compatibility checking easy.

To reliably reduce the cases of possible incompatible designs in practice, we developed automatic compatibility checking approaches for real world routing protocol and metric implementations. Specifically, we divide the compatibility checking in two cases, *protocol checking* and *metric checking*. In *protocol checking*, we assume that a set of routing metric properties is given and the designer's job is to check if an unknown routing protocol will be compatible with metrics that hold these properties. In *metric checking*, we assume that a routing protocol is given and a network designer needs to check if an unknown metric is able to work with this protocol. For each case, we provide a fully automated checking method without any manual effort from the network designer. Our checking method leverages the recent theoretical development on routing algebra and model checking.

Compared with the existing approaches, our method has the following benefits. Unlike existing theoretical analysis, our method requires no manual mathematical analysis on the metrics or routing protocols. Given a protocol implementation, it is able to automatically diagnose whether the protocol is compatible with a set of properties satisfied by a group of routing metrics; given a metric implementation, it can

automatically identify if the metric is compatible with certain routing protocols. In addition, unlike the protocol verification method, which is too complex to be carried out over real implementations, our method can be used over real implementations by just checking the interactions between routing metrics and protocols, which is usually of much smaller size compared to the entire routing system implementation.

1.3 Thesis overview

The contents of each chapter are described as follows.

Chapter 2 describes the theoretical foundation for the design of our automatic checking systems. Chapter 3 states the conception and implementation of the automated protocol checking method. Chapter 4 goes over the basic principle and implementation of our metric checking procedure. Chapter 5 precisely analyzes the experimental results. Finally, chapter 6 concludes our contribution and future work.

Chapter 2

Theoretical foundation of routing compatibility

Table 2.1: Relationship between path weight structure and routing protocols.

Routing protocols	Optimality	Consistency	Loop-freeness
Dijkstra's algorithm + hop-by-hop routing	right-isotonicity + right-monotonicity	right-isotonicity + right-monotonicity + strictly left-isotonicity	right-isotonicity + right-monotonicity + strictly left-isotonicity
Distributed Bellman-Ford algorithm + hop-by-hop routing	left-isotonicity + left-monotonicity	left-monotonicity	left-monotonicity
Greedy algorithm + forward building hop-by-hop routing	N/A	odd symmetry + transitivity + strict order + source independence	odd symmetry + transitivity
CGF algorithm + forward building hop-by-hop routing	N/A	N/A	odd symmetry + transitivity + local minimum free

The design of our automatic checking system is based on the compatibility theories for routing metrics and protocols. In the following, we briefly introduce these theories.

In existing literatures, the compatibility between routing metrics and protocols is usually described with routing algebra. In routing algebra, a routing system is defined

as a tuple

$$A = (S, w(\cdot), \preceq),$$

where S is the set of signatures describing the characteristics of a particular topology, such as some link's capacity, energy consumption, etc. The symbol $w(\cdot)$ denotes a weight function of evaluating these signatures of network topologies and is essentially the abstract representation for a routing metric design. The symbol $\prec$ refers to the preference relation in which $w(T_1) \prec w(T_2)$ means topology T_1 is better than T_2 for routing. A binary operator $\oplus$ joins two topologies into a single one by combining all their nodes and edges.

The compatibility study using above routing algebra usually focuses on whether a specific routing metric and protocol combination can achieve the following three targets: optimality, consistency, and loop-freeness [2]. These three targets are illustrated as follows:

- **Consistency:** If any node v_1 selects path $p(v_1, v_n) = \langle v_1, v_2, \cdots, v_n \rangle$ to forward traffic to any destination node v_n, each node v_i on path $p(v_1, v_n)$ should choose the sub-path $p(v_i, v_n) = \langle v_i, v_{i+1}, \cdots, v_n \rangle$ of $p(v_1, v_n)$ to forward traffic to v_n. where $1 < i < n$. For example, node v_2 should choose the path $p(v_2, v_n) = \langle v_2, v_3, \cdots, v_n \rangle$ for routing.

- **Optimality:** Optimality requires that each path selected by a routing protocol between any pair of nodes in a connected network should has the best weight compared with any other path between this pair of nodes.

- **Loop-freeness:** Given any connected network, a routing protocol is loop-free if it does not yield any packet forwarding loop when it converges. Any routing protocol short of loop-freeness, which is the most important requirement, is not usable because a packet will be recursively forwarded between a number of nodes in a routing loop.

Whether a routing protocol satisfies these three targets highly depends on whether

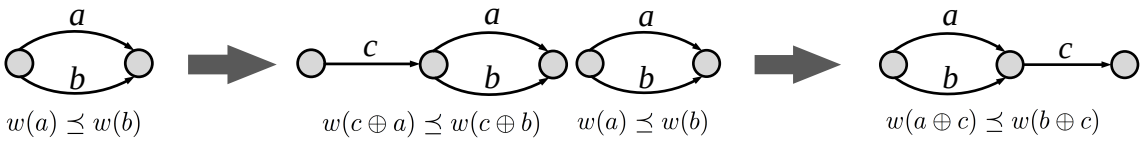
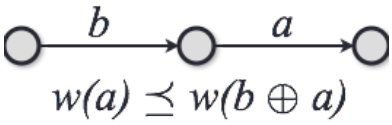


Figure 2.1: Left-isotonicity for paths

Figure 2.2: Right-isotonicity for paths



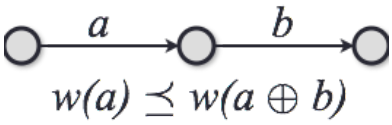


Figure 2.3: Left-monotonicity for paths

Figure 2.4: Right-monotonicity for paths

the routing metric, captured by the weight function $w(\cdot)$, has the necessary properties. Existing works have identified the required metric properties for many routing protocols to achieve each of these goals. Table 2.1 shows some of these results. To illustrate the nature of these metric properties, we present some of them as examples in the following.

In [2], the authors described several critical properties for routing metrics (a.k.a. the weight function $w(\cdot)$) used with link-state or path vector routing protocols. These properties, illustrated in Fig. 2.1 to Fig. 2.4, are defined as follows.

- **Left-(Right-)isotonicity for paths:** A weight function of paths $w(\cdot)$ is left-isotonic for any path a, b and c if $w(a) \preceq w(b)$ always implies $w(c \oplus a) \preceq w(c \oplus b)$. Likewise, $w(\cdot)$ is right-isotonic for paths if $w(a) \preceq w(b)$ always implies $w(a \oplus c) \preceq w(b \oplus c)$.
- **Left-(Right-)monotonicity for paths:** A weight function of paths $w(\cdot)$ is left-monotonic for any path a and b if $w(a) \preceq w(b \oplus a)$ holds. Similarly, $w(\cdot)$ is right-monotonic for paths if $w(a) \preceq w(a \oplus b)$ holds.

In [4], Li and Yang identified the necessary properties for geographic routing metrics. Specifically, in [4], the selection criteria for geographic routing is modelled as a com-

bination of weight function $w(\cdot)$ and threshold value ϕ. Defined over links, weight function $w(l(u,v))$ can be computed based on the geographical positions of node u and v. If an outward link's $w(\cdot)$ weight is the lightest and also lighter than ϕ, the link is selected as the next hop in forwarding. In order to ensure loop-freeness and consistency for different kinds of geographic routing, the weight function $w(\cdot)$ must have some of the following five properties.

- **Odd symmetry:** A weight function $w(\cdot)$ is odd symmetric in terms of ϕ if for $\forall l(v,u)$ and $l(u,v)$, $w(l(u,v)) \prec \phi$ implies $w(l(v,u)) \succ \phi$, and $w(l(u,v)) = \phi$ implies $w(l(v,u)) = \phi$.
- **Transitivity:** A weight function $w(\cdot)$ is transitive if for $\forall l(u,v)$ and $l(v,w)$, $w(l(u,v)) \prec \phi$ and $w(l(v,w)) \prec \phi$ implies $w(l(u,w)) \prec \phi$.
- **Strict order:** There do not exist cases where two links $l(u,v)$ and $l(u,w)$ from node u satisfy $w(l(u,v)) = w(l(u,w))$.
- **Source independence:** A weight function $w(\cdot)$ is source independent if for $\forall l(u,v)$, the value of $w(l(u,v))$ is not related to the source node.
- **Local minimum free:** For $\forall l(u,d)$, there does not exist any other node x satisfying $w(l(u,x)) \preceq w(l(u,d))$ where d is the destination node.

These existing theoretical results have many limitations if directly used in practice. Manually proving that a metric lacks one of these metric properties would require a designer to painstakingly come up with a counterexample that demonstrates the property does not hold for any possible topology. Since these theoretical works are unable to cover all routing protocols, they cannot tell if an unclear routing protocol will be happy with metrics that have some known properties. Fortunately, we have developed effective methods to automatically verify the compatibility between routing metrics and routing protocols based on the results of above existing theoretical works. In the next two sections, we will introduce our automatic checking systems.

Chapter 3

Routing Protocol Checking

3.1 Preference relation based protocol checking method

Assuming that some routing metric properties are given, the goal of protocol checking is to find out whether a routing protocol implementation is able to achieve its target (i.e., optimality, consistency or loop-freeness) when it is combined with any metric that has the given metric properties. Since a routing protocol's implementation may be very large and the potential number of valid metrics that satisfy the given metric properties can be infinite, directly analyzing the combination of the routing protocol with every valid routing metric is not feasible. Traditional program analysis techniques, like symbolic execution, also do not work because they lack scalability for large programs. In addition, these techniques require knowledge of the entire code, including both the protocol and the metric. Yet, the protocol checking problem faces an infinite number of possible metric designs and implementations. To solve this challenge, we propose a novel protocol checking mechanism that analyzes possible interactions between routing metrics and protocols, which can be described by a

finite number of combinations for any given topology.

3.2 Overall protocol checking procedure

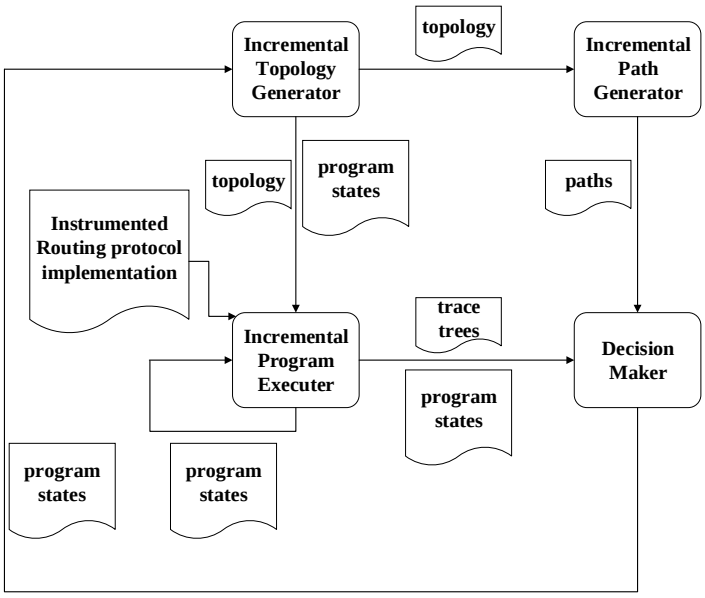


Figure 3.1: Protocol checking procedure.

Generally, given a routing protocol program, a preprocessing procedure first analyzes its control flow graph (CFG) and modifies it by instrumenting all essential labels and checkpoints. Then the instrumented program is sent to the automatic checking system, which consists of an Incremental Topology Generator, an Incremental Path Generator, an Incremental Program Executer and finally a Decision Maker, as shown in Figure 3.1. The Incremental Topology Generator automatically derives new topologies, collects program states returned by the decision maker after previous checking of the old topology. Once a new topology is derived from an old one, it will be submitted to an Incremental Program Executer and Path Genera-

tor, respectively. Having inherited from the old topology all possible paths between a pair of source and destination nodes, the Incremental Path Generator adds new paths containing an new edge (or node) added in the new topology. Once the Incremental Program Executer sees the newly generated topology, it inherits the program states from the previous topology checking, traverses all possible program traces and builds corresponding program trace trees for different starting and destination nodes in the topology. At last, a Decision Maker analyzes the trace trees and the alternative paths identified by the Incremental Path Generator to find counterexamples to a number of expected requirements regarding several essential predefined properties. If no counterexamples can be identified when the order of testing topologies reaches a predefined threshold, we can assume that the protocol satisfies all requirements needed.

3.3 Preprocessing

The entire preprocessing procedure includes a `Path` Variables Marker, a Preprocessor, an Instrumenter and a Postprocessor, as shown in Figure 3.2. With the help of LLVM infrastructure [6], the `Path` Variables Marker translates the input routing protocol program into LLVM intermediate representation (IR) and finds out the names of all variables with datatype `Path`. Once the Preprocessor obtains these names, it goes through the original input program and in sequence adds labels and checkpoint functions before each variable's appearance, as shown in Figure 3.3. Then this program is sent to the Instrumenter, which also utilizes LLVM infrastructure. When analyzing each instruction in the basic blocks, the Instrumenter only cares about `load` and `store` instructions involving variables with datatype `Path` and consequently it marks those irrelevant *checkpoint* functions to be removed and dumps out the renumbering information of labels and *checkpoint* functions. Apart from the search for significant `Path` variables, the Instrumenter also records the names and datatypes of all living variables before each appearance of *checkpoint* functions. These names and datatypes

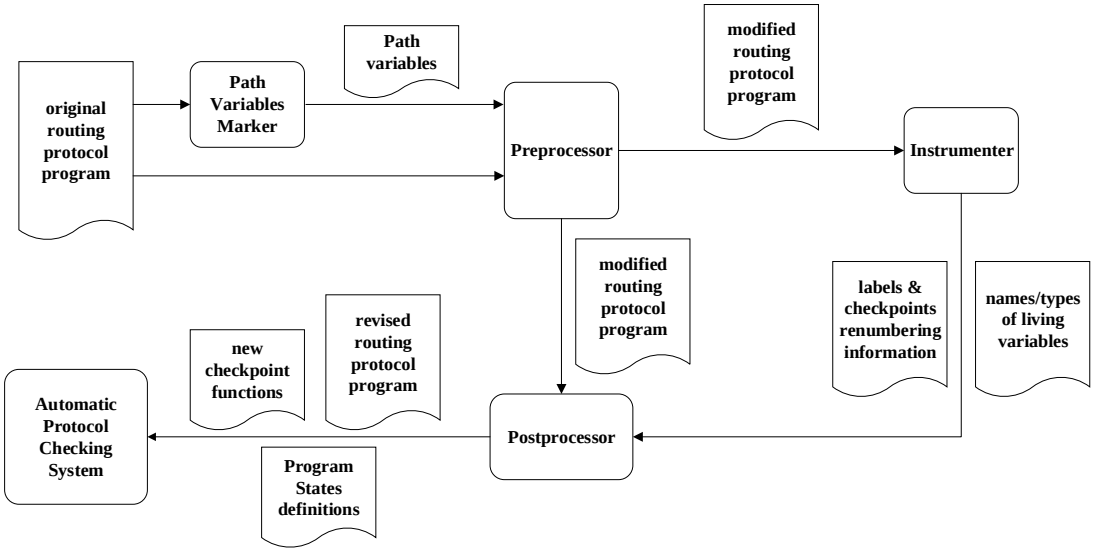


Figure 3.2: Preprocessing procedure.

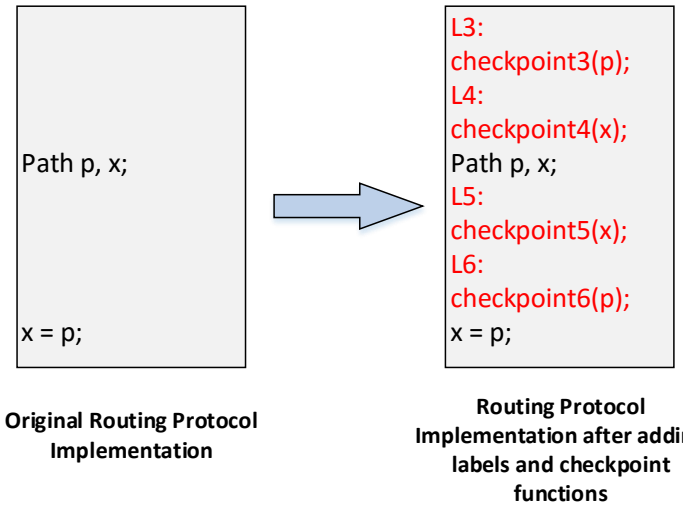


Figure 3.3: Adding labels and checkpoint functions to the original routing protocol implementation.

are used by the Postprocessor to generate external program states and new *checkpoint* functions. For instance, in Code block 3.1, assuming `int a` and `float b` are the only two living variables when program execution reaches function *checkpoint3*, then these two variables will be read as input parameters for function *checkpoint3* and packaged as a *Program State* which is stored in heap memory for future usage.

Code block 3.1: A new checkpoint function.

```
routing_protocol(Topology t, const TraceInfo &traceInfo) {
    switch (traceInfo.getLabel()) {
        case 0: goto L0;
        ...
        case 3: goto L3;
        ...
    }
    int a;
    float b;
L3:
    checkpoint3(traceInfo, a, b);
    ...
}

// A typical definition of checkpoint function
void checkpoint3(TraceInfo &traceInfo, int &a, float &b) {
    switch (traceInfo.getJob()) {
        case LOAD: {
        // load program state and recover each input parameter
            ProgramState3 *ps = static_cast<ProgramState3 *>(traceInfo.getPSPtr());
            a = ps->a;
            b = ps->b;
            traceInfo.setJob(STORE);
        }
        break;
        case STORE: {
            ProgramState3 *new_ps = new ProgramState3(a, b);
            traceInfo.setPSPtr(new_ps);
            TraceInfo copy(traceInfo);
            // store copy in a global container
        }
        break;
        default: break;
    }
}
```

3.3.1 Implementation of Path Variables Marker

Marker first calls clang++ to translate the original program into bitcode file written in LLVM IR and conducts analysis on this bitcode file. A bitcode file normally consists of a module, which is constructed with a number of functions containing several basic blocks. Typically, Marker analyzes all variable declaration instructions in a basic block and makes a record of the variable's names if their datatype are `Path`.

3.3.2 Implementation of Preprocessor

Preprocessor's work is to analyze the original input program line by line and locate each `Path` variable. Then it adds a label and a simple *checkpoint* function right before the line where the `Path` variable appears. Then, the Preprocessor writes each statement into a new file and provides all *checkpoint* function prototypes. Here, the checkpoint functions do not have to be really implemented since Instrumenter only needs their declarations.

3.3.3 Implementation of Instrumenter

Similar to what Marker does, Instrumenter's work also involves variable analysis. However, the Instrumenter has to analyze each variable's scope and record all living variables right before each program point where a checkpoint function located. Specifically, the Instrumenter maintains a global set `predecessors` to record each basic block's predecessors and a number of local set `directPreds` to record each basic block's direct predecessors. For each basic block, the Instrumenter iterates and records its direct predecessors and then finds out the nearest common predecessor (NCP) by intersecting all direct predecessors' predecessors. Within each basic block, the Instrumenter iterates each instruction and uses a local set to record each newly

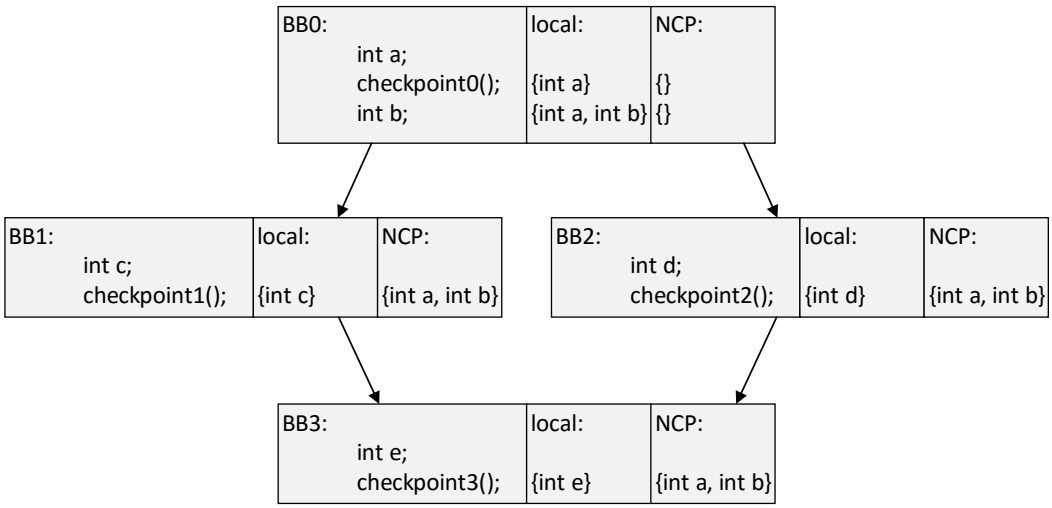


Figure 3.4: Variables' scope across basic blocks.

declared variable's name and datatype. When the Instrumenter reaches an `invoke` instruction that calls a *checkpoint* function, it makes a record of all living variables in both local and NCP's set. Figure 3.4 illustrates a typical example, in which *BB1* and *BB2* are *BB3*'s direct predecessors while *BB0* is the NCP. Obviously, both *BB1* and *BB2* inherit their common direct predecessor *BB0*'s living variables `int a` and `int b` whereas *BB3* only inherits its NCP *BB0*'s living variables rather than *BB1* or *BB2*'s living variables since `int c` and `int d` are just valid within their own basic blocks *BB1* and *BB2*, respectively. During the iteration of basic blocks, the Instrumenter also tries to remove irrelevant labels and *checkpoint* functions before some `Path` variables' appearances and does renumbering afterwards. This is because only `load` and `store` instructions need access to variables with `Path` datatype and affect program execution and final results. An instruction like `Path p;`, which just declares an empty `Path` type variable, has nothing to do with the execution since it does not attend any load or store operation which may change the program state.

3.3.4 Implementation of Postprocessor

Postprocessor does the last preprocessing work before a modified routing protocol program is sent for execution. Generally, after obtaining all living variables in each checkpoint, postprocessor does the following work.

- Removing useless labels and *checkpoint* functions and renumbering the remaining labels and *checkpoint* functions.
- Replacing the old *checkpoint* functions with a number of new *checkpoint* functions which load and store all living variables.
- Constructing a series of `ProgramState` type structures for the new *checkpoint* functions.

Postprocessor uses some mapping information between the old and the new label and *checkpoint* function sequences. By iterating the input program line by line, the removing and renumbering can be easily conducted. However, the construction of new *checkpoint* functions and corresponding `ProgramState` type structures is comparatively complicated.

Before different *checkpoint* functions, the type and number of living variables could be different accordingly. Therefore, multiple different `ProgramState` type structures with different variable members have to be constructed. Since all programs states with different `ProgramState` datatype have to be uniformly stored, C++ dynamic binding technique is leveraged for their management. Precisely, all `ProgramState` datatypes inherit from a base `ProgramState` datatype which contains some important members such as label sequence. From Code block 3.2, we can see that each derived `ProgramState` datatype extends the base datatype by adding corresponding living variables as its members.

Code block 3.2: A base and a derived ProgramState datatype definition.

```
1  // Job type definition
2  typedef enum {LOAD, STORE} Job;
3
4  // Base ProgramState datatype definition
5  struct ProgramState {
6      int label;
7      Job job;
8      ...
9      ProgramState() : label(0), job(LOAD) ...
10     ProgramState(int x) : label(x), job(LOAD) ...
11 };
12
13 // Derived ProgramState1 datatype definition
14 struct ProgramState1 : public ProgramState {
15     int a;
16     Path b;
17     ...
18     ProgramState1(int a, Path b, ...) : ProgramState(1), a(a), b(b) ...
19 };
20
21 //Derived ProgramState2 datatype definition
22 struct ProgramState2 : public ProgramState {
23     int c;
24     Path d;
25     ...
26     ProgramState2(int c, Path d, ...) : ProgramState(2), c(c), d(d) ...
27 };
```

Thus, all program states can be stored in a uniform manner and their real datatypes are resolved at runtime. In practice, the Postprocessor automatically generates these structures and handles a number of pointers to corresponding program states.

Along with the `ProgramState` datatypes, a series of new *checkpoint* functions are automatically created to collect and package all related living variables and labels into a program state and manage its load and store operations. Code block 3.1 shows a typical implementation of a new *checkpoint* function. *checkpoint3* reads an input program state pointer `ps`, a `job` denoting load or store operation and two living local variables `a` and `b`. Based on different job type, the *checkpoint* function may (1) load program state from the input pointer `ps` after dynamic binding at runtime and

then recover all current living variables or (2) create a new program state collecting current living variables and put its pointer into a global container for future usage. In practice, smart pointers are used to replace regular pointers due to their thread safety and better garbage collection mechanism.

To facilitate the input of program states, the Postprocessor modifies the original routing protocol program, as shown in Code block 3.1, by (1) appending a `TraceInfo` to the parameter list (2) adding a `switch-case` statement at the beginning to help skip redundant program execution and directly jump to the checkpoint and recover all living variables before the checkpoint. In fact, `TraceInfo` is the most important structure in the entire protocol checking system and contains a number of critical data including a pointer to a program state.

3.4 Topology generation design

We can see from Code block 3.1 that a typical routing protocol program execution depends on an input topology. By leveraging geng, a nauty graphic tool, the topology generator is able to automatically generate unique topology structures, which is formally termed as non-isomorphic connected graphs. geng constructs the graph starting with vertex 0, then adding vertices $1, 2, 3, \cdots$ in that order. Each graph in the sequence is an induced subgraph of all later graphs in the sequence. To generate all connected non-isomorphic graphs of a given order n, geng extends non-isomorphic graphs of size $n-1$ by connecting a new vertex to the graph in all possible ways. For each extended graph, it checks if the new vertex was canonical. If so, the new graph serves as a new non-isomorphic graph of order n. Otherwise, the new graph is discarded.

geng provides several convenient interfaces for external program calls. The main requirement is to change the name of the main program to be other than `main`. This is done by defining the preprocessor variable `GENG_MAIN`. Besides, a preprocessor

variable `OUTPROC` has to defined as well in order to receive the graphs generated. To call the interfaces shown in Code block 3.3, an argument list `argv[]` also has to be defined. `ADDONEEDGE` provides an interface to feed an existing graph and two nodes `v` and `w` creating an edge to be added. Therefore, the topology generator just needs to obtain the node `v` and `w` in program context.

Code block 3.3: geng interfaces.

```
1  // output interface of a newly generated graph
2  void OUTPROC(FILE *outfile, graph *g, int n) { ... }
3
4  // GENG_MAIN interface
5  #ifdef GENG_MAIN
6  int GENG_MAIN(int argc, char *argv[])
7  #else
8  int main(int argc, char *argv[])
9  #endif
10
11 // interface to add one undirected edge v-w to a graph
12 ADDONEEDGE(graph *g, int v,int w, int m);
13
14 void callgeng() {
15     int geng_argc;
16     char *geng_argv[5];
17
18     // Set up geng argument list.
19     geng_argv[0] = "geng";
20     geng_argv[1] = "-lv";
21     geng_argv[2] = "-c";
22     geng_argv[3] = "4";
23     geng_argv[4] = NULL;
24     geng_argc = 4;
25
26     GENG_MAIN(geng_argc, geng_argv);
27 }
```

In this implementation, `Topology` datatype is rewritten to store all related important data such as program states and all possible paths between any two nodes in the topology. An adjacent matrix is used to store nodes' connections, namely, whether an edge connects two nodes in the topology.

3.5 Path generation approach

In addition to calling geng to generate non-isomorphic graphs, the other critical work is to inherit program states and derive all possible paths from the previous graph after adding a new edge. A naive approach to generate all possible path between any two nodes in a topology is depth-first search (DFS) based on backtracking algorithm. However, a pure path search problem is an NP problem which has exponential time complexity. Since the topology generator always generates a new topology by adding a new edge to an old topology, it is easy to have an $O(n^2)$ time complexity approach where n denotes the number of all possible paths that have been found.

Practically, given a newly added bidirectional edge $u - v$, two existing bidirectional paths $x - u$ and $v - y$, the path generator does the following three work.

- Concatenating $u - v$ to $x - u$ and $v - u$ to $u - x$ to obtain $x - v$ and $v - x$.
- Concatenating $u - v$ to $v - y$ and $v - u$ to $y - v$ to obtain $u - y$ and $y - u$.
- Connecting $x - u$ and $v - y$ using $u - v$ and connecting $y - v$ and $u - x$ using $v - u$ to obtain $x - y$ and $y - x$.

To avoid unnecessary loops when connecting paths, path generator maintains a bitset of nodes constructing a path. Thus, if more than one same nodes exist in both two paths to be connected, the connection is aborted.

3.6 Incremental Program Executer design

The Incremental Program Executer is the most critical part in this automatic routing protocol checking procedure. Generally, with the assistance of a generic routing metric interface and a backend path preference relation database, it traverses all possible program execution traces and builds a trace tree for a specified topology.

3.6.1 Generic routing metric interface

The typical way that a routing protocol interfaces with a routing metric is shown in Figure 3.5, where the routing protocol calls the metric function $w()$ to get the weights of several paths and then make path selection decisions by comparing their weight values. This common way of interfacing is not convenient for protocol checking. In protocol checking, what is under consideration is not a single metric but some metric properties that can be potentially shared by infinite number of metrics. Thus, we cannot have an actual weight returned by function $w()$ while capturing the potentially different weight values produced by all the infinite metrics.

To address this challenge, the Incremental Program Executer converts the conventional interface between routing protocol and metric to a generic form that does not require any actual weight value. In essence, given two paths a and b, a routing protocol prefers path a only because a's weight returned by the metric function $w()$ is smaller than b's weight; the actual weight values are irrelevant. Thus, as long as we can ensure that a routing protocol can correctly get path preference information from the routing metric, whether the routing protocol sees the realpath weight values does not matter. Based on this rationale, using the generic interface code illustrated in Fig. 3.5, we can convert the typical interface implementation to a generic format. Leveraging operator overloading, the conversion only requires the routing metric to give path preference through function *path_compare*(). No actual weight values are returned to the routing protocol side following this generic interface.

With the newly converted interface format, the interactions among routing protocols and metrics are now reflected by the function calls to *path_compare*(). Our protocol checking mechanism then can verify if a routing protocol can achieve its optimality, consistency or loop-freeness goals by analyzing these function calls to *path_compare*().

In Figure 3.5, function *path_compare*() only accepts two parameters `Path a` and `Path`

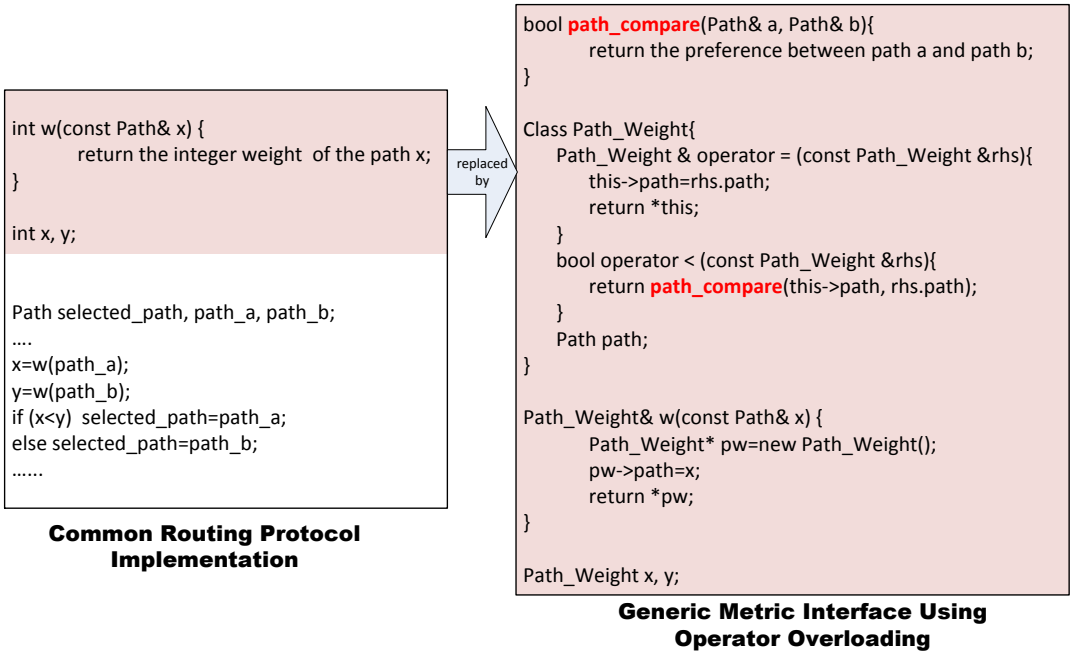


Figure 3.5: Converting a conventional metric interface to a generic format.

`b`. In practice, it is replaced with a function that has the same name and one more parameter with datatype `TraceInfo`. To specify, before sent to execution, the routing protocol program, along with the substituting *path_compare*() function prototype, is first compiled to LLVM IR bitcode file. Then all *path_compare*() function calls are replaced with the new ones and, what is more important, the latest `traceInfo` variable, which is an input of the modified routing protocol program, will be fed to the new *path_compare*() function calls. Code block 3.4 presents a code snippet of the new *path_compare*() function. We can see that *path_compare*() function first tries to query the preference relation between path `a` and `b`. If the result is uncertain, it copies the input `traceInfo` and then insert two contrary preference relation of path `a` and `b` into the original `traceInfo` and its copy, respectively. At last, *path_compare*() function initiates a new execution of routing protocol program with an input a copy

of `traceInfo` that has been inserted a preference relation and return a boolean result to the function calls it.

Code block 3.4: new path_compare function prototype.

```
bool path_compare(Path &a, Path &b, TraceInfo &traceInfo) {
    // Query path preference relation
    int res = traceInfo.query(a, b);
    switch (res) {
        case -1: return true;
        case 1: return false;
        default: break;
    }

    // Make a local copy of traceInfo
    TraceInfo copy(traceInfo);

    // Insert different preference relation between path a and b
    traceInfo.insert(a < b);
    copy.insert(b < a);

    // Initiate a new execution of routing protocol program
    routing_protocol(t, copy);

    return true;
}
```

3.6.2 Backend path preference relation database

In general, the entire protocol checking procedure is based preference relation of different paths. When a routing protocol calls *path_compare*() with two paths a and b, *path_compare*() first finds out which preference relation among $a \prec b$ and $a \succ b$ can satisfy the constraint of the given metric properties and returns this preference to the routing protocol. If both preference relations are feasible, the routing protocol execution is split into two threads, where one thread gets the $a \prec b$ preference while the other thread gets the $a \succ b$ preference. A trace tree as illustrated in Figure 3.6 is built to record the calls to *path_compare*() and its returned preference values to all routing

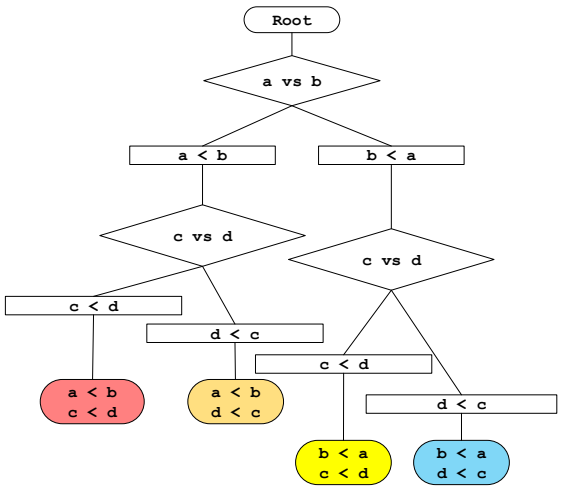
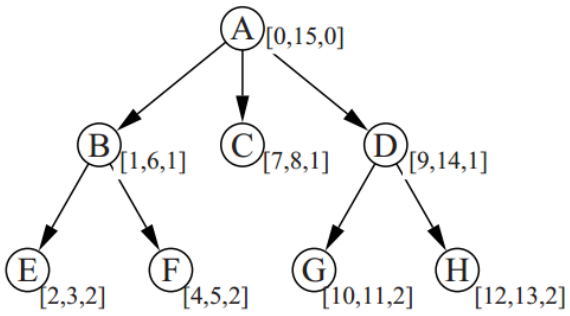


Figure 3.6: A typical trace tree generate by program execution.

protocol threads. In the trace tree, each ◊ node represents a call to *path_compare*() and each □ node represents the returned preference result. When the routing protocol thread is split into two threads due to two valid preferences at a *path_compare*() call, the particular ◊ node will have two □ children. Similar branching continues until all routing protocol threads output all preference relations collected along the top-down execution traces and terminate their path computation. To collect each preference relation along a execution trace, we designed a path preference relation database, which supports storage, insertion, and query of paths' preference relations. In particular, each time a new preference relation is inserted, the database conducts automatic computation of additional preference relations that can be derived from the new preference relation and some given routing metric properties.

To ensure fast query, we used an adjacency list to store all paths' preference relations in the database. In the adjacency list, any two paths with preference relation are treated as two adjacent vertices in a directed graph. Thus, each query of two paths' preference relation becomes a reachability query of two vertices in this directed graph. To deal with these queries, we realized GRIPP index structure [7], which can quickly



(a) Pre- and postorder labeling of a tree.

Figure 3.7: Indexing trees by pre- and postorder labeling.

answer reachability queries in constant time complexity on average and requires only linear space complexity.

As shown in Figure 3.7, GRIPP indexing scheme is based on the pre- and postorder indexing scheme for trees. Given a tree, in the pre- and postorder indexing scheme each vertex in the tree receives three values, a preorder value, a postorder value, and the depth of the vertex in the tree. Pre- and postorder values are assigned to a vertex according to the order in which the vertices are visited during a depth-first traversal of the tree. The preorder value v_{pre} is assigned the first time vertex v is encountered during the traversal. The postorder value v_{post} is assigned after all successor vertices of v have been traversed. Originally, two counters are used, one for the preorder value and one for the postorder value. Both are incremented after each assignment.

The list of vertices together with assigned pre- and postorder values and depth information form an index through which reachability and distance queries on trees can be answered with a single query. If w is reachable from v, w must have a higher preorder and lower postorder value than v, i.e., $w_{pre} > v_{pre} \wedge w_{post} < v_{post}$. During the creation of the index a vertex v always receives its preorder value before its successors get their pre- and postorder values. The postorder value of vertex v is assigned after

all successor vertices have pre- and postorder values. As the counter is incremented after every assignment, the pre- as well as postorder values of any successor vertex w of v must lie within the borders given by the pre- and postorder values of v, i.e., $[v_{pre}, v_{post}]$. Thus, $\text{reach}(v, w) \Leftrightarrow v_{pre} < w_{pre} < v_{post}$.

3.6.3 Incremental multithreading program execution

In Code block 3.4, after *path_compare*() function inserts two contrary preference relations into `traceInfo` and its copy, respectively, a new execution of routing protocol program is initiated. Obviously, *path_compare*() function cannot returns until the newly initiated execution terminates. To expedite the entire protocol checking procedure, the Incremental Program Executer utilizes a thread pool to make the executions run in parallel. By using a thread pool, a new execution can be detached from the execution which initiates it. Thus, *path_compare*() function is able to return at will. The new execution, at the same time, joins the thread pool's queue and waits for any available thread resource.

Apart from the thread pool, the Incremental Program Executer makes use of an incremental topology checking strategy which can greatly reduce redundant program executions. As mentioned in Section 3.4, geng derives a new non-isomorphic topology by adding an edge (or a node) to an old non-isomorphic topology it previously identified. Since the new topology only differs from the old topology by one edge/node, running a routing protocol on the same node in these two topologies will generate two trace trees that also have large common parts, which can be avoided as follows.

Note that the new trace tree only starts to differ from the old trace tree when the routing protocol program starts to involve the newly added edge (or node) in its execution (e.g. checking the existence of the new edge, comparing two paths that involve the new edge, etc.). Therefore, the Incremental Program Executer has to identify the new edge's first appearance position in each execution trace. Since it is

too difficult to predict next edge to be added, the Incremental Program Executer, during the routing protocol's execution process over the old topology, records the program states at all possible positions where any new edge would first appear. It can be noticed that the starting and destination node in a path can be the two nodes connecting the new edge. Thus, the Incremental Program Executer just needs to record the program states at all program points where a new starting and destination node pair from a path appears. The program state includes all local data and program counter (i.e., label) in current program stack. Then, when executing the routing protocol over a new topology, the program executer can directly start the protocol execution from these positions by loading the program states previously saved at these positions. The common parts of the old and new trace trees, hence, do not need to be recomputed.

3.7 Routing protocols and necessary requirements

Table 2.1 shows all the routing protocols implemented in the experiment and corresponding requirements that these routing protocols have to satisfy. Among the four test routing protocols, hop-by-hop routing protocols based on Dijkstra's algorithm and Bellman-Ford algorithm are mostly used in both wired and wireless networks while those based on Greedy algorithm and CGF algorithm are usually seen in geographic routing scenarios. Since these two types of routing protocols may differ in several aspects, e.g., protocol implementation, requirements, and essential routing metric properties, the automatic protocol checking procedure design has to find a uniform scheme to handle both of them.

3.7.1 Requirements on routing protocols

The protocol checking usually focuses on whether a specific routing protocol can achieve three targets: optimality, consistency and loop-freeness.

Optimality

In general, a routing protocol computes its optimal path by issuing a series of calls to *path_compare*(). For each *path_compare*() call, the routing protocol learns a preference relation between two paths. When path preferences among enough path pairs are learned, the routing protocol determines the "optimal" path to its destination node and terminates its path computation. Whether this "optimal" path is truly optimal depends on if there exists another path to the destination node that is more preferable than the "optimal", subject to the constraint of the given routing metric properties. Based on this observation, the protocol checking for optimality is inserted at the end of each routing protocol program execution.

Specifically, when a single protocol program execution finishes searching a given topology for an "optimal" path between two nodes s and d, the decision maker queries the backend database for a comparison result between the "optimal" path and an alternative path previously found by the Incremental Path Generator. If any query result is uncertain, namely, the "optimal" path is not preferable than any other path for sure, the Decision Maker will construct a counterexample by inserting into the backend database a contradiction preference relation showing that the alternative path is possibly more optimal the "optimal" path.

Consistency and Loop-freeness

Consistency and loop-freeness are goals for distributed hop-by-hop routing protocols. In hop-by-hop routing protocols, a node n computes its optimal path to a destination

d and forwards its packets to the next hop node m along the optimal path. Once the packets arrive at node m, node m forwards these packets along his own idea of optimal path to the destination d. If node m's optimal path is not a subpath of node n's optimal path, the routing system is not consistent. If node m's optimal path forms a loop when combined with node n's optimal path, the routing system is not loop-free. Consistent routing guarantees loop-freeness but loop-free routing is not always consistent.

From the above observation, it can be seen that the key for consistency and loop-freeness checking relies on the analysis of the relations between optimal paths computed at different nodes. Note that in the process of checking optimality, the protocol checking system has constructed a trace tree to capture all possible optimal paths from a source node to a destination node and the *path_compare*() function call sequences that lead to these optimal paths. Thus, the decision maker will leverage the trace trees constructed at different nodes to analyze the optimal path relation as follows.

From the trace tree constructed for a node n to a destination d, the decision maker picks a leaf node l_n that contains an optimal path $P_n = (n \rightarrow ... \rightarrow m \rightarrow ... \rightarrow d)$. For the node m that is on the path of P_n, the decision maker examines its trace tree and picks a leaf node l_m whose optimal path is $P_m = (m \rightarrow ... \rightarrow d)$ and P_m is not a subpath of P_n. Since P_m is not a subpath of P_n yet n is on path P_n, P_m and P_n may potentially be an inconsistent routing case. To examine if this potential inconsistent case is feasible, the decision maker next extracts the preferences returned by *path_compare*() calls along the branch that leads to leaf node l_n and the branch that leads to leaf node l_m. If it is feasible for these preferences to co-exist under the constraint of the given metric properties, this inconsistent routing case is confirmed to be feasible. In addition, if combining paths P_n and P_m forms a loop, a routing loop case is also found. If after repeating the above procedure for all the leaf nodes in the trace trees of all nodes, the decision maker still cannot find any inconsistent/loop routing case, it can be concluded that in the given topology, the routing protocol is

consistent/loop-free for the given routing metric properties.

3.7.2 Topology-based routing protocols checking

Dijkstra's algorithm and the Bellman-Ford algorithm are widely used in wireless routing. To guarantee correct behaviors, routing protocols based on these two algorithms has to be enforced with some essential properties in order to satisfy optimality, consistency and loop-freeness. These properties are implemented in the backend preference relation database and co-work with the GRIPP index structure. Since the adjacency list stores all paths' preference relations, when a preference relation $a \prec b$ is inserted into the database, the adjacency list will make self-adjustment based on these properties.

Left-isotonicity for paths

To enforce left-isotonicity, a path a is first divided into two subpaths, a prefix subpath p and a remaining subpath r. Therefore, we can know that $a = p \oplus r$. Then, we traverse the adjacency list which stores all paths that has been visited or automatically inserted based on the property enforcement. During the traversal, we try to collect all paths that are not preferable than subpath r. Assuming path i is among these paths, i.e., $r \prec i$, we can easily imply a preference relation $p \oplus r \prec p \oplus i$, i.e., $a \prec p \oplus r$, and insert it into the database.

Right-isotonicity for paths

Similar to the strategy used in left-isotonicity enforcement, a path a is first divided into two subpaths, a suffix subpath s and a remaining subpath r such that $a = r \oplus s$. During the traversal of adjacency list in the backend database, for any path i that is not preferable than subpath r, namely, $r \prec i$, a preference relation $r \oplus s \prec i \oplus s$,

i.e., $a \prec r \oplus s$, is inserted.

Left-(Right-)monotonicity for paths

The logic to ensure left-(right-)monotonicity is straightforward. Given a path p, we divide it into two subpaths a and b such that $p = a \oplus b$. To satisfy left-monotonicity, we insert into the database a preference relation $b \prec a \oplus b$, i.e., $b \prec p$. Similarly, a preference relation $a \prec a \oplus b$, i.e., $a \prec p$, is inserted into the database to enforce right-monotonicity.

For all the subpaths of path p, we replay this logic until all preference relations are inserted.

3.7.3 Geographic routing protocols checking

In geographic routing, greedy routing and face routing are two typical routing algorithms. The experiments cover two types of geographic routing protocols which are based on pure greedy routing and Combined Greedy-Face (CGF) routing, respectively. In greedy routing, each node tries to forward the message to the most suitable neighbor that is closer to the destination in each step only based on local information. Face routing can be used to recover from dead end situation, where greedy routing reaches a void and cannot find a suitable neighbor to the destination.

In general, geographic routing is based on nodes' positions and quite different from topology-based routing scheme. These positions are, in most cases, euclidean coordinates, which are essential for various geographic routing metrics to select the most suitable neighbor for next hop. Thus, it can be noticed that a geographic routing protocol program does not directly work on actual nodes' positions. In fact, a geographic routing protocol just needs the information of next step that a routing metric provides. In theory, a geographic routing metric computes the next step

for a specified geographic routing protocol based on the traffic flow from the source node, neighbor nodes' coordinates and some significant characteristics of the links from current node to the neighbor nodes. Therefore, the topology-based protocol checking approach works for geographic routing protocol as well. To specify, given a general topology of nodes and links without any further details, a geographic routing protocol program just needs to search for a path from the source to the destination node by following the instruction of a specified routing metric. Particularly, any single link in a topology is treated as a one-link path in geographic routing protocol checking.

Note that greedy routing does not provide delivery guarantee. This is because the greedy nature may direct the traffic into a dead end if a geographic routing metric is unable to select any suitable neighbor as current node's next hop. Therefore, the validity of a link has to be taken into consideration in checking a geographic routing protocol. Commonly, when *path_compare*() function tries to compare two paths a and b, it returns two possible results representing $a \prec b$ and $b \prec a$. In geographic routing protocol checking, *path_compare*() function has to return a third result showing that neither path a nor path b is valid. Thus, *path_compare*() additionally creates a thread for new protocol program execution without inserting any preference relation $a \prec b$ or $b \prec a$. Furthermore, an adjacency matrix *valid* is maintained in the backend database to record the validity of links in a topology. Precisely, if a link $l(u, v)$ is a valid candidate for geographic routing, $valid[u][v]$ will be assigned with value -1. Otherwise, $valid[u][v]$ will be assigned with value 1 to represent an invalid link $l(u, v)$.

In order to ensure loop-freeness and consistency, a geographic routing protocol and metric combination must have some of the following five properties.

Odd symmetry

For two links with opposite directions, the odd symmetry property requires that only one of them can be used in greedy routing. Consequently, $valid[u][v]$ and $valid[v][u]$ cannot be -1 or 1 simultaneously.

Transitivity

The transitivity property requires that the progress to the destination is monotonic. Thus, given a link $l(u, v)$, for any link $l(i, u)$, if both $valid[i][u]$ and $valid[u][v]$ are -1, $valid[i][v]$ must be ensured to be -1 as well.

Strict order

The strict order property enforces that only a unique outgoing link will be selected as a next hop. In the implementation, whether this property is held will be validated in final decision making procedure. To specify, for two links $l(u, v)$ and $l(u, w)$, we must be able to find in the database only a unique preference relation $l(u, v) \prec l(u, w)$ or $l(u, w) \prec l(u, v)$.

Source independence

The source independence property demands that the selection of next hop should not related to the source node. Similar to the logic in implementing strict order property, the decision maker will validate source independence property by checking a number of requirements such as loop-freeness or consistency. In general, for any two paths

$$p(s_{10}, d) = \langle s_{10}, s_{11}, \cdots, s_{1i}, s_{1(i+1)}, \cdots, s_{1(i+k)}, \cdots d \rangle$$

$$p(s_{20}, d) = \langle s_{20}, s_{21}, \cdots, s_{2j}, s_{2(j+1)}, \cdots, s_{2(j+k)}, \cdots, d \rangle$$

where $s_{10} \neq s_{20} \wedge k > 0 \wedge i, j \geqslant 0$, if $s_{1i} = s_{2j}$, there must not exist any following nodes $s_{1(i+k)}$ and $s_{2(j+k)}$ such that $s_{1(i+k)} \neq s_{2(j+k)}$.

Local minimum free

Significant in face routing, the local minimum free property guarantees monotonic progress between any two nodes along a baseline. Precisely, if link $l(u, v)$ is selected as a hop in face routing, link $l(v, u)$ with the opposite direction cannot be selected as the next hop. In other words, if $valid[u][v] = -1$, $valid[u][v] \neq -1$ must be enforced.

Chapter 4

Routing Metric Checking

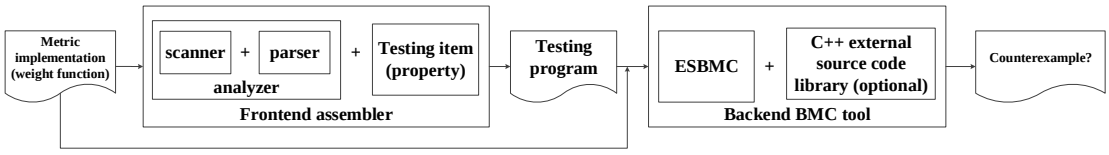


Figure 4.1: Metric checking procedure.

Metric checking assumes that a routing protocol and its requirement on routing metric properties are known. The goal of metric checking is to find out if a routing metric implementation satisfies the required properties of the given routing protocol.

Our automatic metric checking design leverages model-based verification. Model-based verification techniques, such as model checking, are based on models that describe possible system behaviors in a mathematically precise and unambiguous manner. Some algorithms, associated with the system models, are used to systematically explore all possible states of these system models. Since model checking can exhaustively and automatically verify whether a model of system satisfies a specification, we hence utilize it to test whether an unknown routing metric design satisfies the essential properties to be compatible with a given routing protocol. In addition,

model checking may also produce counterexamples to demonstrate the lack of certain property. These counterexamples can be used to validate our method's metric checking conclusion.

To realize automatic verification of metric mathematical property, a popular verification technique called Bounded model checking (BMC) [8] is adopted in our metric checking system due to its searching space efficiency over many other verification techniques.

4.1 Implementation using BMC

As shown in Figure 4.1, the whole metric checking procedure mainly consists of two parts, a frontend assembler and a backend BMC tool. The frontend assembler is further divided into an analyzer, which is constituted of a scanner and a parser, and a testing item referring to a property.

At the beginning of the metric checking procedure, the scanner reads the metric implementation line by line until the metric weight function is found. Commonly, the name of this weight function should be explicitly specified by users of the metric checking system. Without loss of generality, we assume the function is named *weight*(). Code block 4.1 presents a typical weight function prototype that is acceptable to the frontend assembler. Assuming each metric weight function is written in an acceptable format similar to the one in Code block 4.1, the parser then does simple lexical analysis on the function prototype so as to obtain this function's return type, total number of parameters, and each parameter's data type. Based on the analysis results, the frontend assembler creates a testing C/C++ program for the weight function. In general, the user of the automatic metric checking system explicitly informs the assembler the metric properties that he or she wants to check and the assembler will create the testing program with the corresponding testing instructions. Code block 4.2 contains a typical testing program to check whether

a metric holds right-isotonicity property. It can be noted that the testing program gives variable `a`, `b`, `c`, `res0`, `res1`, `res2`, `res3` the correct data type because the parser is able to correctly identify the

Code block 4.1: A typical acceptable weight function prototype.

```
1 template<class T>
2 float weight(const Path<T> p);
```

Code block 4.2: Verifying a metric's right-isotonicity property.

```
1  /* The metric property testing program */
2  #include "wcett.h"
3
4  int main() {
5    Path<float> a, b, c;
6
7    float res0, res1, res2, res3;
8    res0 = weight(a);
9    res1 = weight(b);
10   res2 = weight(a+c);
11   res3 = weight(b+c);
12
13   __ESBMC_assume(res0 >= res1);
14   __ESBMC_assert(res2 >= res3, "Right-isotonicity␣violated!");
15
16   return 0;
17 }
```

data type of the weight function's input parameter and return value through its code analysis. Line 10-16 are created since the user instructed the assembler that he or she wants to check right-isotonicity.

Once the assembling process is done, the testing program, together with the metric implementation, is submitted to the backend BMC tool to conduct a Bounded Model Checking. As a counterexample-driven approach, Bounded Model Checking (BMC) is widely used for bug-finding. There are many BMC-based software that can serve in our metric checking system. In our current prototype, we use a powerful BMC-based softwares called ESBMC [9]. ESBMC is a tool for the formal verification of

embedded C/C++ programs using BMC. It is designed for usability and it supports full ANSI-C language features. Having attended all the four International Competition on Software Verification (SV-COMP), ESBMC presents balanced performance in regards to correctness and efficiency when processing complex C/C++ programs.

A counterexample may be produced if ESBMC finds a case that makes the assertion statements (e.g. line 16 in Code block 4.2) in the testing program fail. If no counterexample can be found by ESBMC, it can be concluded that the metric implementation that is being tested satisfies the specified property.

4.2 Case studies

The design of the automatic routing metric checking system is based on the compatibility theories for routing metrics and protocols. Each property required for routing metrics strictly follows the theoretical definitions. In order to show the effectiveness of our approach, we tested our approach on ten metric designs proposed and examined in existing literatures.

4.2.1 WCETT metric

Draves et al. proposed in [10] a routing metric named Weighted Cumulative ETT (WCETT) to evaluate path qualities based on link quality and intra-flow interference in multi-channel mesh networks. According to their definition, for a path p, WCETT is defined as:

$$w(p) = (1-\beta)\sum_{\text{link } l\in p} ETT_l + \beta \max_{1\le j\le k} X_j,$$

where β is a tunable parameter subject to $0 \le \beta \le 1$. ETT_l here is the expected transmission time of a link l, i.e. the expected MAC layer duration for a successful transmission of a packet at link l. X_j is the number of times that channel j is

used along path p and captures the intra-flow interference at channel j, which is interference among wireless links on the path of a same flow and also operates on the same channel. When WCETT is applied to link-state routing system, it is possible to create routing loops and suboptimal paths since it lacks isotonicity property.

4.2.2 Hop count metric

As the simplest metric, hop count is widely utilized by many routing protocols, such as Routing Information Protocol (RIP). Generally, for a path p, its weight function is defined as:

$$w(p) = X,$$

where X is the number of hops along path p. When combined with common routing protocols such as link-state routing or path vector routing, hop count metric does not create loops or suboptimal paths since it is well-known to have isotonicity and monotonicity, which are two essential properties for proper routings in link-state and path vector routing systems. [2].

4.2.3 Compass routing metric

Compass routing metric is proposed for geographic routing in [11]. By definition, a packet is forwarded to the node with the minimum angle between itself and destination among all the neighbors of the forwarding node.

For compass routing metric, a link $l(u, v) =< (x_u, y_u), (x_v, y_v) >$ joins two nodes u and v who have geographic positions (x_u, y_u) and (x_v, y_v), respectively. The $w(\cdot)$ function of compass routing metric is defined as

$$w(l(u, v)) = -cos\angle\overrightarrow{ud}, \overrightarrow{uv},$$

where $\angle\overrightarrow{ud}, \overrightarrow{uv}$ is the angle from the edge $\overrightarrow{ud}$ to $\overrightarrow{uv}$.

Designed for greedy routing, compass routing metric is whereas not loop-free [11] due to its lack of odd symmetry, transitivity and strict order properties [4].

4.2.4 Most Forward within Radius metric (MFR)

MFR [12] is the most basic greedy routing metric, whose weight function $w(\cdot)$ is defined as follows:

$$\begin{aligned} &w(l(u,v)) = \|v,d\| - \|u,d\| \\ &= \sqrt{(x_v - x_d)^2 + (y_v - y_d)^2} - \sqrt{(x_u - x_d)^2 + (y_u - y_d)^2}, \end{aligned}$$

Essentially, MFR tries to bring the message closer to the destination in each step by choosing the next hop node as the neighbor that is closest to the destination. It can be learned from [4] that MFR metric has odd symmetry and transitivity properties, but lacks strict order property and thus has been proved in [4] to be able to guarantee loop-free routing in geographic greedy routing system but the path selection may be inconsistent.

4.2.5 A variant of Greedy forwarding metric

Another advanced greedy forwarding metric, denoted as Adv. GFv1 in our paper, is introduced in [4], to capture both the location information and link quality design as follows.

$$w(l(x,y)) = g_{x,y} + q_{x,y},$$

where $q_{x,y}$ denotes link quality and $g_{x,y} = \|x,d\| - \|y,d\|$ represents the difference between two nodes x and y's physical distance heading for destination d. The authors provided a counterexample in [4] to show that this design is problematic due to lack of odd symmetry property.

4.2.6 A revised version of Greedy forwarding metric

Since Adv. GFv1 lacks odd symmetry property, Li and Yang also introduced in [4] a revision, denoted as Adv. GFv2 in our paper. Its weight function is shown as follows.

$$w(l(x,y)) = 1_A(g_{x,y}) \times (g_{x,y} + q_{x,y}),$$

where $1_A(\cdot)$ is the indicator function, namely, $1_A(g_{x,y}) = 1$ if $g_{x,y} > 0$, and $1_A(g_{x,y}) = 0$ otherwise. This version has been proved in [4] to successfully guarantees two properties, odd symmetry and transitivity which are the conditions for loop-free greedy routing system.

4.2.7 Nearest with Forward Progress (NFP)

Nearest with Forward Progress (NFP), proposed by Hou in [13], is to forward a packet to the nearest neighbor resulting in forward progress.

The $w(\cdot)$ function of NFP metric is defined as follows:

$$w(l(x,y)) = 1_A(cos\angle\overrightarrow{ud}, \overrightarrow{uv}) \times \|u,v\|,$$

where the indicator function $1_A(\cdot)$ is $1_A(cos\angle\overrightarrow{ud}, \overrightarrow{uv}) = 1$ if $cos\angle\overrightarrow{ud}, \overrightarrow{uv} \geq 0$, and $1_A(cos\angle\overrightarrow{ud}, \overrightarrow{uv}) = -1$ otherwise.

It can be proved that NFP will create routing loops if it is combined with greedy routing-based hop-by-hop routing protocol because it lacks three properties emphasized in [4]: odd symmetry, transitivity, and strict order.

4.2.8 Random Progress Forwarding (RPF)

Random Progress Forwarding (RPF) is also a greedy routing metric. Based on Nelson and Kleinrock's design of RPF in [14], if a straight line joins the forwarding node

and the destination, a packet is forwarded to the neighbor that has the maximum positive projected distance on this line. The weight function $w(\cdot)$ of RPF is

$$\begin{aligned} w(l(u,v)) &= -\|u,v\| cos\angle\overrightarrow{ud},\overrightarrow{uv} \\ &= -\sqrt{(x_v-x_u)^2+(y_v-y_u)^2} cos\angle\overrightarrow{ud},\overrightarrow{uv}, \end{aligned}$$

In [4], RPF has been proved to lack odd symmetry property and hence cannot provide loop-free routing in greedy routing system.

4.2.9 Line Progress Forwarding (LPF) metric

LPF metric, introduced in [4], is similar to RPF metric. However, it computes the projected distance on the straight line that joins the source and the destination. Hence, the $w(\cdot)$ function of LPF metric is

$$\begin{aligned} w(l(u,v)) &= -\|u,v\| cos\angle\overrightarrow{sd},\overrightarrow{uv} \\ &= -\sqrt{(x_v-x_u)^2+(y_v-y_u)^2} cos\angle\overrightarrow{sd},\overrightarrow{uv}, \end{aligned}$$

where $cos\angle\overrightarrow{sd},\overrightarrow{uv}$ is the angle from the edge $\overrightarrow{sd}$ to $\overrightarrow{uv}$.

The authors in [4] concluded that LPF metric has odd symmetry and transitivity properties. Therefore, greedy routing with LPF metric is loop-free.

4.2.10 Virtual Force Forwarding (VFF) metric

VFF metric, also introduced in [4], is designed to avoid routing through some specified areas, e.g., a compromised zone. Mathematically, the weight function $w(\cdot)$ is defined as follows:

$$\begin{aligned} & w(l(u,v)) \\ = & \left[\frac{\alpha}{(x_u-x_d)^2+(y_u-y_d)^2} - \frac{1-\alpha}{(x_u-x_h)^2+(y_u-y_h)^2}\right] \\ - & \left[\frac{\alpha}{(x_v-x_d)^2+(y_v-y_d)^2} - \frac{1-\alpha}{(x_v-x_h)^2+(y_v-y_h)^2}\right], \end{aligned}$$

In this function, α is a coefficient representing some trade-off. (x_d, y_d) and (x_h, y_h) are the coordinates of the destination and the center of the compromised zone, respectively.

In [4], the authors stated that VFF metric has odd symmetry, transitivity, and source independence properties, but lacks strict order property. Hence, it can be concluded that VFF metric can guarantee loop-free in a greedy routing system but may be short in the aspect of consistent routing states.

Chapter 5

Experimental Results and Analysis

All of the experiments were conducted on a shared-memory SGI UV system with 492 2.66GHz Intel Xeon cores and 2.62TB memory. To prove the effectiveness of our protocol checking method, we chose to check whether the four routing protocol implementations shown in Table 2.1 were able to ensure optimality, consistency and loop-freeness for different combinations of properties.

5.1 Incremental protocol checking

By using our incremental protocol checking mechanism, we tested all eligible topologies from two nodes to sixteen nodes. We set an execution time limit of 36 hours. If no counterexample is found within the time limit, we declare the protocol achieves its goal. Table 5.1 shows the experimental results of checking if the four routing protocols introduced in Table 2.1 guarantees essential operation goals when given different group of properties. "TLE" is short for Time Limit Exceeded while $\emptyset$ means no property is specified for checking. "Violated" in here represents a violation of specified operation goals and a consequent end of the checking procedure with an output of

a counterexample. In contrast, "Hold" means that no violation is detected after checking all enumerated topologies within the time limit.

Comparing Table 2.1 with Table 5.1, it can be observed that all the experimental results are identical to the existing theoretical analysis results. For each "Violated" case, a counterexample can be quickly found in less than 1 second; for each "Hold" case, time spent is comparatively longer since no counterexample was discovered after checking all eligible topologies. Particularly, the execution of the routing protocol based on Bellman-Ford algorithm exceeded the time limit in some cases. Considering its $O(|V||E|)$ time complexity where $|V|$ and $|E|$ refer to the number of vertices and edges in the topology, respectively, exceeding the time limit is understandable, especially comparing with the other three routing protocols. The extremely fast detection of counterexamples and violation cases demonstrate our observation that small counterexamples usually exists when a routing protocol and routing metric combination is problematic. When such counterexamples cannot be found after a long search time and enumerating of many large topologies, it is highly likely that the combination is theoretically sound.

Counterexample analysis

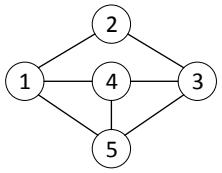


Figure 5.1: A counterexample graph in protocol checking experiment.

Figure 5.1 and Table 5.2 show a counterexample generated in the experiments where the metric is assumed to only has right-isotonicity property. Figure 5.1 is a topology generated in the checking procedure and Table VI shows a path weight preference

Table 5.1: Experimental results of checking routing protocols in Table 2.1.

Routing Protocol	Properties	Requirement	Result	Total Time (s)
Dijkstra's algorithm + hop-by-hop routing	∅	Optimality	Violated	0.008
		Consistency	Violated	0.05
		Loop-freeness	Violated	0.063
	Right-isotonicity	Optimality	Violated	0.007
		Consistency	Violated	0.079
		Loop-freeness	Violated	1.94
	Right-isotonicity + Right-monotonicity + Strictly left-isotonicity	Optimality	Hold	93612.1
		Consistency	Hold	103941.8
		Loop-freeness	Hold	124577.6
Distributed Bellman-Ford algorithm + hop-by-hop routing	∅	Optimality	Violated	0.005
		Consistency	Violated	0.092
		Loop-freeness	Violated	1.63
	Left-isotonicity	Optimality	Violated	0.014
		Consistency	Violated	0.132
		Loop-freeness	Violated	2.13
	Left-isotonicity + Left-monotonicity	Optimality	Hold	TLE
		Consistency	Hold	TLE
		Loop-freeness	Hold	TLE
Greedy algorithm + forward building hop-by-hop routing	∅	Consistency	Violated	0.003
		Loop-freeness	Violated	0.004
	Odd symmetry + Transitivity	Consistency	Violated	0.058
		Loop-freeness	Hold	52305.9
	Odd symmetry + Transitivity + Strict order + Source independence	Consistency	Hold	53114.8
		Loop-freeness	Hold	60819.3
CGF algorithm + forward building hop-by-hop routing	∅	Loop-freeness	Violated	0.007
	Local minimum free	Loop-freeness	Violated	0.006
	Odd symmetry + Transitivity + Local minimum free	Loop-freeness	Hold	59721.2

Table 5.2: A counterexample of path weight preference relation.

1	$1 \to 4 \prec 1 \to 2$	10	$4 \to 3 \prec 4 \to 1$
2	$1 \to 4 \prec 1 \to 5$	11	$4 \to 1 \prec 4 \to 5$
3	$1 \to 5 \prec 1 \to 4 \to 5$	12	$4 \to 5 \prec 4 \to 3 \to 5$
4	$1 \to 4 \to 3 \prec 1 \to 2$	13	$4 \to 3 \to 2 \prec 4 \to 1$
5	$1 \to 4 \to 3 \prec 1 \to 5$	14	$4 \to 3 \to 2 \prec 4 \to 5$
6	$1 \to 2 \prec 1 \to 4 \to 3 \to 2$	15	$4 \to 1 \prec 4 \to 3 \to 2 \to 1$
7	$1 \to 4 \to 3 \to 5 \prec 1 \to 5$	16	$4 \to 1 \prec 4 \to 3 \to 5$
8	$1 \to 4 \to 3 \to 5 \prec 1 \to 2$	17	$4 \to 1 \to 5 \prec 4 \to 3 \to 5$
9	$1 \to 2 \to 3 \to 5 \prec$ $1 \to 4 \to 3 \to 5$	18	$4 \to 1 \to 2 \to 3 \to 5 \prec$ $4 \to 1 \to 5$

configuration over this topology that makes Dijkstra-based link-state routing protocol unable to achieve optimality, consistency and loop-freeness. Specifically, given the preference in Table 5.2, Dijkstra's algorithm on node 1 and 4 identifies two optimal paths $1 \to 4 \to 3 \to 5$ and $4 \to 1 \to 5$. Obviously, optimality is violated because $1 \to 2 \to 3 \to 5$ is better than $1 \to 4 \to 3 \to 5$ and $4 \to 1 \to 2 \to 3 \to 5$ is better than $4 \to 1 \to 5$; consistency is violated because 4 forwards traffic through $4 \to 1 \to 5$, which is not a subpath of $1 \to 4 \to 3 \to 5$; loop-freeness is violated because a packet is recursively forwarded between 1 and 4.

Clearly, our automated protocol checking method is effective to diagnose the incompatibility between routing protocols and metrics.

5.2 Routing metric checking

The model checkers used for implementation are ESBMC 1.23. To prove the correctness of our metric checking method, we selected ten representative routing metrics for testing. Table 5.5 shows all the metric checking results produced by our automatic metric checking procedure in column "Automated Checking Results" and

the execution time of the automatic procedure is shown in column "Time". The theoretically proven results in existing literatures are shown in column "Theoretical Analytical Results". From the table, it can be seen that all our automatic metric checking results match with the theoretically proved result. This demonstrates that our automatic verification-based method is a reliable way to check the compatibility between routing metrics and routing systems.

Since all the results are proven correct, it can be claimed that our model checking approach is usually sound and complete when checking the compatibility between routing metrics and protocols. Moreover, it can be seen that the execution time of our method is very small, usually just a few seconds. Even in the worst case of VFF, it only takes around two minutes to finish. This demonstrates that our method is much more efficient than manual analysis on metrics, which may take hours or even days depending on the experience and mathematical skills of the analyzer.

Counterexample analysis

Our metric checking method also provides counterexamples for metrics that lack the checked mathematical compatibility properties. The following is the output from the WCETT automatic verification procedure, where counterexamples that demonstrate WCETT's lack of Left- and Right-isotonicity are clearly listed.

Each pair in the counterexample in Table 5.3 and Table 5.4 is of the form of (ETT, Channel number) that describes the ETT and channel assignment over a link. For these automatically produced counterexamples, we can easily draw a network topology that demonstrate the violation cases. For the counterexample in Table 5.3, we can calculate that $w(a) = 59$, $w(b) = 54$, $w(c + a) = 94$ and $w(c + b) = 96$. Apparently, $w(a) \succ w(b)$ but $w(c + a) \prec w(c + b)$. Therefore, left-isotonicity is violated. Similarly, for the counterexample in Table 5.4, we can get $w(a) = 30$, $w(b) = 28$, $w(a+c) = 53$ and $w(b+c) = 54$. Also, $w(a) \succ w(b)$ conflicts with $w(a+c) \prec w(b+c)$,

since it is an violation of right-isotonicity.

Table 5.3: WCETT's counterexample violating Left-isotonicity.

Path a	(1,11), (1,2), (1,11), (1,2), (1,2), (1,11), (1,2), (1,2), (1,2), (1,2), (1,2), (4,16), (8,15), (1,2), (2,2), (1,19), (2,7), (3,7), (1,0), (16,19)
Path b	(1,0), (1,0), (1,0), (1,0), (1,0), (1,19), (1,13), (1,13), (1,13), (1,2), (1,2), (1,16), (1,16), (1,16), (1,16), (1,13), (2,10), (6,15), (8,0), (16,13)
Path c	(2,0), (1,0), (1,0), (2,0), (1,0), (1,0), (1,0), (1,19), (1,0), (2,19), (2,16), (1,16), (1,15), (1,2), (1,16), (1,2), (1,10), (1,10), (10,0), (1,16)

Table 5.4: WCETT's counterexample violating Right-isotonicity.

Path a	(1,1), (1,5), (1,13), (1,12), (1,12), (1,5), (1,13), (1,12), (1,12), (1,12), (1,12), (1,14), (1,12), (1,12), (1,14), (1,0), (1,0), (1,14), (1,12), (2,2)
Path b	(1,12), (1,1), (1,12), (1,15), (1,15), (1,13), (1,0), (1,13), (1,18), (1,14), (1,18), (1,12), (1,15), (1,14), (1,15), (1,12), (1,14), (1,14), (1,3), (5,3)
Path c	(1,5), (1,1), (1,5), (1,13), (1,13), (1,13), (1,13), (2,18), (1,18), (1,14), (1,12), (2,14), (1,0), (1,14), (1,5), (1,8), (1,13), (1,14), (1,7), (1,15)

Table 5.5: Experimental results in metric checking experiment.

Routing Metrics	Testing Weight Function Properties	Theoretical Analysis Results	Automated Checking Results	Time (s)
WCETT [10]	Left-isotonicity	Lack	Violated	2.301
	Right-isotonicity	Lack	Violated	4.836
	Left-monotonicity	Hold	Satisfied	29.261
	Right-monotonicity	Hold	Satisfied	30.28
Hop count [2]	Left-isotonicity	Hold	Satisfied	0.101
	Right-isotonicity	Hold	Satisfied	0.126
	Left-monotonicity	Hold	Satisfied	0.072
	Right-monotonicity	Hold	Satisfied	0.071
Compass routing [11]	Odd symmetry	Lack	Violated	0.345
	Transitivity	Lack	Violated	0.909
	Strict order	Lack	Violated	0.522
MFR [12]	Odd symmetry	Hold	Satisfied	0.269
	Transitivity	Hold	Satisfied	1.017
	Strict order	Lack	Violated	0.169
Adv. GFv1 [4]	Odd symmetry	Lack	Violated	1.079
Adv. GFv2 [4]	Odd symmetry	Hold	Satisfied	0.491
	Transitivity	Hold	Satisfied	0.487
NFP [13]	Odd symmetry	Lack	Violated	0.077
	Transitivity	Lack	Violated	0.951
	Strict order	Lack	Violated	0.268
RPF [14]	Odd symmetry	Lack	Violated	0.11
LPF [4]	Odd symmetry	Hold	Satisfied	0.002
	Transitivity	Hold	Satisfied	0.005
VFF [4]	Odd symmetry	Hold	Satisfied	5.2
	Transitivity	Hold	Satisfied	118.114

Chapter 6

Conclusion and Future work

The main contribution of this work is an automatic solution to checking compatibility of routing protocol and metric combinations. By using the automatic routing protocol checking method, we can know whether a new routing protocol is able to reach the optimality, consistency and loop-freeness goals when it is combined with metrics holding some particular metric properties. By using the automatic routing metric checking method, we can know whether a new metric implementation is able to work with a particular routing protocol. By comparing theoretical analysis results with the outputs of our automatic checking systems, we proved the correctness and effectiveness of our method.

In the next step, we plan to deploy our automatic protocol checking system to Hadoop for routing metric and protocol designers. Also, we hope to dig out more theoretical properties for routing metrics and protocols in order to make our system even more effective.

Chapter 7

Bibliography

[1] Ms Veena Bharti and Sachin Kumar. Survey of network protocol verification techniques. *International Journal of Scientific and Research Publications*, 2, 2012.

[2] Yaling Yang and Jun Wang. Design guidelines for routing metrics in multihop wireless networks. In *INFOCOM 2008. The 27th conference on computer communications. IEEE*, pages 1615–1623. IEEE, 2008.

[3] Mingming Lu and Jie Wu. Opportunistic routing algebra and its applications. In *INFOCOM 2009, IEEE*, pages 2374–2382. IEEE, 2009.

[4] Yujun Li, Yaling Yang, and Xianliang Lu. Routing metric designs for greedy, face and combined-greedy-face routing. In *INFOCOM 2009, IEEE*, pages 64–72. IEEE, 2009.

[5] Chuan Han and Yaling Yang. Compatibility between three well-known broadcast tree construction algorithms and various metrics. *Mobile Computing, IEEE Transactions on*, 10(8):1187–1199, 2011.

[6] Chris Arthur Lattner. *LLVM: An infrastructure for multi-stage optimization*. PhD thesis, University of Illinois at Urbana-Champaign, 2002.

[7] Silke Trißl and Ulf Leser. Fast and practical indexing and querying of very large graphs. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 845–856. ACM, 2007.

[8] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. *Symbolic model checking without BDDs*. Springer, 1999.

[9] Lucas Cordeiro, Bernd Fischer, and Joao Marques-Silva. Smt-based bounded model checking for embedded ansi-c software. *International Conference on Automated Software Engineering*, 0:137–148, 2009.

[10] Richard Draves, Jitendra Padhye, and Brian Zill. Routing in multi-radio, multi-hop wireless mesh networks. In *Proceedings of the 10th annual international conference on Mobile computing and networking*, pages 114–128. ACM, 2004.

[11] Evangelos Kranakis, Harvinder Singh, and Jorge Urrutia. Compass routing on geometric networks. In *in Proc. 11 th Canadian Conference on Computational Geometry*. Citeseer, 1999.

[12] Hideaki Takagi and Leonard Kleinrock. Optimal transmission ranges for randomly distributed packet radio terminals. *Communications, IEEE Transactions on*, 32(3):246–257, 1984.

[13] Ting-Chao Hou and Victor Li. Transmission range control in multihop packet radio networks. *Communications, IEEE Transactions on*, 34(1):38–44, 1986.

[14] Randolph Nelson and Leonard Kleinrock. The spatial capacity of a slotted aloha multihop packet radio network with capture. *Communications, IEEE Transactions on*, 32(6):684–694, 1984.