1-1-2009

Verifiable Policy-Based Routing With DRIVER

Anduo Wang
University of Pennsylvania

Changbin Liu
University of Pennsylvania

Boon Thau Loo
University of Pennsylvania, boonloo@cis.upenn.edu

Oleg Sokolsky
University of Pennsylvania, sokolsky@cis.upenn.edu

Prithwash Basu
Network Research Group, BBN Technologies

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-09-12.

Verifiable Policy-Based Routing With DRIVER

Abstract
The Internet today runs on a complex routing protocol called the *Border Gateway Protocol* (BGP). BGP is a *policy-based* protocol, in which autonomous Internet Service Providers (ISPs) impose their local policies on the propagation of routing information. Over the past few years, there has been a growing consensus on the complexity and fragility of BGP routing. To address these challenges, we present the *DRIVER* system for designing, analyzing and implementing policy-based routing protocols. Our system utilizes a *declarative network verifier* (DNV) which leverages declarative networking's connection to logic programming by automatically compiling high-level declarativen networking program into formal specifications, which can be directly used in a theorem prover for verification. In addition to verifying declarative networking programs using a theorem prover, the *DRIVER* system enables a similar transformation of verified formal specifications limited to fragment of second order logic to declarative networking programs for execution. As our main use case, we demonstrate the verification of a component-based specification of BGP protocol where *DRIVER* enables the analysis of convergence properties of Internet routing protocols with customizable policy configuration components. We show that the properties verified with *DRIVER* are indeed preserved in the synthesized implementation by performing experimental evaluation in a local cluster, where the equivalent declarative networking programs derived from the verified specifications displayed consistent behavior with regard to *DRIVER* verification.

Comments
University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-09-12.

Verifiable Policy-based Routing with DRIVER

Anduo Wang[1] Changbin Liu[1] Boon Thau Loo[1]
Oleg Sokolsky[1] Prithwish Basu[2]

[1] Computer and Information Sciences Department, University of Pennsylvania,
3330 Walnut Street, Philadelphia, PA 19104-6389
[2] Network Research Group, BBN Technologies,
10 Moulton Street, Cambridge, MA 02138
{anduo,changbl,boonloo,sokolsky}@seas.upenn.edu pbasu@bbn.com

Abstract. The Internet today runs on a complex routing protocol called the *Border Gateway Protocol* (BGP). BGP is a *policy-based* protocol, in which autonomous Internet Service Providers (ISPs) impose their local policies on the propagation of routing information. Over the past few years, there has been a growing consensus on the complexity and fragility of BGP routing. To address these challenges, we present the *DRIVER* system for designing, analyzing and implementing policy-based routing protocols. Our system utilizes a *declarative network verifier* (DNV) which leverages declarative networking's connection to logic programming by automatically compiling high-level declarativen networking program into formal specifications, which can be directly used in a theorem prover for verification. In addition to verifying declarative networking programs using a theorem prover, the *DRIVER* system enables a similar transformation of verified formal specifications limited to fragment of second order logic to declarative networking programs for execution. As our main use case, we demonstrate the verification of a component-based specification of BGP protocol where *DRIVER* enables the analysis of convergence properties of Internet routing protocols with customizable policy configuration components. We show that the properties verified with *DRIVER* are indeed preserved in the synthesized implementation by performing experimental evaluation in a local cluster, where the equivalent declarative networking programs derived from the verified specifications displayed consistent behavior with regard to *DRIVER* verification.

1 Introduction

The Internet today runs on a complex routing protocol called the *Border Gateway Protocol* or *BGP* in short. BGP enables Internet-service providers (ISP) world-wide to exchange reachability information to destinations over the Internet, and simultaneously, each ISP acts as an autonomous system that imposes its own import and export policies on route advertisements exchanged among neighboring ISPs.

Over the past few years, there has been a growing consensus on the complexity and fragility of BGP routing. Even when the basic routing protocol convergences, conflicting policy decisions among different ISPs have lead to route oscillation and slow convergence. Several empirical studies such as reference [12] have shown that there are prolonged periods in which the Internet cannot reliably route data packets to specific

destinations due to routing errors induced by BGP. In response, the networking community has proposed several Internet architectures and policy mechanisms (e.g. [17, 6]) aimed at addressing these challenges.

Given the proliferation of proposed techniques, there is a growing interest in formal software tools and programming frameworks that can facilitate the design, implementation, and verification of routing protocols. These proposals can be broadly classified as: (1) algebraic and logic frameworks (e.g. [10]) that enable protocol correctness check in the design phase; (2) runtime debugging platforms that provide mechanisms for runtime verification and distributed replay, and (3) programming frameworks that enable network protocols to be specified, implemented, and in the case of the Mace toolkit, verified via model checking [11].

In this paper, we present the *DRIVER* (*Declarative Routing Implementation and VERification*) system for designing, analyzing and implementing network protocols within a unified framework. Our work is a significant step towards *bridging* network specifications, protocol verification, and implementation within a common language and system. The *DRIVER* framework achieves this unified capability via the use of *declarative networking* [14, 13], a declarative domain-specific approach for specifying and implementing network protocols, and theorem proving, a well established verification technique based on logical reasoning.

DRIVER leverages our prior work on a *declarative network verifier (DNV)* [19] which demonstrates that one can leverages declarative networking's connection to logic programming by automatically compiling high-level declarative networking program written in the Network Datalog (*NDlog*) query language into formal specifications, which can be directly used in a theorem prover for verification. The proving process guided by the user is then carried out in a general-purpose theorem prover and proofs are mechanically checked. Declarative networking programs that have been verified in *DRIVER* can be directly executed as implementations, hence bridging specifications and implementations within a unified declarative framework.

In addition to verifying declarative networking programs using a theorem prover, the *DRIVER* system enables a similar transformation of verified formal specifications (limited to fragment of second order logic) to *NDlog* program for execution. This enables a network designer to either directly verify network implementation specified in *NDlog* or conceptualize and verify the design of a network in components aided by a theorem prover prior to implementation.

Theorem proving provides an expressive and powerful verification framework that is particularly well-suited for analyzing the complexities of BGP policies. We introduce a component-based specification of BGP system that is based on the BGP model first proposed by Griffin *et al.* [18]. Components modularize our analysis, enable reuse of code, hence enable us to study the impact of import and export policies on overall protocol behavior. For example, we demonstrate the use of *DRIVER* and component-based reasoning for detecting instances where policy conflicts may lead to protocol divergence, a well-known limitation of the existing BGP system. In *DRIVER*, verified component-based specifications of specific BGP protocol can be easily translated to equivalent *NDlog* programs for execution. We experimentally validate verified properties based on our BGP analysis in equivalent *NDlog* programs executed and evaluated in a local cluster using the P2 declarative networking engine [1].

2 Background

In this section, we will provide a brief overview of declarative networking. Declarative networks are specified using *Network Datalog* (*NDlog*), a distributed logic-based recursive query language first introduced in the database community for querying network graphs. In prior work, it has been shown that traditional routing protocols can be specified in a few lines of declarative code [14], and complex protocols such as Chord DHT in orders of magnitude less code [13] compared to traditional imperative implementations. This compact and high-level specifications enables rapid prototype development, ease of customization, optimizability, and the potentiality for protocol verification. When executed, these declarative networks perform efficiently relative to imperative implementations, as demonstrated by the *P2* declarative networking system [1].

2.1 Datalog Language

NDlog is primarily a distributed variant of Datalog. We first provide a short review of Datalog, following the conventions in Ramakrishnan and Ullman's survey [16]. A Datalog program consists of a set of declarative *rules*. Each rule has the form `p :- q1, q2, ..., qn.`, which can be read informally as "`q1` and `q2` and `...` and `qn` implies `p`". Here, `p` is the *head* of the rule, and `q1`, `q2`,...,`qn` is a list of *literals* that constitutes the *body* of the rule. Literals are either *predicates* with *attributes* (which are bound to variables or constants by the query), or boolean expressions that involve function symbols (including arithmetic) applied to attributes. In Datalog, rule predicates can be defined with other predicates in a cyclic fashion to express recursion. The order in which the rules are presented in a program is semantically immaterial; likewise, the order predicates appear in a rule is not semantically meaningful. Commas are interpreted as logical conjunctions (`AND`). The names of predicates, function symbols, and constants begin with an lowercase letter, while variable names begin with an upper letter.

2.2 Path-vector Protocol

We present an example *NDlog* program that implements the *path-vector* protocol.

```
p1 path(@S,D,P,C):-link(@S,D,C),p=f_init(S,D).
p2 path(@S,D,P,C):-link(@S,Z,C1), path(@Z,D,P2,C2),C=C1+C2,
                   P=f_concatPath(S,P2), f_inPath(P2,S)=false.
p3 bestPathCost(@S,D,min<C>):-path(@S,D,P,C).
p4 bestPath(@S,D,P,C):-bestPathCost(@S,D,C), path(@S,D,P,C).
```

The program takes as input `link(@S,D,C)` tuples, where each tuple corresponds to a copy of an entry in the neighbor table, and represents an edge from the node itself (`S`) to one of its neighbors (`D`) of cost `c`. *NDlog* supports a *location specifier* in each predicate, expressed with "`@`" symbol followed by an attribute. This attribute is used to denote the source location of each corresponding tuple. For example, `link` tuples are stored based on the value of the `S` field.

Rules `p1-p2` recursively derive `path(@S,D,P,C)` tuples, where each tuple represents the fact that the path from `S` to `D` is via the path `P` with a cost of `C`. Rule `p1` computes one-hop reachability trivially given the neighbor set of `S` stored in `link(@S,D,C)`. Rule `P2` computes transitive reachability as follows: if there exists a link from `S` to `Z`

with cost C1, and Z knows about a shortest path P2 to D with cost C2, then transitively, S can reach D via the path f_concatPath(S,P2) with cost C1+C2. Note that p1–p2 also utilize two list manipulation functions to maintain path vector p: f_init(S,D) initializes a path vector with two elements S and D, while f_concatPath(S,P2) prepends S to path vector P2.

Rules p3–p4 take as input hop tuples generated by rules p1–p2, and then derive the hop along the path with the minimal cost for each source/destination pair. The output of the program is the set of bestPath(@S,D,Z,C) tuples, where each tuple stores the next hop Z along the shortest path from S to D. To prevent computing paths with cycles, an extra predicate $f\_inPath(P,S) = false$ is used in rule p2, where the function $f\_inPath(P,S)$ returns true if node *S* is in the path vector *P*.

Event predicates are used to denote transient tables which are used as input to rules but not stored. For example, utilizing the built-in periodic keyword , the following rule enables node X to generate a ping event every 10 seconds to its neighbor Y denoted in the link(@X,Y) predicate: ping(@Y,X) :- periodic(@X,10), link(@X,Y).

3 Overview of DRIVER

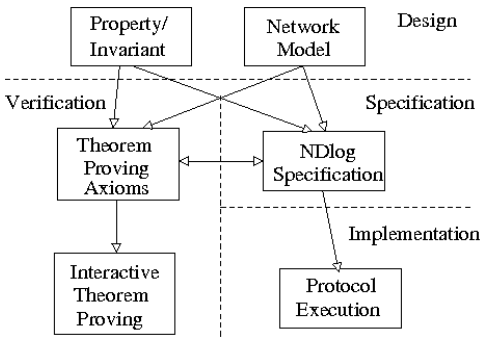

Fig. 1. Overview of DRIVER

Figure 1 provides an overview of *DRIVER*'s basic approach towards unifying specifications, verification, and implementation within a common declarative framework. The approach is broken up into the following four phases: *design*, *specification*, *verification*, and *implementation*.

In the initial design phase of *DRIVER*, a network designer develops a conceptual model for the routing protocol. In practice, this step may be optional, but having such a model is often useful both from the implementation standpoint, and for verifying one's protocol design, as we will demonstrate in Section 4.

Based on the design, two options are available. First, *NDlog* networking programs can be synthesized from the design, and then the *NDlog* implementations can be directly verified in an theorem prover. Second, the designer can first verify the design using a theorem prover and then automatically generate the corresponding *NDlog* program.

Considering the first option, *DRIVER* takes as input *NDlog* program representation of the routing protocol we are interested in. In order to carry out the formal verifica-

tion process, the *NDlog* programs are automatically compiled into formal specifications recognizable by a standard theorem prover (e.g. PVS [2], Coq [3]) using the *axiom generator*, as depicted in the left-part of Figure 1.

At the same time, the protocol designer specifies high-level invariant properties of the protocol to be checked via two mechanisms: invariants can be written directly as theorems in the theorem prover, or expressed as *NDlog* rules which can be automatically translated into theorems using the axiom generator. The first approach increases the expressiveness of invariant properties, where one can reason with invariants that can be only expressible in higher order logic. The second approach has restricted expressiveness based on *NDlog*'s use of Datalog, but has the added advantage that the same properties expressed in *NDlog* can be verified in both theorem prover and checked at runtime.

From the perspective of network designers, as depicted in the left part of Figure 1, they reason about their protocols using the high-level protocol specifications and invariant properties.The *NDlog* high-level specifications are directly executed and also proved within the theorem prover. Any errors detected in the theorem prover can be corrected by changing the corresponding *NDlog* programs. Our initial *DRIVER* prototype uses the PVS theorem prover, due to its substantial support for proof strategies which significantly reduce the time required in the interactive proof process. However, the techniques describe in this paper are agnostic to other theorem provers. We have also verified some of the properties presented in this paper using the Coq [3] proof assistant.

As a second option, *DRIVER* allows the network designer to first utilize a theorem prover to check the protocol design. This requires a network designer first develop formal specifications for the routing protocol of interests. Once the formal representation of the protocol is verified by the prover, corresponding *NDlog* programs are then generated for execution. Similar to the first option, this approach is made possible by the use of *NDlog*, which is particularly amenable to the translation into formal specification recognizable by existing theorem provers (and vice versa), due to its logic-based nature.

Reference [19] provides details on the translation process from *NDlog* programs into formal specification in theorem prover, as well as several verification use cases for standard network routing protocols. In the rest of the paper, we focus on the second approach, using policy-based routing as our driving example.

4 Verifying Policy-based Inter-domain Routing

We present a compositional approach towards the verification of policy-based BGP routing protocol. Our approach demonstrates the second option in *DRIVER* verification as described in Section 3, where we begin with a component-based model of BGP protocol, then formalize and verify the BGP components in theorem prover PVS, and finally translate the verified BGP components into *NDlog* for execution.

BGP assumes a network model in which routers are grouped into various Autonomous Systems (*AS*) administrated by Internet Server Provider (*ISP*). Individual ASes exchange route advertisements with neighboring ASes using the path-vector protocol described in Section 2.2. Upon receiving a route advertisement, an AS may choose to accept or ignore the advertisement based on its *import policy* dictated by its business

considerations and peering agreements. Similarly, an AS may choose to export only selected routes that it knows to its neighboring ASes. The route advertisements received by each node is then used to compute the next hop (neighboring AS) along the best paths to each destination.

We have selected BGP as one of our main use cases as complex policy interactions have been known to result in delayed protocol convergence. The problem is exacerbated by the lack of global knowledge on policies. While *DRIVER* does not directly address the lack of global knowledge, our goal here is to demonstrate that *DRIVER* provides a clean foundation for network administrators to formally reason about such policy interactions and their impact on protocol convergence. Moreover, the verified specifications are directly implementable as declarative networks.

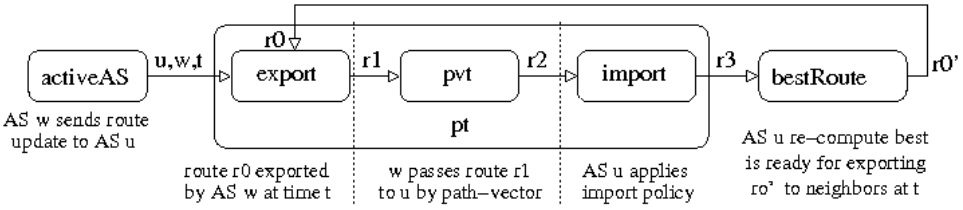

Fig. 2. Overview of the BGP model

In our formalization, we adopt Griffin's BGP model [9, 18], which views BGP protocol as a series of *route transformations*, where each transformation is represented as a component that takes as input received routes, performs internal transformation based on the component specifications, and generates the output routes.

Figure 5 shows the general structure of a component that takes as m input routes $i_1, \cdots, i_m$, applies additional transformation constraints, and generates n output routes $o_1, o_2, \cdots, o_n$. This transformation from m inputs to n outputs is captured by a predicate $TRANS(i_1, i_2, \cdots, i_m, o_1, o_2, \cdots, o_n)$, such that the predicate is true if the given m input and n output values satisfies all the constraint predicates specified in $TRANS$.

Our use of components enables us to modularize our analysis, hence enabling us to study the impact of import and export policies on overall protocol behavior. This compositional approach towards verification is well-explored in the formal methods community, and has been successfully used in domains such as hardware verification [2]. In addition, by formalizing BGP protocol in a component-based fashion, it eases the translation of the verified PVS formalization to the equivalent *NDlog* rules for execution.

Figure 2 shows an overview of the abstract BGP model, which consists of the following five components: `activeAS`, `export`, `pvt`, `import`, and `bestRoute`. The triggering component `activeAS(u,w,t)` specifies at time `t`, AS `w` advertises its current best routes `r0` to each neighbor AS `u`. The export policies are imposed by the `export` component) where output route `r1` is generated. This is followed by route propagation via the path-vector protocol (`pvt`) component which transforms route `r1` to `r2`. Upon receiving the route advertisement `r2`, AS `u` applies its import policies via the `import` component and produces route `r3`. Finally, in the `bestRoute` component, AS `u` recomputes/selects its best route based on the all new route advertisement

r3. In the next iteration t+1, the same process repeats itself when triggered by the activeAS(u,w,t+1) event.

One should view activeAS as a periodic event predicate (see Section 2.2) that is invoked at each AS node to propagate its routes to neighbors, and all routes propagated have a lifetime set to the duration of the period. If a current route is no longer valid, it will no longer be advertised in the next time period.

4.1 Component-based Specification Generation

First consider the *peer-transformation* component (pt), which consists of three sub-components (export, pvt, and import). The equivalent PVS specification is as follows:

```
pt(u,w,r0,r3,t): INDUCTIVE bool =
EXISTS (r1,r2): activeAS(u,w,t) AND export(u,w,r0,r1,t)
  AND pvt(u,w,r1,r2,t) AND import(u,w,r2,r3,t) AND bestRoute(w,t,r0)
```

The concept of peer-transformation was proposed in the original BGP model, and involves neighboring ASes exchanging routes and updating their respective routing tables. pt(u,w,r0,r3,t) represents the route propagation between neighboring AS u and w, such that AS w advertises its best route r0 to its neighbor AS u at time t. The advertised route r0 undergoes a series of transformation and eventually is received by u as route r3.

More specifically, the route propagation is first triggered by activeAS(u,w,t). The predicate export(u,w,r0,r1,t) is true if w can export route r0 to u at time t, in which case, the exported route would be r1. Route r1 is then propagated via the path-vector component pvt, resulting in a new route r2 with u prepended to r1. In the final transformation, an import policy is applied at u, in which the route r2 is accepted as r3 based on the policies specified at u. Note that our formalization assumes that the sub-components are executed within the same time period t before the next round of route updates. This is consistent with the idealized BGP model described in prior works.

Given the above pt component, we next proceed to define each sub-component as follows:

```
export(u,w,r0,r1,t): INDUCTIVE bool =
  activeAS(u,w,t) AND exportPolicy(w,r0,r1) AND bestRoute(w,r0,(t-1))

pvt(u,w,r1,r2,t): INDUCTIVE bool =
  (NOT member (u, as_path(r1))) => as_path(r2)=cons(u,as_path(r1))
    AND dst(r2)=dst(r1) AND loc_pref(r2)=loc_pref(r1)

import(u,w,r2,r3,t): INDUCTIVE bool =
  EXISTS r1: pvt(u,w,r1,r2,t) AND importPolicy(u,r2,r3)
```

Note that BGP route consists of additional parameters is defined as a record in the PVS as follows:

```
routeRecord: TYPE =
[#dst: nat,as_path: list[AS],loc_pref: nat#]
```

The above type definition indicates that each route record (used as `r`, `r0`, ..., `r3` variables) has three parameters: `dst` denotes the destination of the route, `as_path` the AS path, and `loc_pref` the local preference that are manipulated by export and import policies[3].

One of the main takeaways from the prior PVS formalization is its natural composability given the BGP model, where components are defined and expanded in a top-down fashion. The `export` and `import` sub-components are further defined by their respective policies `exportPolicy` and `importPolicy` which are supplied based on the actual policy configurations.

The path-vector transformation (`pvt`) component takes as input route `r1`, and generates a new route `r2` with the same destination address (`dst`) and local preferences (`loc_pref`). At the same time, `u` is prepended to the input route `r1`.

Finally, we define the `bestRoute` component, which unlike the earlier transformations, is a select operation determining the best route for a given destination applied after aggregation. The PVS specifications are as follows:

```
bestRoute: AXIOM
route(u,t,r1) <=> (EXISTS r2: (bestRoute(u,t-1,r2) AND r1=r2)
  AND (FORALL w: NOT activeAS(u,w,t))) OR
(EXISTS (r2,r4): (pt_best(u,t,r4) AND bestRoute(u,t-1,r2) AND
  (EXISTS w: activeAS(u,w,t) AND
   NOT conflicting(u,w,t)) AND best(r4,r2,r3)) AND r1=r3) OR
(EXISTS (r3): (pt_best(u,t,r3) AND
  (EXISTS w: activeAS(u,w,t) AND conflicting(u,w,t)) AND r3=r1))
```

The above axiom indicates that an AS can compute its bestRoute under three scenarios (connected by the logical `OR`):

Case 1: If the AS did not receive any route updates from any neighbors in the previous time period, the bestRoute computed in the previous time period should be retained as the best route in the current period.

Case 2: If the AS is active and receives route updates from some of its neighbors, then the new best route is selected from its previous best route and route updates received.

Case 3: Similar to case 2, except that the AS receives a route update that conflicts (based on policies defined in the `conflicting` predicate) with its current best route. In this case, the current best route is invalidated and a new one is computed.

Next, we present the definition of `conflicting` `conflicting` in PVS as follows:

```
conflicting(u,v,t): INDUCTIVE bool =
EXISTS (w,r1,r2,r3): bestRoute(u,(t-1),r1) AND pt(u,w,r2,r3,t)
  AND (member(u,as_path(r3))) AND (member(w,as_path(r1)))
```

A conflicting situation occurs when the best route at two neighboring ASes both set their path to go through its neighbor. Such circulated routes should be dropped and re-computation of best route is required. Note that we have explicitly define conflicting condition at neighboring ASes that are caused by the importing policies. The circulated route dropping is implicit in Griffin's original presentation.

The definition of `pt_best` as follows:

[3] Note that the equivalent *NDlog* program expands each record definition into attributes in the predicate since it does not support nested tuples.

```
pt_best(u,t,r4): INDUCTIVE bool = EXISTS (w,r0): pt(u,w,r0,r4,(t-1))
    AND (FORALL r3: pt(u,w,r0,r3,(t-1)) => best(r3,r4,r4))
```

which says at AS `u` time `t`, `r4` is the best route among all routes received by component `pt`, where `best` is defined as follows:

```
best(r1,r2,r3): INDUCTIVE bool =
(loc_pref(r1)>loc_pref(r2) => r3=r1) OR
(loc_pref(r2)>loc_pref(r1) => r3=r2) OR
(loc_pref(r1)=loc_pref(r2) =>
  (length(as_path(r1)>length(as_path(r1))) => r3=r2)) OR
  (length(as_path(r1)<length(as_path(r1))) => r3=r1)) OR
  (length(as_path(r1)=length(as_path(r1))) => r3=r2))
```

which selects the route with higher local preference, or route with shorter path if the values of local preference are the same.

4.2 Analyzing Policy Conflicts

Figure 3 shows a specific network of three ASes based on the `Disagree` scenario [9, 18]. This scenario leads to delayed convergence when two neighboring ASes have conflicting policies. We demonstrate the use of *DRIVER* to verify that delayed convergence indeed occurs, and can be further generalized to an arbitrary network.

Specific Instance of Delayed Convergence Figure 4 shows the sequence of best route updates that leads to convergence delay. The delay is caused by route propagation sequence and the conflicting import policies at AS `1` and `2`. These two ASes prefer to traverse each other to a common destination `0`, hence violating `conflicting` defined in Section 4.1. From time `1` to `3`, we note that AS `1` oscillates between two best paths `[1,0]` and `[1,2,0]`, and this oscillation can repeat indefinitely.

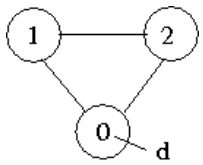

Fig. 3. Example BGP system: `Disagree`

time `t`	`as_path` of best route at `1`	`as_path` of best route at `2`
0	[ ]	[ ]
1	[1,0]	[2,0]
2	[1,2,0]	[2,1,0]
3	[1,0]	[2,0]
...	...	...
T	[1,0]	[2,0]
T+1	[1,2,0]	[2,0]
...	[1,2,0]	[2,0]

Fig. 4. Route Updates over Time

Similarly, AS `2` oscillates between paths `[2,0]` and `[2,1,0]` indefinitely. Again the oscillation is caused by the inherent policy conflict: the best paths of `1,2,0]` and `[2,1,0]` computed at time `2` contradict each other. The conflict however can be detected and resolved at each AS locally based on route updates. For instance, when AS `1` receives a route update for `[2,1,0]`, it will invalidate its existing best route `[1,2,0]` and replace it with `[1,0]`, resulting in the oscillation above. The protocol eventually convergences at time `T` given a different ordering of route updates, where only one of the two ASes sends its route updates to the other but not vice versa.

Formally Verifying Route Oscillation Due to the compositional structure of our BGP specifications, The verification of the above behavior in PVS only requires the specification of conflicting policies at AS 1 and AS 2. More specifically, the policies at AS 1 are defined as follows:

```
importPolicy(u, r1, r2): INDUCTIVE bool =
(u=1 AND as_path(r1)=[1,0] AND dst(r2)=dst(r1)
        AND as_path(r2)=as_path(r1) AND loc_pref(r2)=dlp) OR
(u=1 AND as_path(r1)=[1,2,0] AND dst(r2)=dst(r1)
        AND as_path(r2)=as_path(r1) AND loc_pref(r2)=dlp+1)
```

Where AS 1 assigns higher local preference to route received from 2 (those with as_path=[1,2,0]), i.e. AS 1 prefers route going through its neighbor 2. The conflicting policies at AS 2 that prefers route via neighbor 1 is defined similarly.

To demonstrate delayed convergence, first we need to show that route oscillation such as that shown in time 1 to 3 in Figure 4 occurs. This is captured by the follows PVS theorem:

```
route_oscillate_1: THEOREM
FORALL t:
  (t>=1) AND route(1,t,r1) AND route(2,t,r2) AND
  as_path=(r1)=[1,0] AND as_path(r2)=[2,0]
  => (route(1,(t+1),r3) => as_path(r3)=[1,2,0])
```

which says at time t ASes 1 and 2 have routes [1,0] and [2,0] respectively implies that at the next time t+1, AS 1's route would be set to [1,2,0]. This theorem can be proven easily in PVS by expanding import definition introduced in Section 4.1. Similarly, we prove a similar theorem that the best route at 2 will revert back to [2,1,0] in the next period, hence resulting in the oscillation.

With these two theorems, we further prove the general divergence theorem at AS 1 as follows:

```
route_diverge_1: THEOREM
FORALL t: t>=1 AND route(1,t,r1) AND route(1,(t+1),r2)
  => NOT (r1=r2)
```

The theorem states that at any time t, the route r1 stored at AS 1 will be replaced by a different route r2 in the next round of route exchange at time (t+1). We can specify and prove divergence at AS 2 in a similar way.

Formally Verifying Delayed Convergence The Disagree scenario eventually converges once an alternative sequence of route updates occurs. For instance, in one time period, either AS 1 or 2 receives neighbor updates but not both. Under this circumstances, the following convergence theorem can be proved:

```
route_converge: THEOREM
EXIST T: (FORALL t: t>=T =>
     (EXISTS r1: route(1,t,r1)=>route(1,t+1,r1)))
```

which states that there exists an initialization time T for Disagree to reach convergence state such that best route at AS 1 will reach r1 and keep r1 as its best route from then on.

Note that, due to the componentized nature of our formalization, the BGP axioms specified in Section 4.1 are reused in its entirety.

4.3 Generalizing the Conflict Analysis

Our earlier analysis demonstrates that delayed convergence due to route oscillation occurred in a three-AS network. In this section, we demonstrates the strength of theorem proving to scale to large and even infinite large network. Interestingly, while such a generalization proof is simple to achieve in theorem prover, it would be much harder to perform such checks using a model checker due to the state explosion problem. We also benefit from the formal argument by adopting component-based specification of BGP.

The basic observation is that route oscillation phenomenon that causes convergence delay can be generalized to arbitrary large network given the `Disagree`-like topology depicted in Figure 3 occurs: two connected ASes `1` and `2` have their paths `p1,p2` to reach a third AS `0`, and both `1` and `2` prefers routes via each other. By using component-based specification generation, we can reuse all the codes we present in 4.1, and modify the policy conflicts in Section 4.2 as follows. We show the generalized policy at AS `1` as example:

```
importPolicy(u, r1, r2): INDUCTIVE bool =
(u=1 AND (NOT member(2,as_path(r1))) AND dst(r2)=dst(r1)
  AND as_path(r2)=as_path(r1) AND loc_pref(r2)=dlp) OR
(u=1 AND member(2,as_path(r1)) AND dst(r2) = dst(r1)
  AND as_path(r2)=as_path(r1) AND loc_pref(r2)=dlp+1)
```

The above definition states that AS `1` would assign a higher local preference to routes that includes AS `2` in the path. It is essentially a more general version of the earlier import policy, because there is no restrictions that `1` and `2` be immediate neighbors as long as they are connected.

Based on this generalized import policy, we show route oscillation by prove the `route_diverge_1` theorem we have seen in Section 4.2. Not surprising, the proof also follows exactly the same structure as the one for `Disagree`. Our experience in proving route oscillations on a generalized network demonstrates the strength of theorem proving in formal argument and formal proof reuse.

4.4 Verifying a BGP Alternative

To demonstrate the flexibility and expressiveness of *DRIVER*, we outline an additional use case based on a BGP alternative called *Hybrid Link-state and Path-vector Protocol (HLP)* [17]. As its name suggests, HLP is a variant of link-state and path-vector protocol. Intuitively, in HLP, the network is organized into domains of ASes, where within each domain, ASes maintain a customer-provider hierarchy and execute the link-state (LS) protocol. Across domains, peer nodes run the *fragmented path vector (FPV)* protocol to exchange routes, where the internal structure of each domain is hidden.

Due to the compositional nature of our specifications, given LS and PV sub-components, HLP can be specified with minimal changes as follows:

```
FPV(u, p, c): INDUCTIVE bool =
(EXISTS (p1,w):  AND PEERS(u,w) AND (LSA(w, p1, (c-1))
  AND p=[u,w]) OR (FPV(w,p1,(c-1)) AND p=cons(u,p1))) OR
(EXISTS (p1,w): FPV(w, p1, (c-1)) AND
  DirectCP(w,u) and not member(u,p1) AND p= cons(u,p1))
```

`LSA(w,p,c)` says the path `p` computed by link state protocol which is used by node `w` to reach `d`.

FPV(w,p,c) says that path p computed by fragmented path vector protocol is used by w to reach d. The above definition therefore states that a HLP path p used to reach d at node u can be derived in the following two ways as connected by the logical OR:

Peer AS updates: AS u receives a route advertisement from its immediate neighbor (peer) in the form of a link state update (LSA) or a fragmented path vector update (FPV). AS u can then conclude that path (u,p1) with cost c can be used to reach d, i.e. FPV(u,p,c) can be derived.

Provider updates: AS u received a HLP route advertisement in the form of a fragmented path vector update FPV(w,p1,(c−1)) from its provider w, then u can conclude that it would be able to reach d with path (u,p1) with cost c.

Based on this definitions, we can then verify HLP is loop-free by proving the following PVS theorems:

```
export_guideline: THEOREM
FPV(u,p,c) => NOT (exists (p1,w): FPV(w,p1,(c-1))
    AND p=cons(u,p1) and DirectCP(u,w))
```

which states that if u can derive a new route FPV(u,p,c) by receiving advertisement FPV(w,p1,(c−1)) from w, u cannot be w's provider, i.e. w cannot advertise to its provider u. By unfolding HLP definitions in PVS, the proof process can be carried out easily within *DRIVER*. Note that this theorem shows HLP adheres to the following *route export guideline*: at any node u, route advertised by its peer or provider is not forward to another provider.

5 *NDlog* Program Generation and Experimental Validation

Reference [19] demonstrates a natural translation from *NDlog* programs to formal specification recognizable by PVS based on the proof-theoretic semantics of Datalog. Interestingly, by adopting a component-based approach, as we have done in Section 4 for the BGP system, there is a straightforward translation from PVS formalization to *NDlog* programs. This section briefly outlines the basic translation, which serves as the basis for generating equivalent *NDlog* programs from verified PVS formalization. We validate the translation based on an actual experimental validation of the generated *NDlog* programs in a local cluster testbed.

5.1 PVS to NDlog Translation

Fig. 5. Component with n inputs and m outputs

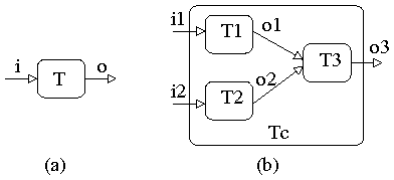

Fig. 6. PVS Components to *NDlog*

To illustrate the translation, we make use of an example component T shown in Figure 6 (a) with input i and output o. The PVS axiom is as follows:

```
T(i,o):INDUCTIVE bool = Constraint C(o,i)
```

The above definition specifies T as a relation over i,o in terms of constraint C. Note that C itself may be a conjunction of predicates that denote sub-components.

The equivalent *NDlog* rule is as follows:

```
T T_out(o) :- T_in(i), Constraint C(o,i)
```

The above rule specifies component T as a rule T that take T_in(i) as input and output T_out(o) if constraint c is also satisfied. Obviously, this equivalent translation can be easily generalized to components with multiple inputs and outputs.

Next, consider a component consists of several sub-components, as depicted in Figure 6 (b). Example component Tc is built from sub-components T1,T2,T3. The PVS representations are as follows:

```
T1(i,o): INDUCTIVE bool = Constraint C1(i,o)
T2(i,o): INDUCTIVE bool = Constraint C2(i,o)
T3(i,i',o): INDUCTIVE bool = Constraint C3(i,i',o)

Tc(i1,i2,o3): INDUCTIVE bool = EXISTS (o1,o2):
  T1(i1,o1) AND T2(i2,o2) AND T3(o1,o2,o3)
```

The above PVS definition specifies component Tc as a relation Tc in terms of T1,T2,T3, where C1,C2,C3 are constraints imposed by the sub-components. This composition can be compiled into the following equivalent NDlog rules:

```
t1 T1_out(o1) :- T1_in(i1), C1(i1,o1).
t2 T2_out(o2) :- T2_in(i2), C2(i2,o2).
t3 T3_out(o3) :- T3_in(o1,o2), C3(o1,o2,o3).
```

The above rules t1–t3 specifies the sub-components as three *NDlog* rules. Note that the predicates C1, C2, and C3 may themselves be a set of conjunctive predicates used to denote sub-components. And the notion of component Tc is implicitly implied in the three rules where T1, T2 takes input i1,i2, and T3 outputs o3.

To annotate the above *NDlog* program with the appropriate location specifiers and materialized lifetimes, additional predicate schema information is required as input for the translation process.

5.2 Experimental Evaluation

We perform an experimental evaluation of equivalent *NDlog* declarative networking programs that implement the BGP system verified in Sections 4.1-4.3. As noted earlier, our component-based approach enables a straightforward mapping from PVS specifications to *NDlog* rules for execution.

Our evaluation setup is based on execution of the *NDlog* program in a local cluster. It consists of 15 Pentium IV 2.8GHz PCs with 2GB RAM running Fedora 9 with kernel version 2.6.23, which are interconnected by high-speed Gigabit Ethernet. For our experiments, we emulate a 60-node network, where each node is an instance of the *P2* declarative networking engine [1] executing the *NDlog* programs. Each one of these 60 nodes runs an instance of the *P2* system, and is deployed on one of the 15 physical machines in the cluster (i.e. each physical machine executes 4 *P2* instances).

The main goal of our evaluation is to validate that divergence due to route oscillation can happen in practice in the presence of policy conflicts, as we have verified

with *DRIVER* in Section 4.2. Our equivalent *NDlog* program mapped from the PVS specifications consists of 13 *NDlog* rules.

Our input network consists of a 60-node network with 90 random bi-directional links (i.e. the average degree of each node is 3). Given this topology, our experimental setup executes the *NDlog* program such that all paths are computed (i.e. protocol converges). To measure bandwidth utilization required for convergence, each *NDlog* program executed in a distributed fashion given the input topology until all routes have been computed.

We experiment with two policy configurations: *25%-conflict* and *50%-conflict*, meaning that 25% and 50% respectively of all best paths are a result of policy conflicts (when two neighboring nodes prefer each other to a common destination as opposed to selecting the shortest paths). Our baseline (best case convergence) for comparison is *No-conflict*, where all-pairs shortest paths are computed without any conflicting policies.

In our experiments, we measure the bandwidth utilization over time (sampled at 0.1s interval) when executing the *NDlog* program to compute all routes until the protocol converges (i.e. no more route changes in the network). We make the following observations. First, as the percentage of conflicts increases, the protocol takes longer time to converge. While *No-conflict* converges in 0.3 seconds, *25%-conflict* and *50%-conflict* require 3.2 seconds and 3.9 seconds respectively to converge. Moreover, the protocols that involve such conflicts utilize significantly more bandwidth. The aggregate bandwidth utilization for *No-conflict* , *25%-conflict* and *50%-conflict* is 6 MB, 38 MB and 55 MB, respectively. In the extreme case where all computed routes are a result of policy conflicts, the convergence time is even longer, at 5.5 seconds, requiring 122MB in bandwidth.

In general, our experimental evaluation demonstrates that the delayed convergence proved formally in Section 4.2 is indeed observed in the actual *NDlog* implementation. This matched behavior suggests the utility of the overall *DRIVER* system: formally verified BGP protocols can be easily synthesized into implementations that preserve the verified properties. Furthermore, one can measure the performance implications (e.g. bandwidth utilization and actual convergence time) with regard to the properties that have been formally verified.

6 Related Work

We briefly compare *DRIVER* with existing works on network protocol verification and recent development of declarative networks.

Model checking is a collection of algorithmic techniques for checking temporal properties of system instances based on exhaustive state space exploration. Recent advances in model checking network protocol implementations include *MaceMC* [11] and *CMC* [7]. Compared to *DRIVER*'s use of theorem proving, these approaches are *sound* as well, but not *complete* in the sense that the large state space persistent in network protocols often prevents complete exploration of the huge system states. They are typically inconclusive and restricted to small network instances and temporal properties.

Classical theorem proving has been used in the past few decades for verification of network protocols (e.g. [2, 5, 8]). Despite extensive work, this approach is generally restricted to protocol design and standards, and cannot be directly applied to protocol implementation. A high initial investment based on domain expert knowledge is often required to develop the system specifications acceptable by some theorem prover (up to

several man-months). Therefore, even after successful proofs in the theorem prover, the actual implementation is not guaranteed to be error-free. *DRIVER* is hence a significant improvement over existing usage of theorem proving which typically require several man-months to develop the system specifications, a step that is reduced to a few hours through the use of declarative networking.

Runtime verification techniques provide a mechanism for checking at runtime that a system does not violate expected properties. Since declarative networks utilize a distributed query engine to execute its protocols, these checks can be expressed as *monitoring queries* in *NDlog*. However, any runtime verification scheme will incur additional runtime overheads, and subtle bugs may require a long time to be encountered. Moreover, the properties can be checked in this case are restricted to those can be expressed in *NDlog*.

The recently proposed *cardinality abstraction technique* [15] introduces an abstraction based on the operational semantics of the P2 declarative networking implementation. It is used to perform the checking of counting-based invariants of a Byzantine fault tolerant protocol written as P2 declarative network. Our *DRIVER* system has several advantages over cardinality abstraction. First, by adopting proof-theoretic semantics of Datalog, we have a natural translation to formal specifications that does not rely on any particular interpretation of the operational semantics of P2. Second, by utilizing theorem proving and higher-order logic, *DRIVER* is more expressive and flexible, while verification based on cardinality abstractions are limited to counting-based invariants. Finally, given the flexibility and generality of *DRIVER*, to our best knowledge, *DRIVER* is the first verification framework that enables complex policy based routing protocol deployed among autonomous ISPs to be formally analyzed, and in addition, verified specifications can be directly translated into declarative networking programs for execution.

In summary, compared with existing tools, by adopting a theorem-proving based approach that can be integrated with component-based declarative protocol development, *DRIVER* provides a unifying framework that bridges specification, verification, and implementation.

7 Conclusions

In this paper, we present the *DRIVER* system for designing, analyzing and implementing network protocols. *DRIVER* utilizes *theorem proving*, a well established verification technique where formal specifications are automatically generated to capture network semantics, and a user-driven proof process is used to establish network correctness properties. *DRIVER* takes as input declarative networking specifications written in the *NDlog* query language, and maps that automatically into formal specifications that can be directly used in existing theorem provers to verify desired protocol properties. Moreover, the verified formal specifications can be compiled into *NDlog* programs for execution, hence enabling the synthesis of implementation from verified protocol.

Our initial experiences suggest that *DRIVER* is a promising approach towards a unified framework that integrates specification, implementation, and verification. Moving forward, we have identified a few areas of future work.

First, we are working towards integrating the use of model checking techniques to specify general protocol invariants such as *reachability* and *solvability* [18] in temporal logic. Second, we are exploring more automatic proof support to make *DRIVER*

more approachable for non theorem proving expert. Most general-purpose theorem provers utilize an interactive proof process that requires experience of the proof system of these provers. To ease the user-directed proof construction, we plan to introduce into *DRIVER* network-specific proof strategies by leveraging the PVS built-in proof strategy language [2], hence lowering the barrier for adoption by network designers.

Finally, recent work on boolean satisfiability (SAT) solving and satisfiability modulo theories (SMT) [4], as well as the development in automatic first-order theorem provers have enable fully automated verification of various software and hardware problems. This provides an alternative proof automation support to PVS network-specific proof strategies developments that we plan to incorporate into *DRIVER*.

References

1. P2: Declarative Networking System. http://p2.cs.berkeley.edu.
2. PVS Specification and Verification System. http://pvs.csl.sri.com/.
3. The Coq Proof Assistant. http://coq.inria.fr.
4. Yices: An SMT Solver. http://yices.csl.sri.com/.
5. R. Cardell-Oliver. On the use of the hol system for protocol verification. In *TPHOLs*, 1991.
6. C. T. Ee, B.-G. Chun, V. Ramachandran, K. Lakshminarayanan, and S. Shenker. Resolving Inter-Domain Policy Disputes. In *SIGCOMM*, 2007.
7. D. Engler and M. Musuvathi. Model-checking large network protocol implementations. In *NSDI*, 2004.
8. A. P. Felty, D. J. Howe, and F. A. Stomp. Protocol verification in nuprl. In *CAV*, 1998.
9. T. G. Griffin, F. B. Shepherd, and G. Wilfong. The Stable Paths Problem and Interdomain Routing. *IEEE Transactions on Networking*, 10:232–243, 2002.
10. T. G. Griffin and J. L. Sobrinho. Metarouting. In *ACM SIGCOMM*, 2005.
11. C. Killian, J. Anderson, R. Jhala, and A. Vahdat. Life, death, and the critical transition: Finding liveness bugs in systems code. In *NSDI*, 2007.
12. C. Labovitz, G. Malan, and F. Jahanian. Internet Routing Instability. *ACM/IEEE Trans. on Networking*, 1998.
13. B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing Declarative Overlays. In *ACM SOSP*, 2005.
14. B. T. Loo, J. M. Hellerstein, I. Stoica, and R. Ramakrishnan. Declarative Routing: Extensible Routing with Declarative Queries. In *ACM SIGCOMM*, 2005.
15. J. A. N. Perez, A. Rybalchenko, and A. Singh. Cardinality abstraction for declarative networking applications. In *CAV*, 2009.
16. R. Ramakrishnan and J. D. Ullman. A Survey of Research on Deductive Database Systems. *Journal of Logic Programming*, 23(2):125–149, 1993.
17. L. Subramanian, M. Caesar, C. T. Ee, M. Handley, M. Mao, S. Shenker, and I. Stoica. HLP: A Next-generation Interdomain Routing Protocol. In *SIGCOMM*, 2005.
18. Timothy G. Griffin et. al. An Analysis of BGP Convergence Properties. In *SIGCOMM*, 1999.
19. A. Wang, P. Basu, B. T. Loo, and O. Sokolsky. Declarative network verification. In *11th International Symposium on Practical Aspects of Declarative Languages*, 2009.