Declarative Network Verification

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

[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,boonloo,sokolsky}@seas.upenn.edu pbasu@bbn.com

Abstract. In this paper, we present our initial design and implementation of a *declarative network verifier* (*DNV*). *DNV* utilizes *theorem proving*, a well established verification technique where logic-based *axioms* that automatically capture network semantics are generated, and a user-driven proof process is used to establish network correctness properties. *DNV* takes as input declarative networking specifications written in the Network Datalog (*NDlog*) query language, and maps that automatically into logical *axioms* that can be directly used in existing theorem provers to validate protocol correctness. *DNV* is a significant improvement compared to existing use case of theorem proving which typically require several man-months to construct the system specifications. Moreover, *NDlog*, a high-level specification, whose semantics are precisely compiled into *DNV* without loss, can be directly executed as implementations, hence bridging specifications, verification, and implementation. To validate the use of *DNV*, we present case studies using *DNV* in conjunction with the PVS theorem prover to verify routing protocols, including eventual properties of protocols in dynamic settings.

Keywords: declarative networking, network protocol verification, domain-specific languages, theorem proving

1 Introduction

In recent years, we have witnessed a proliferation of new overlay networks [24] that use the existing Internet to enable deployable network evolution and introduce new services. Concurrently, as sophisticated, bandwidth-intensive, and even mission-critical services are being deployed over heterogeneous network infrastructure, there is increased demand for new network routing protocols that can flexibly adapt to a wide range of variability in network connectivity and data traffic patterns. This has cummulated into recent efforts at clean-slate efforts aimed at redesigning the Internet.

Given the proliferation of new architectures and protocols, there is a growing consensus on the need for formal software tools and programming frameworks that can facilitate the design, implementation, and verification of new protocols.

This has lead to several recent proposals broadly classified as: (1) *algebraic and logic frameworks* [11, 9] that enable protocol correctness in the design phase; (2) *testing platforms* [16, 27] that provide mechanisms for runtime verification and distributed replay, and (3) *programming toolkits* [8, 14] that enable network protocols to be specified, implemented, and model-checked.

In this paper, we present our initial design and implementation of a *declarative network verifier* (*DNV*). Our work is a significant step towards *bridging* network specifications, protocol verification, and implementation within a common language and system. *DNV* achieves this unified capability via the use of *declarative networking* [20, 19, 18], a declarative domain-specific approach for specifying and implementing network protocols, and *theorem proving*, a well established verification technique based on logical reasoning.

In declarative networking, network protocols are specified using a declarative logic-based query language called *Network Datalog* (*NDlog*). In prior work, it has been shown that traditional routing protocols can be specified in a few lines of declarative code [20], and complex protocols such as Chord DHT [31] in orders of magnitude less code [19] 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].

Recent significant advances in model checking of network protocol implementations include *MaceMC* [13] and *CMC* [7]. Compared to these proposals, *DNV* has the advantage that it achieves complete verification for networks of arbitrary size, a long-standing challenge in any practical network verification system. Incomplete verification is a common limitation in *MaceMC* and *CMC* due to the the state-explosion problem, particularly when used to verify large networks with complex protocol behavior. In addition, since *DNV* directly verifies declarative networking specifications, an explicit model extraction step via execution exploration is not required.

This paper makes the following two contributions:

- First, we propose *DNV*, a declarative network verifier that leverages declarative networking's connection to logic programming to automatically compile high-level *NDlog* program into formal specifications as *axioms* without semantics loss, which can be further used in a theorem prover to validate protocols. A semi-automated proof guided by the user is then carried out and mechanically checked in a general-purpose theorem prover to establish the protocol correctness properties. High-level *NDlog* programs that have been verified in *DNV* can be directly executed as implementations, hence bridging specifications and implementations within a unified declarative framework.
- Second, we demonstrate that *DNV* enables the verification of network protocols in *dynamic settings*, where protocols continuously update network state based on incoming network events. *DNV* achieves this via its use of declarative networking which incorporates the notion of periodic *soft-state* [26] maintenance of network state into its query language and semantics. Soft state is

central and critical in networking implementations because in a very simple manner it provides eventually correct semantics in the face of reordered messages, node disconnection, and other unpredictable occurrences.

DNV aims to provide a practical solution towards network protocol verification, one that achieves a unifying framework that combines specifications, verification, and implementation. Our work is a significant improvement over existing usage of theorem proving [12, 10] which typically require several man-months to develop the system specifications, a step that *DNV* reduces to a few hours through the use of declarative networking. To our best knowledge, *DNV* is also one of the first attempts at using theorem proving to verify eventual semantics of protocols in dynamic settings.

2 Background: Declarative Networking

In this section, we will provide a brief overview of declarative networking. Interested readers are referred to references [20, 19, 18, 17] for more details.

2.1 Datalog Language

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. *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 [25]. 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 variable names begin with an upper letter, while constants names begin with an lowercase letter. An optional `Query` rule specifies the output of interest (i.e. result tuples).

2.2 Path-vector Protocol

We present an example *NDlog* program that implements the *path-vector* protocol [23], a standard textbook route protocol used for computing paths between any two nodes in the network.

```
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(Z,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).
Query bestPath(@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(Z,P2)` with cost `C1+C2`. Note that `p1-p2` also utilizes 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(Z,P2)` prepends `Z` 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 `bestPathHop(@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.

The execution model of declarative networks is based on a distributed variant of the standard evaluation technique for Datalog programs that is commonly known as *semi-naïve* (SN) evaluation [18], with modifications to enable pipelined asynchronous evaluation suited to a distributed setting. Reference [18] provides details on the implementation and execution model of declarative networking.

For the purposes of formal verification, we do not consider the location specifiers within the proof. This does not affect the program in terms of the set of eventual facts being generated but does affect the notion of data distribution. We will revisit this later in Section 7.

3 Overview of DNV

Figure 1 provides an overview of *DNV*'s basic approach towards unifying specifications, verification, and implementation within a common declarative framework. *DNV* takes as input *NDlog* program specifications of the declarative protocol (see Section 2 for an example). Since most theorem provers leverage type information, *DNV* further includes a *Type Schema* with the *NDlog* program specifications. This is not unlike a database-like schema storing the attribute types of all network state being used.

In order to carry out the formal verification process, the *NDlog* programs and schema information are automatically compiled into formal specifications

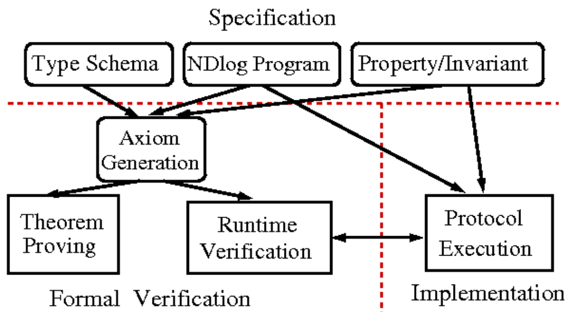

Fig. 1. *DNV overview block diagram. Arrows denote flow of information.*

recognizable by a standard theorem prover (e.g. PVS [21], 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 into the theorem prover, or expressed as *NDlog* rules which are then 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 by both theorem prover and 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 *NDlog* specifications. Our initial *DNV* prototype uses the PVS theorem prover, due to its substantial support for proof strategies which significantly reduce the time required in the actual proof process. However, the techniques describe in this paper are agnostic to other theorem provers. We have also validated some of the verification presented in this paper using the Coq [3] prover.

To illustrate the verification process, we step through the path-vector protocol example, shown in Section 2. For ease of exposition, we defer the treatment of soft-state derivations and events to Section 4, focusing instead on traditional *hard-state* data (with infinite lifetimes) that are valid until explicitly deleted.

3.1 From *NDlog* rules to PVS Axioms

The first step in *DNV* involves the *automatic* generation of PVS *formalization* (or axioms) directly from *NDlog* rules. Based on the proof-theoretic semantics of Datalog [30], there is a natural and automatic mapping from *NDlog* rules to PVS axioms.[3] Before showing the actual PVS encoding for the path-vector protocol, it is informative to understand the proof-theoretic semantics of `p1` and `p2` as inference rules used in proof system:

The inference rule `p1` expresses the logical statement

$$\forall(S, D, P, C).link(S, D, C) \wedge (P = f_{init}(S, D)) \implies path(S, D, P, C)$$

Rule `p2` is slightly more complex as some attribute variables do not appear in the resulting head. The general technique to express these variables is in terms of existential quantification. Accordingly, rule `p2` expresses the logical statement that

$$\begin{aligned}&\forall(S, D, P, C).\exists(C_1, C_2, Z, P_2).\\&\quad(link(S, Z, C_1) \wedge bestPath(Z, D, P_2, C_2) \wedge C = C_1 + C_2 \wedge\\&\quad P = f_{concatPath}(Z, P_2)) \implies path(S, D, P, C)\end{aligned}$$

From the above logical statements, *DNV* generates the following axioms:

```
path_generate: AXIOM
 FORALL (S,D,Z:Node)(C:Metric)(P:Path):
  (link(S,D,C) AND P=f_init(S,D)) OR
  ((EXISTS (P2:Path)(C1,C2:Metric): (link(S,Z,C1) AND bestPath(Z,D,P2,C2)
    AND C=C1+C2 AND P=f_concatPath(Z,P2))) =>path(S,D,P,C)

path_close: AXIOM
 FORALL (S,D,Z:Node)(C:Metric)(P):path(S,D,P,C)
  =>((link(S,D,C) AND P=f_init(S,D)) OR
    (EXISTS (Z:Node)(P,P2:Path)(C1,C2:Metric): (link(S,Z,C1) AND
        bestPath(Z,D,P2,C2) AND C=C1+C2 AND P=f_concatPath(Z,P2))))
```

The first `path_generate` axiom is generated in a straightforward manner from rules `p1` and `p2`, where the logical `OR` indicates that `path` facts can be generated from either rule. The `path_close` axiom indicates that the `path` tuple is the smallest set derived by the two rules, ensuring that these axioms automatically generated in *DNV* correctly reflected the minimal model of *NDlog* semantics. The list manipulation functions `f_concatPath` and `f_init` are predefined from PVS primitive types. We omit this discussion due to space constraints.

PVS provides *inductive definitions* that allows the two axioms above to be written in a more concise and logically equivalent form:

```
path(S,D,(P: Path),C): INDUCTIVE bool =
  (link(S,D,C) AND P=f_init(S,D) AND Z=D) OR
  (EXISTS (C1,C2:Metric) (Z2:Node) (P2:Path):
        link(S,Z,C1) AND path(Z,D,P2,C2) AND C=C1+C2
        AND P=f_concatPath(S,P2) AND f_inPath(S,P2))
```

The universal quantifiers over the attributes to `path` (i.e. `S,D,Z...`) are implicitly embedding and existential quantifiers such as `C1` and `C2` are explicitly

[3] The equivalence of *NDlog*'s proof-theoretic semantics and operational semantics guarantees that *DNV* is sound in the sense that, the correctness property established by *DNV* corresponds precisely to the operational semantics of *NDlog* execution.

stated. *DNV* axiom generator always produces this inductive definition, and employs the axiom form only in the presence of mutual dependencies among the head predicates which makes PVS inductive definition impossible.

Accordingly, *NDlog* rules `p3-p4` are automatically complied into PVS formalization in a similar way:

```
bestPathCost(S,D,min_C): INDUCTIVE bool =
  (EXISTS (P:Path): path(S,D,P,min_C)) AND
  (FORALL (C2:Metric): (EXISTS (P2:Path): path(S,D,P2,C2)) => min_C<=C2)

bestPath(S,D,P,C):INDUCTIVE bool =
  bestPathCost(S,D,C) AND path(S,D,P,C)
```

In addition to the above PVS encoding for *NDlog* rules, type definitions are produced automatically from the database schema information. For instance, given a database schema definition for

```
link(src:string, dst:string, metric:integer)
```

the corresponding PVS type declaration is

```
link:[Node,Node,Metric -> bool]
```

where `Node` is declared as a string type and `Metric` as an integer type.

3.2 Proving Route Optimality

The next step is to establish the actual properties in PVS. Properties are represented by PVS *theorems* and serve as the *starting points* (or *proof goals*) in the proof construction process. In this section we illustrate in high level this proof process by walking through the *route optimality* property in the path-vector protocol captured by the following PVS theorem `bestPathStrong`:

```
bestPathStrong: THEOREM
 FORALL (S,D:Node) (C:Metric) (P:Path):
   bestPath(S,D,P,C) =>
     NOT (EXISTS (C2:Metric)(Z:Node)(P2:Path): path(S,D,Z,P2,C2) AND C2<C)
```

`bestPathStrong` specifies the following logic formula

$$\forall(S,D,C,P).bestPath(S,D,P,C) \implies \neg(\exists C_2, Z, P_2.path(S,D,Z,P_2,C_2) \wedge (C_2 < C))$$

that a given tuple `bestPath(S,D,P,C)` implies that there does not exist another path from `S` to `D` with better metrics. This formula therefore represents the fact that the derived tuple `bestPath(S,D,P,C)` must correctly denote the optimal path `P` from `S` to `D`, by excluding the existence of a different better path from `S` to `D` with lower cost.

Given this above theorem, *DNV* utilizes PVS to carry out the interactive proof process.

3.3 Interactive Proof Process

In any general purpose theorem prover like PVS, one performs the proof in a *goal-directed* fashion interactively by supplying proof commands. In this case, starting from the `bestPathStrong` goal, and then recursively reducing it to subgoals until all subgoals are trivially true.

To automate proof development, PVS has approximately 100 built-in proof strategies, of which 20 are usually sufficient to automate a majority of the proof effort. While using a theorem prover is an acquired skill, there are a few general "rules of thumb" that one can utilize as part of the proof process. We will describe in detail these rules in Appendix A.

In the remaining of this section, we provide an intuition on the proof process, by presenting the strawman proof process that does not utilize any user-defined proof strategies specific to declarative network beyond PVS's built-in proof commands:

```
("" (skosimp*) (expand bestPath) (prop) (expand bestPathCost)
    (prop) (skosimp*) (inst -2 C2!1) (grind))
```

This proof script reflects the interactive PVS proof process directed by the user, where PVS takes care of all low level proof details and allows the user to concentrate on high-level proof strategies. Without going into details of each PVS command, we provide a high-level intuition of each step. The first command `skosimp*` performs repeated skolemization that removes universal quantifiers `S,D,C` and `P` in the theorem. Skolemization is generally the first proof step to try in proving any universal quantified theorems. The subsequent two `expand` commands are used to unfold the inductive definition we defined in 3.1, each followed by `prop` that performs proportional simplification. Then `skosimp*` is employed to remove universal quantifiers and `inst` to instantiate the existential quantified variable with proper instance (`C2!1`). The rest of the proof is complete by using PVS's `grind` command which performs skolemization, heuristic instantiation, propositional simplification and decision procedures for linear arithmetic and equality.

Once the above proof script is supplied, PVS requires only fraction of a second to carry out and automatically check the actual proof. Note that the proof covers *all* instances of the network. This is in contrast to model checking, which explores only specific network instances. In addition to proving the route optimality property of the declarative path-vector protocol, we have proven properties such as the potential cycles in the protocol if the cycle check (enforced using the `f_inPath` function) is removed.

Given that our target users are network designers, the proof process should ideally be automated. Beyond the general proof methodology presented in Appendix A that provides a starting point for users, PVS provides mechanisms to express high-level proof strategies via its proof strategy language that enables domain-specific knowledge to be exploited and installed in the proof. In Section 7, as our future work, we will discuss the potential of using domain-specific PVS strategies tailored to declarative networking to support proof automation, as well as survey recent work on automated provers that are potentially useful for further automating the search process.

4 Soft-state, Events and Dynamics

Up to this point, we have limited our proofs to a subset of the complete *NDlog* language by omitting the treatment of *soft-state tuples (i.e. predicates)*. This simplification enable us to generate axioms recognizable by a theorem prover directly from *NDlog* programs without having to worry about the semantics of time and data expiration. In practice, soft-state data and events are central in network protocols, and adopted in many declarative network implementations. In the rest of this section, we will introduce the soft-state model in declarative networking, describe how rules with soft-state predicates (referred as *soft-state rules*) can be verified in a similar fashion as shown in Section 3, by first rewriting soft-state rules into logically equivalent rules with only hard-state predicates (i.e.*hard-state rules*).

4.1 Soft-state Model in *DNV*

Declarative networking incorporates support for *soft-state* [26] derivations commonly used in networks. In the soft state storage model, all data (input and derivations) has an explicit "time to live" (TTL) or lifetime, and all tuples must be explicitly reinserted with their latest values and a new TTL or they are deleted.

To support soft-state, an additional language feature is added to the *NDlog* language, in the form of a `materialize` [19] declaration at the beginning of each *NDlog* program that specifies the TTL of predicates. For example, the expression `materialized(link,10,keys(1,2))` specifies that the `link` tuple is stored at a table with primary key set to the first and second attributes (denoted by `keys(1,2)` and that each `link` tuple has a lifetime of 10 seconds[4]. If the TTL is set to infinity, the predicate will be treated as *hard-state*.

The soft-state storage semantics are as follows. When a tuple is derived, if there exists another tuple with the same primary key but differs on other attributes, an *update* occurs, in which the new tuple replaces the previous one. On the other hand, if the two tuples are identical, a *refresh* occurs, in which the existing tuple is extended by its TTL.

For a given predicate, in the absence of any `materialize` declaration, it is treated as an *event* predicate with lifetime set to zero. Since events are not stored, they are primarily used to trigger other rules or in response to network events. Reference [17] provides more details on how soft-state storage model and events are implemented within a declarative networking engine.

4.2 Soft-state to Hard-state Rewrite

The rule rewrite consists of two steps. First, all soft-state predicates of the form `p(...)` where "`...`" refer to predicate arguments, are translated into an equivalent hard-state predicate of the form `p(...,Tc,Tl)`, where the additional

[4] Following the conventions of the *P2* declarative networking system, attribute 0 is reserved for the predicate name.

attributes `Tc` and `Tl` denote the creation time and lifetime of each tuple `p` respectively. This initial rewrite step makes explicit the creation time and lifetime by adopting `Tc`, `Tl` in each soft-state predicate. Event predicates are rewritten in a similar fashion. However, `Tl` is omitted since events have zero lifetime by definition.

After the first step, additional constraints reflecting soft-state semantics are added to ensure that all soft-state facts only process with other facts valid within the same window period of time, as expressed in terms of constraints over `Tc` and `Tl`. Consider soft-state rules of the form, $e :- e_1, s_1, s_2, ..., s_n$. This rule triggered by input event e_1 with creation time Tc_{e1}, takes as input both the triggering event and several soft-state predicates $s_1, s_2, ..., s_n$, and generates a event. The rewritten equivalent hard-state rules is of the form:

$$
\begin{aligned}
e(..., Tc_{e1}) :- \ & e_1(..., Tc_{e1}), s_1(..., Tc_{s1}, Tl_{s1}),\\
& s_2(..., Tc_{s2}, Tl_{s2}), ...,\\
& s_n(..., Tc_{sn}, Tl_{sn}),\\
& Tc_{s1} < Tc_{e1} \leq Tc_{s1} + Tl_{s1}, ...,\\
& Tc_{sn} < Tc_{e1} \leq Tc_{sn} + Tl_{sn}
\end{aligned}
$$

Since the event e_1 directly triggers the derivation of s(or e), the creation time of the derived event e is set to be the same as that of the input e_1 (i.e. Tc_{e1}). An additional n constraints $Tc_{si} < Tc_{e1} \leq Tc_{si} + Tl_{si}$ are added to ensure that only soft-states s_i with valid time interval $[Tc_{si}, Tc_{si} + Tl_{si}]$ that always overlaps with Tc_{e1} are used to generate e.

Another possible class of soft-state rules are of the form, $e :- s_1, s_2, ..., s_n$, where a event is generated by sets of soft-states. The main difference compared to the previous soft-state rule is the lack of a triggering event. The rewritten hard-state rule is of the form:

$$
\begin{aligned}
e(..., Tc) :- \ & s_1(..., Tc_{s1}, Tl_{s1}), s_2(..., Tc_{s2}, Tl_{s2}),\\
& ..., s_n(..., Tc_{sn}, Tl_{sn}),\\
& Tc = max < Tc_{s1}, Tc_{s2}, ..., Tc_{sn} >,\\
& Tc_{s1} < Tc \leq Tc_{s1} + tl_{s1}, ...,\\
& Tc_{sn} < Tc \leq Tc_{sn} + Tl_{sn}
\end{aligned}
$$

Note that Tc is set to the max of all possible creation times of the input soft-state predicates (since the derived fact is true only when all the input facts are present).

The same rewrite process applies to rules with rule-head that is a soft-state predicate., and an additional `Tl` attribute set to the declared lifetime in corresponding table (indicated in the `materialize` statement) is added.

4.3 Neighbor-Maintenance

We provide a concrete example of a simple soft-state program that implements periodic neighbor maintenance, where each node periodically sends a keep-alive message `pingMsg` to its neighbors, and then remove the neighbor when a response `pongMsg` is not received after a period of time.

```
materialize(link,10,keys(1,2)).
pp1 pingMsg(S,@D) :- periodic(@S,5), link(@S,D,C).
pp2 pongMsg(@S,D) :- pingMsg(S,@D),link(@D,S,C).
pp3 link(@S,D,C)  :- pongMsg(@S,D), C=1.
```

The program utilizes soft-state predicate `link` with a TTL of 10 seconds. In the absence of any `materialize` declarations, predicates such as `pingMsg` and `pongMsg` are treated as events with zero lifetimes. Rule `pp1` is triggered by a special built-in `periodic` predicate, which denotes an infinite stream of periodic event tuples generated at node `S` every `5` seconds. This allows rule `pp1` to generate a `pingMsg` every five seconds to all neighbors `D`. Each neighbor `D` that receives the `pingMsg` replies with a `pongMsg` (rule pp2), which then results in refreshing `link(@S,D,C)`. Note that the cost of each link is trivially set to `1` in the earlier program, although one can also introduce other link metrics, such as round-trip-time or bandwidth, computed via additional *NDlog* rules. Given that the lifetime of each `link(@S,D,C)` tuple is set to 10 seconds, in the absence of a `pingMsg` reply from neighbor `D`, the corresponding `link(@S,D,C)` entry will be deleted (upon expiration).

The resulting hard-state program after applying the rewrite in Section 4.2:

```
hpp1 pingMsg(S,@D,Tc):- periodic_ping(@S,5,Tc), link(@S,D,C,Tc',10),
                        Tc'<Tc<Tc'+10.
hpp2 pongMsg(@S,D,Tc):- pingMsg(S,@D,Tc), link(@D,S,C,Tc',10),
                        Tc'<Tc<Tc'+10.
hpp3 link(@S,D,C,Tc,10):- pongMsg(@S,D,Tc),C=1.
hpp4 periodic_ping(@S,5,Tc'):- periodic_ping(@S,5,Tc), Tc'=Tc+5.
```

All predicates in the above program are materialized as hard-state (TTL=∞), as we omit the `materialize` statements for brevity. Since the lifetime of `link` is specified as 10 seconds, the last attribute `Tl` of `link` after the rewrite is set to 10. An additional rule `hpp4` shown above is added to explicitly specify the use of built-in `periodic` predicate with interval of 5 seconds. Our rewrite emulates the behavior of the original *NDlog* `periodic` predicate by generating a `periodic_ping` event every 5 seconds.

5 Distance-vector in a Dynamic Network

In this section, we illustrate the capability of *DNV* in reasoning about eventual semantics of protocols in dynamic networks. We base our illustration on the verification of the *distance-vector* protocol, commonly used for computing shortest routes in a network. Due to space constraints, we are not able to show exhaustively all the PVS specifications and proofs. The interested reader is referred to reference [6] for the complete PVS axioms, theorems, and proofs. In particular, the proofs are available for replay within PVS, given the axioms and theorems as input.

5.1 Distance Vector Specification in *NDlog*

The following soft-state *NDlog* program implements the distance-vector protocol, computing best paths with least cost:

```
materialize(hop,10,keys(1,2,3)).
materialize(bestHop,10,keys(1,2)).
materialize(bestHopCost,10,keys(1,2)).
#include(pp1,pp2,pp3)
dv1 hop(@S,D,D,C) :- link(@S,D,C).
dv2 hop(@S,D,Z,C) :- hopMsg(@S,D,Z,C).
dv3 bestHopCost(@S,D,min<C>) :- hop(@S,D,Z,C).
dv4 bestHop(@S,D,Z,C) :- bestHopCost(@S,D,C), hop(@S,D,Z,C).
dv5 hopMsg(@N,D,S,C1+C2):- periodic(@S,5), bestHop(@S,D,Z,C1),
                           link(@S,N,C2).
Query bestHop(@S,D,Z,C)
```

The `#include` declaration in first line introduces into distance-vector protocol the rules `pp1-pp3` from the *Ping-Pong* program in the previous section for maintaining neighbor information as `links`. The program derives soft-state predicates `hop`, `bestHop`, and `bestHopCost` with TTL of 10 seconds, and an event predicate `hopMsg`, and takes as input `link` tuples generated from rules `pp1-pp3`.

First, rules `dv1-dv2` derive `hop(@S,D,Z,C)` tuples, where `Z` denotes the next hop (instead of the entire path) along the path from `S` to `D`. Second, the protocol is driven by the periodic generation of `hopMsg(@S,D,Z,C)` in rule `dv5`, where each node `S` advertises its knowledge of all possible best hops table (`bestHop`) to all its neighbors. Note that bidirectional connectivity and cost is assumed. Each node receives the advertisements as `hopMsg` events (rule `dv2`) which it then stores locally in its `hop` table. Finally, Rules `dv3-dv4` compute the best hop for each source/destination pair in a similar fashion as the earlier path-vector protocol.

Unlike the path-vector protocol presented in Section 2.2, the distance-vector protocol computes only the *next hop* along the best path, and hence does not store the entire path between any two nodes.

5.2 Axiom Generation for Soft-State

The following *NDlog* rules `dv1-dv6` shows the equivalent hard-state rules after applying the soft-state rewrite process described in Section 4.2.

```
#include(hpp1,hpp2,hpp3,hpp4)
dv1 hop(@S,D,D,C,Tc,10):- link(@S,D,C,Tc,10).
dv2 hop(@S,D,Z,C,Tc,10):- hopMsg(@Z,D,W,C2,Tc2), Tc=Tc2+5, C=C2+1.
dv3 bestHopCost(@S,D,min<C>,Tc,10):- hop(@S,D,D,C,Tc,10).
dv4 bestHop(@S,D,Z,C,Tc,10):- bestHopCost(@S,D,C,Tc,10),
                              hop(@S,D,Z,C,Tc1,10), Tc1<Tc<=Tc1+10.
dv5 hopMsg(@N,D,Z,C,Tc):- periodic_dv(@S,5,Tc), bestHop(@S,D,Z,C,Tc1,10),
                          link(@S,N,C,Tc2,10),Tc2<Tc<=Tc2+10,
                          Tc1<Tc<=Tc1+10.
dv6 periodic_dv(@S,5,Tc):- periodic_dv(@S,5,Tc2), Tc=Tc2+5
Query bestHop(@S,D,Z,C,Tc,Tl)
```

Rules `dv1-dv5` are the corresponding hard-state rewrites, and `dv6` emulates the behavior of `periodic` streams employed in `dv5`, as described in Section 4.2. We introduce an extra constraint `Tc=Tc2+5` in rule `dv2`. This condition is required so that causality of rule execution is preserved within one interval: resulting `hopMsg` events generated within one periodic interval derives `hop` facts used in the next period internal and not vice versa. We note that this addition constraint is automatically added: required only in cases when rules depend on each other in a cyclical fashion (e.g. `hop` derived in `dv1-dv2`, `hopMsg` in `dv5`, and `bestHop` in `dv4`), a dependency that can be detected via static check.

Based on this rewritten program, the automatically generated PVS axioms are as follows:

```
hopMsg(S,D,Z,C,Tc): INDUCTIVE bool =
 (EXISTS (Tc2,T3:Time):
   bestHop(S,D,Z,C,Tc2,10) AND periodic(S,5,Tc) AND
        link(S,D,Tc3,10) AND Tc2<Tc<=Tc2+10 AND Tc3<Tc<=Tc3+10 AND C=1)

hop(S,D,Z,C,Tc,Tl): INDUCTIVE bool =
  (link(S,D,Tc,10) AND Z=D AND Tl=10 AND C=1) OR
  (EXISTS (C2:Metric):
    hopMsg(S,D,Z,C2,Tc2) AND C=C2+1 AND Tl=10 AND Tc=Tc2+5)

bestHopCost(S,D,MIN_C,Tc,Tl): INDUCTIVE bool =
  EXISTS (Z:Node):
   hop(S,D,Z,MIN_C,Tc) AND Tl=10 AND
    (FORALL (C:Metric): (EXISTS (Z:Node): hop(S,D,Z,C,Tc,10))=>MIN_C<=C)

bestHop_refresh: AXIOM
  FORALL (S,D,Z:Node) (C:Metric) (Tc:Time):
    (bestHopCost(S,D,C,Tc,10) AND hop(S,D,Z,C,Tc,10))
        =>bestHop(S,D,Z,C,Tc,10)

bestHop_close: AXIOM
  FORALL (S,D,Z:Node) (C:Metric) (Tc:Time):
    bestHop(S,D,Z,C,Tc,10) =>
      (bestHopCost(S,D,C,Tc,10) AND hop(S,D,Z,C,Tc,10))

periodic_dv(S,I,Tc): INDUCTIVE bool =
  EXISTS (Tc2:Time): periodic_dv(S,I,Tc2) AND Tc=Tc2+5 AND I=5
```

Recall automatic axiom generation process in Section 3.1, PVS axioms would be explicitly used in face of mutual dependencies between rules (that derive `bestHop`, `hop`, and `hopMsg`). To break the dependency, we therefore specify `dv4` with two axioms `bestHop_refresh` and `bestHop_close` .

5.3 Convergence in Stable Network

The lack of knowledge of the entire path in the distance-vector protocol comes at the expense of potential update loops in the presence of link updates. This is a well-known limitation of the distance-vector protocol, commonly known as the *count-to-infinity* problem.

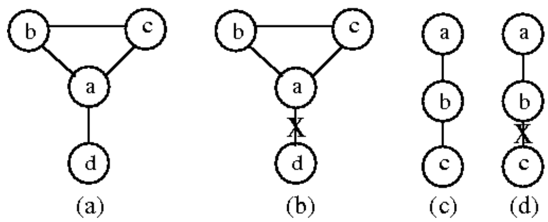

Fig. 2. *Network Dynamics*

Our verification is performed on a 4-node network instance as shown in Figure 2(a). Note that this instance represents a loop consisting of three nodes (`a`, `b`, and `c`) that can reach the rest part of the network via a fourth node `d`, and the results of this verification apply to *arbitrary* network that contains such a loop. For ease of exposition and avoid redoing the proof established for the *Ping-Pong* program in Section 4 on neighbor maintenance, we supply this network instance using the following PVS inductive definition, where each clause connected by logical operator `OR` represents a link between two nodes:

```
link(S,D,C,Tc,Tl): INDUCTIVE bool =
   (S=a AND D=b AND C=1 AND Tl=10 AND
     (EXISTS (i:posnat): Tc=5*i))
OR (S=b AND D=c AND C=1 AND Tl=10 AND
     (EXISTS (i:posnat): Tc=5*i))
OR ...
   (S=a AND D=d AND C=1 AND Tl=10 AND
     (EXISTS (i:posnat): Tc=5*i))
```

Network convergence is expressed using the following theorem:

```
bestHopCost_converge: THEOREM
  EXISTS (j:posnat): FORALL (S,D:Node)
    (C:Metric)(i:posnat): (i>j)
       => bestHopCost(S,D,C,5*i,10) = bestHopCost(S,D,C,5*j,10)
```

Given an input network, the distance-vector protocol requires a number of rounds of communication among all nodes, for route advertisements (in the form of `hopMsg`) to be propagated in the network. In the above theorem, the existential quantified variable `j` denotes the initial number of periodic intervals (set to be at least the network diameter) required to propagate all route advertisements. The theorem states that for any arbitrary time `i` after `j`, the value of `bestHopCost` always converges (i.e. no longer changes).

5.4 Count-to-Infinity Analysis

Next, we demonstrate the capability of *DNV* to prove the presence of the *count-to-infinity* problem in the distance-vector protocol. This is a well-studied limitation where the protocol potentially *diverges* (i.e. not reach steady state) in the presence of link failures.

Before showing the actual proofs, we provide a textbook example [23] that clearly demonstrates the problem intuitively. Revisiting the network in Figure 2(b), when `link(a,d)` fails, node `a` would advertises this information to its immediate neighbors `b` and `c`. However, despite the fact that `d` is no longer reachable from either `a` `b` or `c`, based on information that `c` can reach `d` in two hops, `b` would conclude that it can reach `d` in three hops. Node `c` makes a similar conclusion. In the next round of updates, node `a` learns that `b` and `c` can reach `d` in three hops, and updates its distance to `d` as four accordingly. This cycle continues indefinitely, resulting in the count-to-infinity problem.

Proving Divergence The proof requires a network scenario that results in a count-to-infinity problem. Using the example described above, we supply this network dynamics using the following PVS inductive definition:

```
link (S,D,C,Tc): INDUCTIVE bool =
   (S=a AND D=b AND C=1 AND
     (EXISTS (i:posnat): Tc=5*i) AND Tc<100)
OR (S=b AND D=a AND C=1 AND
     (EXISTS (i:posnat): Tc=5*i) AND Tc<100)
OR ...
   (S=a AND D=d AND C=1 AND
     (EXISTS (i:posnat): Tc=5*i) AND Tc<100)
OR (S=d AND D=a AND C=1 AND
     (EXISTS (i:posnat): Tc=5*i) AND Tc<100)
OR (S=a AND D=b AND C=1 AND
     (EXISTS (i:posnat): Tc=5*i) AND Tc>=100)
OR (S=b AND D=a AND C=1 AND
     (EXISTS (i:posnat): Tc=5*i) AND Tc>=100)
OR ...
   (S=c AND D=b AND C=1 AND
     (EXISTS (i:posnat): Tc=5*i) AND Tc>=100)
OR (S=b AND D=c AND C=1 AND
     (EXISTS (i:posnat): Tc=5*i) AND Tc>=100)
```

The definition indicates that the `link(a,d)` and `link(d,a)` facts are only present before time `100`, denoting that a link failure between nodes `a` and `d` happens at time `100`. The count-to-infinity theorem is expressed as follows:

```
bestHop_increase_to_infinity: THEOREM
  FORALL (a,b,d:Node)(t:Time)(c:Metric):
    (t>100 AND bestHop(a,d,b,c,t,10))
      => (EXISTS (t':Time)(c':Metric):
           (t'>t AND c'>c AND bestHop(a,d,b,c',t',10)))
```

The theorem above states that if the distance vector protocol diverges, the best hop from `a` to `d` will increase indefinitely over time, a symptom of the count-to-infinity problem. In reference [6], we have the complete proof of this theorem, as well as addition theorems that further verify the presence of the count-to-infinity problem.

Interestingly, we have been able to prove in PVS a set of *stronger* theorems specific to this three-node network cycle, stating that the cost of the `bestPath` between both b and d, as well as those between a and d would increase following a precise pattern, as expressed in the following PVS theorems:

```
bestHop_bd_increase_to_infinity: THEOREM
  FORALL (t: Time) (c: Metrics):
    (EXISTS (i:posnat): t=i*5 and t>100)
      => (bestHop(b,d,a,c,t,10) => bestHop(b,d,a,c+2,t+10,10))

bestHop_ad_increase_to_infinity: THEOREM
  FORALL (t:Time) (c:Cost):
    (EXISTS (i:posnat): t=i*5 and t>100)
      => (bestHop (a,d,b,c,t,10) => bestHop(a,d,b,c+2,t+10,10))
```

These theorems predict precisely that the value of cost argument increases by 2 at every two update intervals of 10 seconds. Since node b and c are symmetric in the loop, the same set of theorems apply to c.

Split Horizon Solution *Split-horizon* is a well-known solution to the count-to-infinity problem. Interestingly, the changes to the *NDlog* specification to implement the split-horizon solution is minimal, requiring only adding one additional selection predicate to rule dv5, hence further demonstrating the power of declarative programming. The modified rule dv5 is appended by the additional predicate N!=Z as:

```
dv5 hopMsg(@N,D,Z,C,Tc):- periodic(@S,5,Tc), bestHop(@S,D,Z,C,Tc1,10),
                          link(@S,N,C,Tc2,10), Tc2<Tc<=Tc2+10,
                          Tc1<Tc<=Tc1+10, N!=Z.
```

Rule dv5 expresses that if node Z learns about the path to D from node N , then node Z does not report this path back to to N.

One of the well-known limitations of the split-horizon fix is that it is limited to only solving the count-to-infinity problem in the cases where there are two-node cycles. Using *DNV*, we have successfully verified this limitation in the case where there are three-node cycles [6]. In the rest of this section, we outline our proofs that demonstrate that this fix indeed works for a two-node cycle.

Consider the network with two-node loop of a,b as depicted in Figure 2(c), the invariant stating that split horizon prevents counting-to-infinity in face of link failure works for two-node-loop is expressed as theorems:

```
bestHop_ac: Theorem
  FORALL (t: Time):
    t>= 110 AND (EXISTS (i: posnat): t=5*i)
      => NOT (EXISTS (z: Node) (C: Cost): bestHop (a,c,z,C,t,10))

bestHop_bc: Theorem
  FORALL (t: Time):
    t>= 110 AND (EXISTS (i: posnat): t=5*i)
      => NOT (EXISTS (z: Node) (C: Cost): bestHop (b,c,z,C,t,10))
```

These theorems states that the link failure in Figure 2(d) would be correctly reflected in the computation of bestHop: as long as node c is no longer reachable due to the link failure, bestHop between c and rest of the loop (i.e. node a, b) can no longer be derived. Our results generalizes to any network involving this two-node topology, and therefore we actually showed that in general, by applying split-horizon, distance vector protocol can prevent count-to-infinity problem we have seen in Section 5.4.

Furthermore we prove that split horizon works only in networks containing loops of two nodes, and counting-to-infinity cannot be avoided in loops involving three nodes. The fact split horizon fails to prevent counting-to-infinity when link fails, in a three-node-loop in Figure 2(a) is captured by the following set of PVS theorems:

```
bestHop_ae_increase_to_infinity: THEOREM
  FORALL (t:Time) (c:Cost):
   (EXISTS (i:posnat): t=i*5 and t>100)
     => (bestHop (a,e,b,c,t,10) => bestHop(a,e,b,c+3,t+15,10))

bestHop_bea_increase_to_infinity: THEOREM
  FORALL (t: Time) (c: Cost):
    (EXISTS (i:posnat): t=i*5 and t>100)
      => (bestHop(b,e,a,c,t,10) => bestHop(b,e,a,c+3,t+15))

bestHop_bec_increase_to_infinity: THEOREM
  FORALL (t: Time) (c: Cost):
    (EXISTS (i:posnat): t=i*5 and t>100)
      => (bestHop(b,e,c,c,t,10) => bestHop(b,e,c,c+3,t+15,10))
```

Note that again, these theorems not only state the fact that count-to-infinity occurs, but also explicitly predict the precise increasing pattern. In establishing the theorems here, PVS with its automatic and enforced proof checking mechanism manifests its value in verifying complicated properties that is not obvious from intuition: we did not discover a minor but critical error in the increasing pattern derived on paper and pencil until the use of PVS. Indeed, the mechanical PVS proofs allows *DNV* verification results to be replayed and checked by any third party. These three theorems is proved via a set of intermediate theorems (i.e. lemmas), displayed as follows:

```
bestHop_ce_exist: Theorem
  FORALL (T: Time) (C: Cost):
    T>110 AND (EXISTS (i: posnat): T=i*5) AND
    bestHop(a,e,b,C,T-5,10) AND bestHop(a,e,c,C,T-5,10) AND
      (FORALL (z: Node) (c: Cost):
         NOT (bestHop(b,e,z,c,T-5,10) OR (bestHop(c,e,z,c,T-5,10))))
            => bestHop(c,e,a,C+1,T,10)

bestHop_ce_exist2: Theorem
  FORALL (T: Time) (C: Cost):
    T>110 AND (EXISTS (i: posnat): T=i*5) AND
    bestHop(c,e,a,C,T-5,10) AND bestHop(b,e,a,C,T-5,10) AND
    (FORALL (z: Node) (c: Cost): NOT bestHop(a,e,z,c,T-5,10))
      => bestHop(c,e,b,C+1,T,10)

bestHop_ae_t: Theorem
```

```
FORALL (T: Time) (C: Cost):
  T>110 AND (EXISTS (i: posnat): T=i*5) AND
  bestHop(b,e,c,C,T-5,10) AND bestHop(c,e,b,C,T-5,10)
    => bestHop(a,e,c,C+1,T,10)
```

Note that these results derived on a specific instance, can be generalized to arbitrary network that contains a subnetwork has the topology in Figure 2(a). A complete list of mechanically checked proof can be found at [6].

6 Related Work

We briefly survey existing work on network protocol verification.

Classical theorem proving has been used in the past few decades for verification of network protocols [29, 5, 10, 4]. 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. *DNV* avoids this problem by using a common executable declarative networking language that can be directly verified in a theorem prover.

Runtime verification techniques (e.g. [15, 16, 27]) is 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*. In particular, any universal quantified properties, such as `bestPathStrong` we demonstrated in Section 3.2 is not checkable in runtime verification based on *NDlog* query engine.

Model checking is a collection of algorithmic techniques for checking temporal properties of system instances based on exhaustive state space exploration. Recent significant advances in model checking network protocol implementations include *MaceMC* [13] and *CMC* [7]. Compared to *DNV*, 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. While the heuristics used in exploration maximize the chances of detecting property violations, they are typically inconclusive and restricted to small network instances and temporal properties.

By adopting a theorem-proving based approach in this paper, *DNV* is more expressive and flexible compared to *MaceMC* and *CMC*, since higher-order logics can be used to specify network properties. In addition, since *DNV* directly verifies declarative networking specifications, an explicit model extraction step via execution exploration is not required.

7 Conclusion and Future Work

In this paper, we present *DNV*, a declarative network verifier that utilizes theorem proving to establish properties of declarative networking protocols. *DNV* addresses a practical problem in the networking domain: the ability to *specify*, *execute* and *verify* a network protocol within a unified language and system. *DNV* achieves this via the use of a declarative language, that can not only be used to execute network protocols, but provide a natural mapping to logical axioms for validating protocol correctness within a general purpose theorem prover. Using the PVS theorem prover, we validate the use of *DNV* to prove properties of various declarative routing protocols. To our best knowledge, *DNV* is one of the first attempts at using theorem proving to verify system implementation and eventual semantics of protocols with soft-state properties in dynamic networks.

We are in the process of applying *DNV* to more complex overlay networks, and reasoning about routing protocols, particularly when integrated with policies [11, 9]. Our initial experiences suggest that *DNV* 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, most general-purpose theorem provers utilize an interactive proof process that requires experience of these provers. We are currently exploring ways in which one can automate the proof construction by using domain-specific proof strategies that users can develop with the PVS proof strategy language [22, 2], hence lowering the barrier for adoption by network designers. In Appendix A, we outline the general proof strategies that we have used for all the proofs presented in this paper. Our general rules of thumb in proving these theorems suggests that an automated proof construction process is attainable. However, more work needs to be done to validate this possibility.

Second, recent work on boolean satisfiability (SAT) solving and satisfiability modulo theories (SMT) [33, 34], as well as the development in automated first-order theorem provers [28] have enable fully automated theorem proving based approaches to software and hardware verification. This provides an alternative proof automation support to PVS proof strategies developments that we plan to incorporate into *DNV*.

Third, we have limited our proofs to a subset of the complete *NDlog* language, omitting the treatment of *location specifiers* for distribution. We conjecture that in the presence of reliable in-order communication channels, handling of location specifiers does not require radically different treatment since they affect distributed evaluation of *NDlog* programs but not *NDlog* semantics *per se*. Interestingly, the prevalent use of soft-state and periodic refreshes in *NDlog* programs provides a natural mechanism for ensuring eventually correct semantics in the face of reordered messages, node disconnection, and other unpredictable occurrences that are common in distributed systems.

Finally, despite the lack of completeness in *MaceMC* and *CMC*, there are situations when model-checking based techniques are useful for facilitating proof automation in theorem proving. We intend to explore combining model checking and theorem proving in the following way: (1) model-checking a abstract scaled-down finite-state instance (2) generalize the scaled-down instance to the original problem by theorem proving, and (3) justify in the theorem prover that the correctness of the downscaled system established by model checking implies the correctness of the original full-blown system. This approach has proven successful in a variety of industrial-strength protocol verification experiments [12]. We plan to leverage PVS's support for CTL (variant of temporal logic) model-checking [21] to integrate model checking into *DNV*.

8 Acknowledgments

This work was partially sponsored by NSF CNS-0721845 and DARPA through Air Force Research Laboratory (AFRL) Contract FA8750-07-C-0169. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

References

1. P2: Declarative Networking. http://p2.cs.berkeley.edu.
2. M. Archer, B. D. Vito, and C. Muñoz. Developing user strategies in PVS: A tutorial. In *STRATA'03*, NASA/CP-2003-212448, 2003.
3. Y. Bertot and P. Castéran. Interactive theorem proving and program development. coq'art: The calculus of inductive constructions, 2004.
4. K. Bhargavan, D. Obradovic, and C. A. Gunter. Formal verification of standards for distance vector routing protocols. *J. ACM*, 49(4):538–576, 2002.
5. R. Cardell-Oliver. On the use of the hol system for protocol verification. In *TPHOLs*, pages 59–62, 1991.
6. DNV use cases for protocol verification. http://www.seas.upenn.edu/~anduo/dnv.html.
7. D. Engler and M. Musuvathi. Model-checking large network protocol implementations. In *NSDI*, 2004.
8. A. R. et. al. MACEDON: Methodology for Automatically Creating, Evaluating, and Designing Overlay Networks",. In *NSDI*, 2004.
9. N. Feamster and H. Balakrishnan. Correctness Properties for Internet Routing. In *Allerton Conference on Communication, Control, and Computing*, 2005.
10. A. P. Felty, D. J. Howe, and F. A. Stomp. Protocol verification in nuprl. In *CAV*. Springer-Verlag, 1998.
11. T. G. Griffin and J. L. Sobrinho. Metarouting. In *ACM SIGCOMM*, 2005.
12. K. Havelund and N. Shankar. Experiments in theorem proving and model checking for protocol verification. In *FME*, 1996.
13. C. Killian, J. Anderson, R. Jhala, and A. Vahdat. Life, death, and the critical transition: Finding liveness bugs in systems code. In *NSDI*, 2007.
14. C. E. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. M. Vahdat. Mace: language support for building distributed systems. In *PLDI*, 2007.
15. I. Lee, S. Kannan, M. Kim, O. Sokolsky, and M. Viswanathan. Runtime assurance based on formal specifications. In *PDPTA*, 1999.
16. X. Liu, Z. Guo, X. Wang, F. Chen, X. L. J. Tang, M. Wu, M. F. Kaashoek, and Z. Zhang. D3S: Debugging Deployed Distributed Systems. In *NSDI*, 2008.
17. B. T. Loo. The Design and Implementation of Declarative Networks (Ph.D. Dissertation). Technical Report UCB/EECS-2006-177, UC Berkeley, 2006.

18. B. T. Loo, T. Condie, M. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative Networking: Language, Execution and Optimization. In *ACM SIGMOD*, 2006.
19. B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing Declarative Overlays. In *ACM SOSP*, 2005.
20. B. T. Loo, J. M. Hellerstein, I. Stoica, and R. Ramakrishnan. Declarative Routing: Extensible Routing with Declarative Queries. In *ACM SIGCOMM*, 2005.
21. S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. K. Srivas. PVS: Combining Specification, Proof Checking, and Model Checking. In *CAV*, 1996.
22. S. Owre and N. Shankar. Writing PVS proof strategies. In *STRATA'03*, 2003.
23. L. Peterson and B. Davie. *Computer Networks: A Systems Approach, Fourth Edition*. Morgan-KaufMann, 2007.
24. L. Peterson, S. Shenker, and J. Turner. Overcoming the Internet Impasse Through Virtualization. In *HotNets-III*, 2004.
25. R. Ramakrishnan and J. D. Ullman. A Survey of Research on Deductive Database Systems. *Journal of Logic Programming*, 23(2):125–149, 1993.
26. S. Raman and S. McCanne. A model, analysis, and protocol framework for soft state-based communication. In *SIGCOMM*, pages 15–25, 1999.
27. P. Reynolds, C. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat. Pip: Detecting the Unexpected in Distributed Systems. In *NSDI*, 2006.
28. A. Riazanov and A. Voronkov. The design and implementation of vampire. *AI Commun.*, 15(2):91–110, 2002.
29. J. Rushby. Specification, proof checking, and model checking for protocols and distributed systems with PVS. Tutorial FORTE X/PSTV XVII '97, 1997.
30. Serge Abiteboul, et.al. *Foundations of Databases*. Addison-Wesley, 1995.
31. I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable P2P Lookup Service for Internet Applications. In *SIGCOMM*, 2001.
32. A. Wang, P. Basu, B. T. Loo, and O. Sokolsky. Declarative Network Verification. University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-08-34, 2008.
33. Yices. yices.csl.sri.com/.
34. Z3. http://research.microsoft.com/projects/Z3/.

A Interactive Theorem Proving

In this appendix, we present a set of proof methodologies that are aimed towards aiding a non theorem-proving expert on the proof process, as long she/he has basic knowledge of standard proof techniques used in logic and mathematics. The proof methodologies presented also lays the groundwork for future automation in the proof process.

We present the proof methodologies from the perspective of PVS, which is the theorem prover adopted by *DNV*. While the terminologies are specific to PVS, the general techniques are agnostic, and hence applicable to any general purpose theorem prover. For instance, the proof methodologies used in the Coq [3] theorem prover are similar, when used for proving the network protocols presented in this paper.

In the rest of this section, we first present the general methodology in Section A.1, followed by a specific example in Section A.2 based on a theorem presented and proven in this paper.

A.1 PVS Proving Guide

The proof objective in PVS is to develop a proof tree where all leaves are true. Accordingly PVS proof development is carried out in a goal-directed style where user starts from the proof goal, and applies PVS command recursively to reduce proof root to subgoals that are trivially true. Each node in the proof tree represents a intermediate logic sequent derived from the proof root. A sequent of the form

$$\begin{array}{ll} \{-1\} & A_1 \cdots \\ \{-m\} & A_m \\ \vdash & \\ \{1\} & C_1 \cdots \\ \{n\} & C_n \end{array}$$

is interpreted as $(A_1 \wedge \cdots \wedge A_m) \implies (C_1 \vee \cdots C_n)$, i.e. the conjunction of the set of antecedents $A_1 \cdots A_m$ implies the disjunction of consequences $C_1 \cdots C_n$. In the language of sequent calculus, the proof task is then to reduce the sequent into one of the following forms: (1) one of the antecedents is false or (2) one of consequences is true or (3) one of the antecedents is logically equivalent to one of the consequences.

PVS Build-in Commands One distinguished feature of PVS is its significantly larger deduction steps (hence simplifier proof development) compared with other popular proof assistants. This is largely due to PVS's build-in tools for automating portions of the proof process. PVS provides two levels of proof commands to aid user-derived proof development: (1) high-level proof strategies or decision procedure that take care of tedious and repeated proof patterns and (2) low-level commands that enables user finer control when built-in heuristic alone does not completely discharge the proof goal.

In principle, it is always good and safe to try the high-level and fully automated proof strategies first. Such proof strategy either completely discharges the goal or reduce the goal to a more manageable form by performing certain simplification. [5]

The following build-in proof strategies are typically the first to be attempted:

- `skosimp*`, `skolem!` and `inst`, for manually quantifier proof. The intuitive behind `skosimp*, skolem!` is to replace the quantified variable with a fixed arbitrary constant, followed by repeated simplification, whereas `inst` instantiates the existential quantified variable with a proper instance.

[5] A subtle issue is that a greedy heuristic (proof strategy) sometimes would reduce a provable goal to un-provable form. This is because a reduction can be strengthening (or weakening) both the antecedents and consequences, and may make implication between antecedents and consequences no longer true. However, one can always revert to proof status before applying such proof strategy and therefore it is always safe to try high-level strategy first.

- `induct` and `induct-rule` for induction. While `induct` performs induction on natural numbers, `induct-rule` performs induction according to some recursive definition.

Other most frequently-used high-level strategies for propositional reasoning include the following:

- `expand`, `grind`, `ground` and `prop` for non-inductive, definition expansion, arithmetic, equality, and quantifier reasoning.

General "Rules of Thumb" We have identify the following eight forms of proof goals that we frequently encounter, as well as providing a *solution* that works most of the time in our proof development.

1. **Remove universal quantifiers:**

   ```
   {-1} A1 ...
   |------
   {1} (FORALL x): P(x)...
   ```

   Try apply `(skosimp*)` to automatically remove universal quantified variable x by replacing x with fixed and arbitrary skolem constant. Note that P can be a very complex compound predicate built from many primitive functions and predicts over x. A dual-form is *existential quantified antecedent*:

   ```
   {-1} (EXISTS x): P(x) ...
   |------
   {1} C1 ...
   ```

   Try apply `skolem!`, which performs a similar standard skolemization over x.
2. **Remove implications:** if the goal of the form
   $\vdash (A_1 \wedge A_2 \wedge \cdots \wedge A_m \implies (C_1 \vee C_2 \vee \cdots \vee C_n))$ as denoted by PVS sequent:

   ```
   |------
   A1 AND A2 AND Am => (C1 OR C2 OR Cn)
   ```

   Try apply `(prop)` to reduce the goal to more the following manageable form

   ```
   {-1} A1 ...
   {-m} Am
   |------
   {1} C1...
   {n} Cn
   ```

   Furthermore, Since proof of any of the consequents: C_i is sufficient to prove the goal, one could try `(delete 1 ... (i-1) (i+1) ... m)` to remove all the irrelevant consequents $C_j, j \neq i$, given that the user correctly predict the truth of such C_i.
3. **Unfold user-defined definitions:** if the goal does not contain quantifications as shown in case 1, or implications as shown in case 2, but contains some user-defined predicate or function P as denoted in the following form:

```
|------
{1} P(c)
```

Then try `(expand "1" "P")` to unfold the definition of `P`, and allow the user to continue proof development by using useful information in the definition of `P`.

4. **Inductive reasoning:** if the goal does not contain quantifications or implication as in case 1 and 2, but contains some user-defined recursive definition/function/predicate r as denoted in the following form:

   ```
   |------
   {1} ... r(x) ...
   ```

   Then try `(induct "1" "x")` or `(induct_rule 1 "r")` to perform inductive reasoning. The difference is that `induct` performs induction over natural number x whereas `induct_rule` performs induction according to the recursive generating rule defined by r.
5. **Instantiate existential variables**: if the goal contains a existential quantified formula $\exists x.P(x)$ where there exists a constant c which makes $P(c)$ true, as displayed:

   ```
   |------
   {1} (EXISTS x): P(x)
   ```

   Then by picking a constant `c` such that `P(x)` evaluates to truth, `(inst "1" "c")` would solve this goal, and complete this branch of proof tree.
6. **Using lemmas and axioms**: if all the above proof attempts failed in reducing or solving the proof, but some established lemma/theorem proved before or some user-defined axiom can be used to reduce the current goal, then either `(lemma "theorem_name")` or `(use "theorem_name")` would bring that lemma/theorem/axiom identified by name `theorem_name` to the antecedent of the current goal.
7. **PVS build-in strategies**: whenever one is stuck in the proof, it is typically useful to attempt one of the standard PVS built-in proof strategies such as `(grind)`, `(assert)`, `(ground)`, and `(prop)` to complete the rest of the proof. While this strategy does not work in all cases, they are typically useful in getting to the next step in the proof process.
8. **Backtracking and postponing:** Often times, it is useful to backtrack in the proof process or defer the current goal to be proven. This is achievable using the command `(undo)` which reverts to the previous proof state by eliminate the effects of last command. Another useful control command is `(postpone)` which allows the user to traverse through all the subgoals, by postpone the proof of the current goal.

A.2 Example: `bestPathStrong`

Having presented the basic methodology for interactive theorem proving, we now proceed to demonstrate this methodology using the `bestPathStrong` theorem presented in Section 3.2. Recall that we have proven the theorem `bestPathStrong` $\vdash \forall(S, D, C, P).bestPath(S, D, P, C) \implies \neg(\exists C_2, Z, P_2.path(S, D, Z, P_2, C_2) \wedge (C_2 < C))$

Using the actual PVS proof transcripts of `bestPathStrong`, We illustrate the interactive proof process by employing the rules we identified above as well as the significant proof automation support in PVS.
First, when the PVS prover is started, it begins with the original goal to be proven, i.e. the theorem `bestPathStrong` as follows:

```
bestPathStrong :
  |-------
{1} FORALL (S, D: Node)(C: Metric)(P: Path):
      bestPath(S,D,P,C) =>
        NOT (EXISTS (C2:Metric)(Z:Node)(P2:Path):
             path(S, D, Z, P2, C2) AND C2 < C)
Rule?
```

PVS is now ready to prove $\vdash bestPathStrong$ [6] and prompt `Rule?` is waiting for PVS a commands to guide the proof development. Observe that this universal quantified goal matches the first proof development pattern we described above, we then try `skosimp*` to perform *Skolemization*, and PVS reduced `bestPathStrong` to the following for us:

```
Rule? (skosimp*)
Repeatedly Skolemizing and flattening,
this simplifies to:
bestPathStrong :
{-1}  bestPath(S!1, D!1, P!1, C!1)
{-2}  path(S!1, D!1, Z!1, P2!1, C2!1)
{-3}  C2!1 < C!1
  |-------
```

Recall that if there is no consequent presented, the proof task is to show one of the antecedent evaluates to false. Therefore, this simplification/reduction says to prove `bestPathStrong` it is sufficient to find out some skolem constant `S!1, D!1, P!1, ...` such that one of the three antecedent {`-1,-2,-3`} evaluates to false. We observed that this is the case of rule 3, and we then try `expand` to unfold the definition of `bestPath`:

```
Rule?: (expand bestPath)
Expanding the definition of bestPath,
this simplifies to:
bestPathStrong :
{-1}  (bestPathCost(S!1,D!1,C!1)) AND
        (EXISTS (Z:Node):path(S!1,D!1,Z,P!1,C!1))
[-2]  path(S!1,D!1,Z!1,P2!1,C2!1)
[-3]  C2!1 < C!1
  |-------
```

We observe some changes in antecedent {`-1`} and would like PVS to reduce it to a more readable form. Since it is non-inductive and quantifier free, we try the propositional simplification strategy `prop`, and PVS produces the following:

[6] PVS starts from sequent with only one consequence `bestPathStrong` (labeled {`1`}) and no antecedent, which intuitively means that we are required to prove that `bestPathStrong` must be unconditionally true.

```
Rule?: (prop)
Applying propositional simplification,
this simplifies to:
bestPathStrong :
{-1}  (bestPathCost(S!1,D!1,C!1))
{-2}  EXISTS (Z: Node): path(S!1,D!1,Z,P!1,C!1)
[-3]  path(S!1,D!1,Z!1,P2!1,C2!1)
[-4]  C2!1<C!1
  |-------
```

Regarding the 8 rules, we then repeatedly guess and supply following commands: `(expand bestPathCost) (prop) (skosimp*) (inst -2 C2!1)` And finally arrive at the following display

```
[-1]  path(S!1, D!1, Z!2, P!2, C!1)
{-2}  (EXISTS (Z2:Node)(P2:Path):
        path(S!1,D!1,Z2,P2,C2!1)) => C!1<=C2!1
[-3]  path(S!1, D!1, Z!3, P!1, C!1)
[-4]  path(S!1, D!1, Z!1, P2!1, C2!1)
[-5]  C2!1 < C!1
  |-------
```

At this stage, all user-defined definitions are unfolded, no induction reasoning is required, and most important of all, we have manually provide instantiation in `(inst -2 C2!1)`. Therefore, we decide it is time for PVS to try discharge the goal completely for us, and we try the most greedy high level heuristic `grind`:

```
Rule?: (grind)
Trying repeated skolemization, instantiation,
and if-lifting,Q.E.D.
```

The proof is hence completed.