# A Theorem Proving Approach Towards Declarative Networking

Anduo Wang[1] Boon Thau Loo[1]
Changbin Liu[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.** We present the *DRIVER* system for designing, analyzing and implementing network protocols. *DRIVER* leverages declarative networking, a recent innovation that enables network protocols to be concisely specified and implemented using declarative languages. *DRIVER* takes as input declarative networking specifications written in the Network Datalog (*NDlog*) query language, and maps that automatically into logical specifications that can be directly used in existing theorem provers to validate protocol correctness. As an alternative approach, network designer can supply a component-based model of their routing design, automatically generate PVS specifications for verification and subsequent compilation into verified declarative network implementations. We demonstrate the use of *DRIVER* for synthesizing and verifying a variety of well-known network routing protocols.

## 1 Introduction

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* [13, 12], 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)* [18] 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.

## 2 Background

Declarative networks are implemented 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 implemented in a few lines of declarative code [13], and complex protocols in orders of magnitude less code [12] compared to traditional imperative implementations. This compact and high-level language 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 illustrate *NDlog* using a simple example of two rules that computes all pairs of reachable nodes:

```
r1 reachable(@S,N) :- link(@S,N).
r2 reachable(@S,D) :- link(@S,N), reachable(@N,D).
Query reachable(@S,D).
```

The rules `r1` and `r2` specify a distributed transitive closure computation, where rule `r1` computes all pairs of nodes reachable within a single hop from all input links (denoted by `neighbor`), and rule `r2` expresses that "if there is a link from `S` to `N`, and `N` can reach `D`, then `S` can reach `D`."

*NDlog* supports a *location specifier* in each predicate, expressed with the `@` symbol followed by an attribute. This attribute is used to denote the source location of each corresponding tuple. For example, all `reachable` and `link` tuples are stored based on the `@S` address field. The output of interest is the set of all `reachable(@S,D)` tuples, representing reachable pairs of nodes from `S` to `D`.

### 2.2 Soft-state Storage Model

Declarative networking incorporates support *soft-state* [15] 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.

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 zero lifetime. Since events are not stored, they are primarily used to trigger rules periodically or in response to network events.

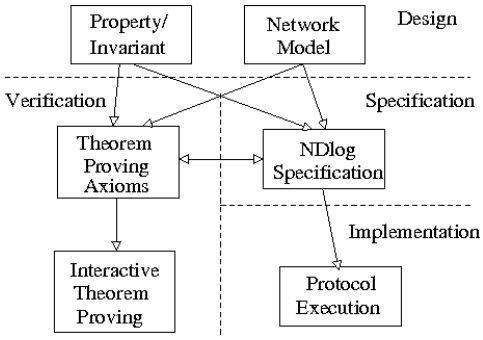


**Fig. 1.** Overview of DRIVER

## 3 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.

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 verification 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 substan-

tial 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 [18] 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.

## 4 Implementing Networks from Verified Specifications

We present the second option for bridging network verification and implementation as described in the previous section. In this approach, a network designer first develops component-based models for their networks. These models are then used to generate formal specifications that can be directly verified in PVS, and the verified PVS specifications can be further compiled into *NDlog* programs for execution.

Our main driving example is based on a component-based model of routing protocols. Given this model, a large family of routing protocols can be implemented simply by customizing subcomponents. The use of components provides the usual benefits of modularity and re-usability. More importantly, in our context, it enables a straightforward translation to formal specifications for verification in PVS or any other general purpose theorem prover.

Our approach of representing and verifying system implementation as components and predicates is adapted from similar techniques in hardware verification [5, 16], where circuits can easily be modeled by composing together electrical components. Interestingly, component-based abstractions have similarly been explored in the networking literature (e.g. [11, 14, 17]), where network protocols are typically composed from existing ones by layered (vertically) or bridging (horizontally) with existing protocols.

### 4.1 Component-based Model

Before going into the specifics of routing implementations derivation from verified specification, we first provide an introduction to the component-based model adapted from hardware verification, and describe its translation to PVS axioms and equivalent *NDlog* programs. The translation is made possible via the use of *predicative specifications* [5, 16]. In a nutshell, each component is specified as a predicate (relation) over all its external attributes. The external attributes constitute the component's interface, whereas the internal sub-components constitute its implementation in terms of sets of constraints.

To illustrate, we consider an example component `c` shown in Figure 2 (a) with input `I1, I2` and output `O`. This component is implemented in terms of three sub-components `c1, c2, c3`, as shown in Figure 2 (b).

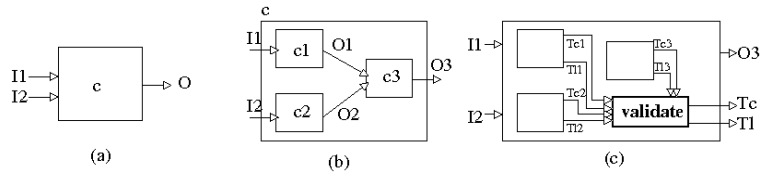


**Fig. 2.** *Component Representation*

The predicative specification of the corresponding component in PVS is then written as:

```
c(I1,I2,O3): INDUCTIVE bool =
     EXISTS (O1,O2): c1(I1,O1) AND c2(I2,O2) AND c3(O1,O2,O3)

c1(I,O): INDUCTIVE bool = Constraints ct1(I,O)
c2(I,O): INDUCTIVE bool = Constraints ct2(I,O)
c3(I1,I2,O): INDUCTIVE bool = Constraints ct3(I1,I2,O)
```

The top-level component `c` is defined as the conjunction of its sub-components. Within each subcomponents `c1-c3`, there are additional constraints `ct1-ct3` which are typically a conjunction of predicates. We use PVS `Inductive` definition to ensure that all components represent the smallest sets satisfying the constraints in the body.

**Translating to Datalog programs:** Given the above PVS axioms, there is a straightforward translation to equivalent Datalog programs, by leveraging the proof-theoretic semantics of Datalog. For instance, the following four Datalog rules `t1-t4` implement component `C`:

```
t1 c(I1,I2,O)  :- c1(I1,O1), c2(I2,O2), c3(O1,O2,O3).
t2 c1(I,O)     :- ct1(I,O).
t3 c2(I,O)     :- ct2(I,O).
t4 c3(I1,I2,O) :- ct3(I1,I2,O).
Query c(I1,I2,O).
```

The above translation is feasible as long as each individual set of component constraints (i.e. `ct1`, `ct2` and `ct3`) can be specified as conjunction of set of predefined predicates. Additional location specifiers are supplied by the network designer as part of the model in order to compile the equivalent distributed *NDlog* programs with the correctly annotated location specifiers. Together with the above rules, one can further specify the output of interest in `Query` statement, which in this case is simply the `c` component, i.e. `c(I1,I2,O)`.

**Adding soft-state constraints:** To support protocol verification in dynamic networks via soft-state semantics as described in Section 2.2, we add two additional attributes `Tc` and `Tl` to each component's interface, as shown in Figure 2 (c). `Tc` and `Tl` denote the creation time and lifetime of each component respectively. The creation times `Tc` are typically a variable in PVS specification, and

lifetimes Tl are soft-state lifetimes initialized by the network designer. To illustrate by a concrete example, consider the component c introduced earlier. The equivalent PVS specification capturing soft-state semantics is as follows:

```
c(I1,I2,O3,Tc,Tl): INDUCTIVE bool =
EXISTS (O1,O2,Tc1,Tl1): c1(I1,O1,Tc1,Tl1) AND c2(I2,O2,Tc2,Tl2) AND
  c3(O1,O2,O3,Tc3,Tl3) AND validate(Tc,Tl,Tc1,Tl1,Tc2,Tl2,Tc3,Tl3)
```

We further add a validate component as a subcomponent to impose the constraints that only components active within the same period can interact with each other. The validate component also determines the creation time Tc and lifetime Tl for the top-level component based on those of sub-components. For example, consider the following PVS specification of a validate component:

```
validate(Tc,Tl,Tc1,Tl1,Tc2,Tl2,Tc3,Tl3): INDUCTIVE bool =
Tl=TTL AND Tc1<Tc<=Tc1+Tl1 AND Tc2<Tc<=Tc2+Tl2 AND Tc3<Tc<Tc3+Tl3
  AND Tc=max(Tc1,Tc2,Tc3)
```

The above validate subcomponent takes as input the lifetimes of all the other components, and ensures that the creation time Tc of the resulting component is set to the max of all creation times among its components, and that Tl is set to the specified soft-state lifetime (TTL) of the component c. The validate also includes additional constraints that ensure that only active components whose lifetimes overlap within the same time window are allowed to interact with each other.

The translation to equivalent Datalog program is straightforward and we omit for brevity.

### 4.2 Distance Vector Protocol

To demonstrate the component-based model and translation to PVS specifications and *NDlog* programs, we provide a representative example based on the *distance-vector protocol*. This protocol is typically used for *inter-domain* routing, i.e. computing shortest-paths within a local area network or administrative domain (Internet service provider). On the other hand, the path vector protocol presented earlier is generally used for computing routes across over administrative domains. Interestingly, mapping from one protocol to another only require only minor changes to the component specifications.

In the distance vector protocol, each router in the network executing the protocol maintains a routing table, and periodically advertises its current best routes to its neighbors, and updates its routing knowledge when receiving route advertisements from neighbors. Figure 3 shows the component-based model for an instance of the distance-vector protocol in which route advertisements are generated every 5 seconds, and all received routes are stored at each node for 10 seconds to recompute current best hops along shortest paths with minimal hop cost.

The model consists of three main components: hop, hopMsg and bestHop respectively, depicted as **bold** boxes in Figure 3. Besides these three top-level composite components that are built upon sub-components, we introduce a set of *atomic components* depicted by regular (not in bold) boxes, to represent input data (representing a node's prior knowledge of the network) or a pre-defined function (for arithmetic and path concatenation). For example link denotes the

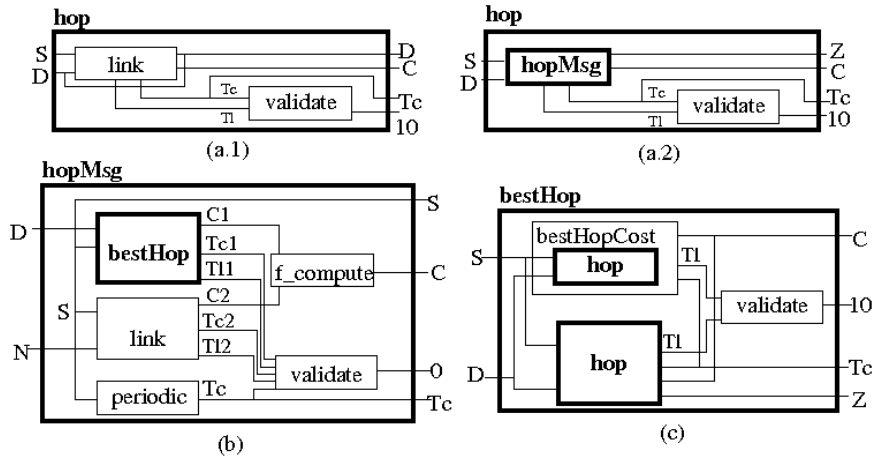


**Fig. 3.** *Component Representation*

set of neighbors for each node, and `f_compute` is a function computing best route cost.

Figure 3 (a) shows `hop` component with attributes `S,D,Z,C` is used to compute all possible routes from `S` to `D` via next hop `Z` with cost C. The component actually have two instances: the left `hop` component instance (a.1) is derived via a direct one-hop link, whereas in (a.2) the component instance computes hops of increasing hop cost recursively from advertisements received from neighbors (`hopMsg`).

Since hops have associated lifetimes, soft-state attributes described in Section 4.1 are added as additional attributes to each component. The PVS specification for `hop` is as follows:

```
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 Tl=10 AND Tc=Tc2+5)
```

Since route advertisements are triggered every 5 seconds, the additional constraint `Tc=Tc2+5` defined by the `validate` component requires that the creation time of each newly computed `hop` to be advanced accordingly. For ease of exposition, we unfold the `validate` component constraints into the above PVS specification.

Figure 3 (b) shows the `hopMsg` component that denotes a route advertisement. The component's implementation is represented by the conjunction of constraints imposed by all its internal components `bestHop`, `link`, `periodic`, `f_compute` and `validate`. The `periodic` component is a special atomic component `periodic` used to periodically trigger route advertisement at node `S` at time `Tc`. `f_compute` is a function used to compute new route costs from existing route advertisement and `link` costs. The PVS specification for `hopMsg` is as follows:

```
hopMsg(N,D,S,C,Tc,Tl): INDUCTIVE bool =
(EXISTS (Tc2,Tc3,C1,C2:Time): periodic(S,5,Tc) AND
  bestHop(S,D,Z,C1,Tc2,10) AND link(S,N,C2,Tc3,10) AND
  Tc2<Tc<=Tc2+10 AND Tc3<Tc<=Tc3+10 AND C=C1+C2 AND Tl=0)
```

Intuitively it means S will send N a route to reach destination D of cost C via itself if and only if S knows a best route to reach D of cost C1, as indicated by bestHop(S,D,Z,C1,Tc2,10), and that S is N's direct neighbor with a link of cost C1. Note that validate component above sets the creation time of hopMsg as that of the triggering component periodic, and in addition requires all the internal components to be alive during the same window of period.

Finally, Figure 3 (c) shows the bestHop component, which is formalized in PVS as follows:

```
bestHop(S,D,Z,C,Tc,Tl): INDUCTIVE bool =
  bestHopCost(S,D,C,Tc,10) AND hop(S,D,Z,C,Tc,10)
```

For each pair of source S and destination D, the above PVS specifications computes the next hop Z with minimal cost C along the shortest path to the destination. The component utilizes a subcomponent bestHopCost that is used to compute the min aggregation over attribute C of hop to select the best (lowest) cost. We view this aggregation as an atomic service provided by bestHopCost and specify it in PVS as follows:

```
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))
```

Given the above PVS specifications and additional information on location specifiers of predicates, the following *NDlog* program can be automatically generated.

```
dv1 hop(@S,D,D,C,Tc,10):-link(@S,D,C,Tc,10).
dv2 hop(@S,D,Z,C,Tc,10):-hopMsg(@S,D,Z,C,Tc2),Tc=Tc2+5
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,0):- periodic_dv(@S,5,Tc), link(@S,N,C2,Tc2,10),
     bestHop(@S,D,Z,C1,Tc1,10),C=C1+C2,Tc2<Tc<=Tc2+10,Tc1<Tc<=Tc1+10.
```

The above *NDlog* program implements the declarative distance-vector protocol as presented in references [13, 12]. The *min* keyword in rule dv3 is a built-in aggregation construct commonly used in database query languages, and it corresponds to the min computation in the bestHopCost component. The interested reader is referred to these references on detailed performance evaluation of the above declarative protocol using the *P2* declarative networking engine.

### 4.3 Example Proofs: Convergence and Divergence Analysis

Given the PVS specifications, one can verify a variety of properties of the distance-vector protocol. Assuming distance vector is executed every 5 seconds, and all soft-state predicates have a lifetime of 10 seconds, network convergence can be expressed as:

```
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` converges (i.e. no longer changes).

The distance-vector protocol converges in the static case. However, in a dynamic network with link failure, the protocol can diverge, caused by a well-known problem known as the *count-to-infinity* problem where the protocol diverges in the presence of link failures. In a network of three nodes `a,b,d` with link failure occurred at time 100, divergence is captured by the following theorem:

```
bestHop_count_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 the distance vector protocol will diverge after link failure, because the best hop from `a` to `d` will increase indefinitely over time, a symptom of the count-to-infinity problem. Due to space constraints, we omit the proofs of the above theorem. The proof for the convergence case is relatively straightforward. The divergence proofs require us to supply additional axioms that describe link dynamics within a three-node cycle. We have also verified that the count-to-infinity problem exists in a cycle of nodes, and well-known fixes such as the *split-horizon* solution can avoid any two-node cycle, and that this solution is insufficient for preventing count-to-infinity problem in three-node cycle. For a complete list of theorems and proofs, refer to reference [7].

## 5 Related Work

In addition, we briefly compare the *DRIVER* system with existing work on network protocol verification and development.

*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* [10] and *CMC* [8]. 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 [2, 6, 9, 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. *DRIVER* is hence a significant improvement over existing usage of theorem proving [2, 9] 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.

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.

## 6 Future Work

We are exploring more automatic proof support to make *DRIVER* more approachable to 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.

## 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. 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. A. Camilleri, M. Gordon, and T. Melham. Hardware verification using higher-order logic. Technical Report 91, Computer Laboratory, University of Cambridge, June 1986.
6. R. Cardell-Oliver. On the use of the hol system for protocol verification. In *TPHOLs*, pages 59–62, 1991.
7. DNV use cases for protocol verification. http://www.seas.upenn.edu/∼anduo/dnv.html.
8. D. Engler and M. Musuvathi. Model-checking large network protocol implementations. In *NSDI*, 2004.
9. A. P. Felty, D. J. Howe, and F. A. Stomp. Protocol verification in nuprl. In *CAV*. Springer-Verlag, 1998.
10. C. Killian, J. Anderson, R. Jhala, and A. Vahdat. Life, death, and the critical transition: Finding liveness bugs in systems code. In *NSDI*, 2007.
11. E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. *ACM Transactions on Computer Systems*, 18(3):263–297, 2000.
12. B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing Declarative Overlays. In *ACM SOSP*, 2005.
13. B. T. Loo, J. M. Hellerstein, I. Stoica, and R. Ramakrishnan. Declarative Routing: Extensible Routing with Declarative Queries. In *ACM SIGCOMM*, 2005.
14. Y. Mao, B. T. Loo, Z. Ives, and J. M. Smith. MOSAIC: Unified Platform for Dynamic Overlay Selection and Composition. In *CoNEXT*, 2008.
15. S. Raman and S. McCanne. A model, analysis, and protocol framework for soft state-based communication. In *SIGCOMM*, pages 15–25, 1999.
16. M. Srivas, H. Rueß, and D. Cyrluk. Hardware verification using PVS. 1997.
17. The Ensemble Distributed Communication System. http://dsl.cs.technion.ac.il/projects/Ensemble/.
18. A. Wang, P. Basu, B. T. Loo, and O. Sokolsky. Declarative Network Verification. In *11th International Symposium on Practical Aspects of Declarative Languages (PADL)*, 2009.