Formally Verifiable Networking

Anduo Wang* Limin Jia* Changbin Liu*
Boon Thau Loo* Oleg Sokolsky* Prithwish Basu†
University of Pennsylvania* BBN Technologies†
{anduo,liminjia,changbl,boonloo,sokolsky}@seas.upenn.edu, pbasu@bbn.com

ABSTRACT

This paper proposes *Formally Verifiable Networking* (*FVN*), a novel approach towards unifying the design, specification, implementation, and verification of networking protocols within a logic-based framework. In *FVN*, formal logical statements are used to specify the behavior and the properties of the protocol. *FVN* uses *declarative networking* as an intermediary layer between high-level logical specifications of the network model and low-level implementations. A theorem prover is used to statically verify the properties of declarative network protocols. Moreover, a property preserving translation exists for generating declarative networking implementations from verified formal specifications. We further demonstrate the possibility of designing and specifying well-behaved network protocols with correctness guarantees in *FVN* using meta-models in a systematic and compositional way.

1. INTRODUCTION

In recent years, there has been growing interest in the formal verification of network protocol design and implementation. On the theoretical front, algebraic models such as metarouting [9] are used to formalize routing protocols with convergence guarantees. Concurrently, several practical software tools and testing platforms have been proposed to facilitate the verification of existing networked systems. These include *runtime verification* platforms (e.g. [20]) that provide mechanisms for checking at runtime that a system does not violate expected properties; and *model-checking* tools (e.g. [5, 13, 19]) that use a collection of algorithmic techniques for checking temporal properties of system instances based on exhaustive state space exploration.

At one extreme, practical tools based on runtime verification or model checking are applied to actual protocol implementations. However, they are in general not complete. Any runtime verification scheme will incur additional runtime overheads, and subtle bugs may require a long time to be encountered. On the other hand, model-checking suffers from the state explosion problem, where 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.

At the other extreme, formal models such as metarouting use a *correct-by-construction* approach: the verification of convergence is done once for the idealized algebra, and any routing protocol that implements the algebra is correct. The monotonicity requirements imposed by the idealized model limits the range of permissible protocols, and are unlikely to be adopted by actual routing protocol implementations.

In this paper, we aim to bridge the gap between formal verification of network protocols designs and the verification of actual implementations. Our proposed *Formally Verifiable Networking* (*FVN*) framework uses formal logical statements to specify the behavior and the properties of network protocols, and abstract network meta-models such as metarouting. The specified formal properties can be fed to a mechanized theorem prover such as PVS [17] or Coq [2] for static verification.

One advantage of using modern theorem proving techniques is that the logics of those provers can express properties beyond temporal properties that the majority of the model-checking techniques are bound to. Another advantage is that it is sound and complete: once a property is verified, it holds for all instances of the protocols. However, *FVN* does not exclude other incomplete techniques, such as model checking. In fact, one overarching goal of *FVN* is to smoothly integrate a variety of verification techniques for different classes of properties.

One important piece of the design of *FVN* is the use of *declarative networking* [16, 15], a declarative logical framework for protocol implementation. We utilize the *Network Datalog* (*NDlog*) declarative networking language as the intermediary layer between high-level logical specifications of the network model and low-level implementations. *NDlog* naturally bridges formal network models and protocol implementations. In one direction, *NDlog* programs can be automatically translated into logical statements that capture the semantics of the *NDlog* program and can be fed directly into existing tools for verification. Conversely, the verified logical specifications can be used to generate *NDlog* programs for execution via a *property-preserving* translation.

Moreover, when *NDlog* is used in conjunction with expressive logics (e.g. linear logic), *FVN* can provide a better model for soft-state, and more importantly, the promise to directly produce system models for model checking tools.

FVN leverages two bodies of work that in conjunction has significantly lowered the barriers of adoption of formal reasoning techniques in networking research. First, there has been significant progress on the understanding of logic-based techniques for bridging the specification and implementation divide. In addition to metarouting and declarative networking, there are now logical frameworks for reasoning about forwarding planes [12]. Second, modern theorem provers come with powerful proof engines that support a large portion of automated proof exploration which enable the proof of non-trivial difficulty problems with relatively modest human effort. Besides the built-in proof support, modern provers provide well-designed interfaces for customizing domain specific proof search strategies that can be further automated via user defined decision procedure, and integration with model-checkers, boolean satisfiability (SAT) solving and satisfiability modulo theories (SMT) [25].

2. OVERVIEW

We first present an overview of *FVN* framework, followed by a brief background on declarative networking.

2.1 FVN Framework

Figure 1 shows an overview of the *FVN* framework, which consists of the following four main components: *design*, *specification*, *verification*, and *implementation*. In the initial design phase, a network designer develops an abstract *network model* for the network protocol, and descriptions of the desired properties of the protocol, possibly informally. 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. In fact, a formal specification of the model is desirable at the design phase for verifying the model itself. We will provide an example based on metarouting in Section 3.3.

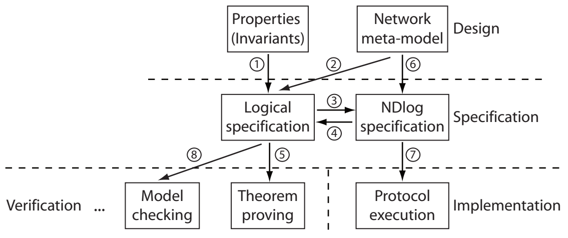

Figure 1: Overview of *FVN*

After initial design, the designer writes down formal logical specification for properties of concern (arc 1), and verifies that the implementation indeed satisfies these properties. This is the place where declarative networking is relevant. Instead of verifying a low-level implementation, we use *NDlog* as an intermediary language that properties are verified against. The benefit of using *NDlog* has two folds: first, *NDlog* is in itself declarative, and there is a natural synergy between the formal logical specification of properties and *NDlog* programs; second, robust tools [18] are available for generating low-level implementation from *NDlog* programs (arc 7).

Due to the synergy between *NDlog* programs and logic, there exists a two-way translation between *NDlog* program and logical specifications (arc 3 and 4). In *FVN*, a protocol designer has the choice of either generating *NDlog* programs directly, and then compiling *NDlog* programs into its logical specifications (arc 4) via an automatic tool and verifying properties in an theorem prover (arc 5); or first generating the logical specifications of the protocol's design (arc 1 and 2) and verifying the design using a theorem prover (arc 5), and then automatically generating the corresponding *NDlog* program from the verified logical specification (arc 3).

Ultimately, we envision that *FVN* will serve as a unifying framework, that uses formal logics as the specification language for properties, and eventually incorporate a number of different verification techniques, all of which share the specification language. For instance, aside from static verification using theorem provers, *FVN* can also take advantage of model-checking tools once it uses expressive logics such as linear logic for specifying protocols in the style of state transitions. We describe these opportunities (particularly arcs 6 and 8) in Section 4.

2.2 Background on Declarative Networking

Declarative networks are specified using *Network Datalog* (*NDlog*), which is a distributed recursive query language used for querying network graphs. Declarative networking programs are compiled into distributed execution plans that are based on the Click [14] execution model. When executed, these declarative networks perform efficiently relative to imperative implementations. We present an example *NDlog* program that implements the *path-vector* protocol.

```
r1 path(@S,D,P,C):-link(@S,D,C), P=f_init(S,D).
r2 path(@S,D,P,C):-link(@S,Z,C1), path(@Z,D,P2,C2),
                   C=C1+C2, P=f_concatPath(S,P2),
                   f_inPath(P2,S)=false.
r3 bestPathCost(@S,D,min<C>):-path(@S,D,P,C).
r4 bestPath(@S,D,P,C):-bestPathCost(@S,D,C),
                       path(@S,D,P,C).
```

The program takes as input `link(@S,D,C)` tuples, where each tuple 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 `r1-r2` recursively derive `path(@S,D,P,C)` tuples, where each tuple represents the fact that there is a path `P` from `S` to `D` with cost `C`. Rule `r1` computes one-hop reachability trivially given the neighbor set of `S` stored in `link(@S,D,C)`. Rule `r2` computes transitive reachability as follows: if there exists a link from `S` to `Z` with cost `C1`, and `Z` knows about a path `P2` to `D` with cost `C2`, then transitively, `S` can reach `D` via the path `f_concatPath(S,P2)` with cost `C1+C2`. Note that `r1-r2` also utilize two list manipulation functions to maintain path vector: `P`= `f_init(S,D)` initializes a path vector with two elements `S` and `D`, while `f_concatPath(S,P2)` prepends `S` to path vector `P2`. To prevent computing paths with cycles, an extra predicate `f_inPath(P,S)=false` is used in rule `r2`, where `f_inPath(P,S)` returns true if `S` is in the path vector `P`.

Rules `r3-r4` take as input `hop` tuples generated by rules `r1-r2`, and then derive the hop along the path with the minimal cost for each source/destination pair. The program outputs the set of `bestPath(@S,D,P,C)` tuples, each of which stores the shortest path `P` from `S` to `D`.

3. VERIFICATION IN FVN

To demonstrate *FVN*'s verification mechanisms concretely, we show the verification of *NDlog* programs (arc 4) in Section 3.1 via examples, and discuss the generation of *NDlog* programs from verified specifications (arc 3) in Section 3.2 where verified component-based specifications can be directly translated into executable *NDlog* programs. In Section 3.3, we demonstrate that the network meta-model itself can be formally specified in design phase, by using metarouting as our driving example. While we use PVS as the theorem prover, the methods are general, and other theorem provers such as Coq will work similarly.

3.1 NDlog Verification

One method to carry out the formal verification process, proposed by Wang *et al.* [22], is to automatically compile declarative networking programs written in *NDlog* into formal specifications recognizable by a theorem prover. This translation is depicted by arc 4 in Figure 1.

The verification is made possible by the natural mapping from *NDlog* rules to PVS axioms. We provide the following high-level intuitions behind the translation from *NDlog* to PVS formalizations. The translation leverages the proof-theoretic semantics of Datalog [1], the set of *NDlog* rules defining a predicate is equivalent to an inductively defined data type in PVS [1]. For instance, the following *inductive definition* in PVS is logically equivalent to rule `r1` and `r2` from Section 2.2.

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

The universal quantifiers over the attributes of `path` (i.e. `S,D,P,C`) are implicitly embedded and existential quantifiers such as `C1` and `C2` are explicitly stated. Next, the protocol designer specifies high-level properties of the protocol directly as theorems in the theorem prover. For instance, the *route optimality* property in the path-vector protocol can be expressed as follows:

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

The above theorem states that `P` is the optimal path from `S` to `D` with cost `C` implies that there does not exist another path `P2` from `S` to `D` with cost lower than `C`.

Given the above theorem, one can utilize PVS to carry out the proof process interactively. PVS, and other interactive theorem provers, provide some degree of automation to aid the process of proof construction.

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

The `bestPathStrong` theorem takes 7 proof steps. We omit the details of the proof process. The main takeaways is that PVS requires only a fraction of a second to carry out the actual proof, and built-in commands are available to mechanically advance the proof. When the proof is completed, it covers *all* instances of the network. In addition to the route optimality property above, reference [22] demonstrates additional proofs, for instance, the presence of count-to-infinity loops in the distance-vector protocol.

3.2 Verified Code Generation

The previous approach requires one to specify protocols in *NDlog* prior to verification. In this section, we demonstrate that *reverse translation*: given a conceptual network model at design phase, logical specifications are generated from the network model. This process corresponds to arc 2 in Figure 1. Once the logical specifications are verified, they are translated into *NDlog* programs for execution (arc 3).

We further observe that component-based network models are particularly amenable to the generation of *NDlog* programs from verified specifications. When formalized as logical specifications for verification, there is a straightforward translation to *NDlog* programs for execution (see Section 3.2.2).

3.2.1 Component-based BGP Model

As our driving example, we demonstrate *FVN*'s facility for generating executable *NDlog* programs from verified logical specifications based on a component conceptual model of routing protocols, with a focus on policy-based routing in BGP. Note that while our treatment of the model itself in this section is conceptual, one can formalize and verify this model itself at design phase, as we demonstrate in Section 3.3.

We adopt Griffin's BGP model [8, 7] which views BGP protocol as a series of *route transformations*. Each transformation is represented by a component that takes as input received routes, performs internal transformation based on the component specifications, and produces the output routes.

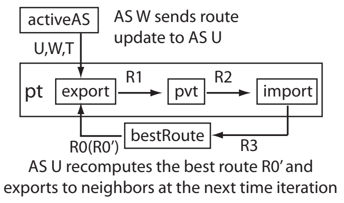
activeAS
AS W sends route update to AS U
U,W,T
pt
export
R1
pvt
R2
import
bestRoute
R0(R0')
R3
AS U recomputes the best route R0' and exports to neighbors at the next time iteration

Figure 2: BGP model as a series of route transformations.

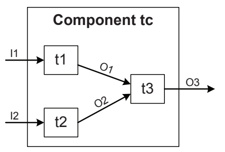

Figure 3: Example tc

Figure 2 shows an overview of the BGP model. At the top level, the BGP protocol is decomposed into three components: `activeAS`, `pt`, and `bestRoute`. The triggering component `activeAS(U,W,T)` specifies that at time `T`, AS `W` advertises its current best routes `R0` to each neighbor AS `U`. The *peer-transformation* component `pt(U,W,R0,R3,T)` represents the route propagation between neighboring AS `U` and `W`, such that AS `W` advertises its route to its neighbor `U` at time `T`. The input routes of peer-transformation are denoted by `R0` and output routes by `R3`. Finally, in the `bestRoute` component, AS `U` recomputes/selects its best route based on attributes such as path length and local preferences in all route advertisements `R3`. In the next iteration `T+1`, the same process repeats itself when triggered by the `activeAS(U,W,T+1)` event.

The above conceptual model is captured using the following PVS definition:

```
bgp(U,W,R0,R3,T): INDUCTIVE bool =
  EXISTS (R1,R2): activeAS(U,W,T) AND
    pt(U,W,R0,R3,T) AND bestRoute(W,T,R0)
```

The peer-transformation `pt` component can be further decomposed into three sub-component (`export`, `pvt`, and `import`). The `export(U,W,R0,R1,T)` component species that AS `W` applies a filter to input route `R0`, which is then advertised to its neighbor `U` as route `R1`. The actual route advertisement is performed by the sub-component `pvt(U,W,R1,R2,T)`, which implements the path vector protocol by propagating the exported route `R1` from `W` to node `U`. The propagated route is received as `R2` at AS `U`. Upon receiving this route, AS `U` applies its import policies via sub-component `import` which transforms `R2` into `R3`.

Again, the series of route transformation is capture by the following PVS definition in a straightforward way:

```
pt(U,W,R0,R3,T): INDUCTIVE bool = export(U,W,R0,R1,T)
  AND pvt(U,W,R1,R2,T) AND import(U,W,R2,R3,T)
```

In a top-down fashion, one can further define the sub-components of `pt`, i.e. the `export` and `import` components which specifies the export and import policies for filtering routes, and the `pvt` component for the actual path-vector protocol itself.

Reference [23] provides details on the PVS specifications and example proofs of various properties BGP, which includes the `Disagree` scenario [8, 7] in the presence of policy conflicts, and generalization of the proofs to an arbitrary large network via induction.

3.2.2 NDlog *Program Generation*

We next describe the translation steps required to generate property preserving *NDlog* programs from component-based network models. We consider an atomic component `t` with a list of inputs `I`, and list of outputs `O`, and any additional constraints and assignments are denoted by `C1(I,O)`, `C2(I,O)`,... For ease of notation, we denote the set of all constraints in the component as `CT(I,O)`. Here, we assume that all inputs `I` are themselves generated from one input component, and all outputs are sent to one output component. The PVS specification of this component is given by:

```
t(I,O):INDUCTIVE bool = CT(I,O)
```

The equivalent *NDlog* rule is as follows:

```
t_out(O) :- t_in(I), CT(I,O)
```

The basic idea is to specify component `t` as a rule that takes as input a `t_in(I)` predicate in the rule body, and derives `t_out(O)` in the rule head, whenever all additional constraints and assignments in `CT` are satisfied.

The above translation works for a single component that is connected with one other input and output component. The translation can however be easily generalized to a component connected to multiple input and output components. In this case, each input component will generate one `t_in(I)` predicate in the rule body, and for each output component, a *NDlog* rule is generated with the corresponding rule head denoting the output.

Given a component with sub-components, one can recursively define the rules in a top-down fashion. Consider the compositional component `tc` shown in Figure 3 that consists of three subcomponents `t1,t2,t3`, each of which has additional constraints `C1`, `C2`, and `C3`. The PVS definitions are as follows:

```
tc(I1,I2,O3): INDUCTIVE bool = EXISTS (O1,O2):
  t1(I1,O1) AND t2(I2,O2) AND t3(O1,O2,O3)

t1(I,O): INDUCTIVE bool = C1(I,O)
t2(I,O): INDUCTIVE bool = C2(I,O)
t3(I,O',O): INDUCTIVE bool = C3(I,I',O)
```

The equivalent *NDlog* rules are as follows:

```
t1_out(O1) :- t1_in(I1), C1(I1,O1).
t2_out(O2) :- t2_in(I2), C2(I2,O2).
t3_out(O3) :- t1_out(O1), t2_out(O2), C3(O1,O2,O3).
```

To annotate the above *NDlog* program with the appropriate location specifiers for each predicate, additional predicate schema information is required as input for the translation process. Reference [23] validates distributed executions of translated *NDlog* programs implementing a path-vector protocol with export and import policies within a local cluster environment, and observe delayed convergence in the presence of policy conflicts.

3.3 Meta-Theoretic Model

Section 3.2 demonstrates that verified *NDlog* programs can be generated from a conceptual BGP model. However, the model itself is not formally specified and checked. To develop complex models in a systematic and compositional way with correctness guarantee, one would like to also formally specify the network model and verify the model directly at design phase.

FVN aims to provide a *meta-theoretic* framework for specifying new formal network models at the design stage. Once verified, it can be used to generate logical specifications for additional verification, and *NDlog* programs for implementation. To illustrate this process, we demonstrate a subset of *FVN*'s built-in network meta-model based on metarouting [9]. Our goal here is to demonstrate the use of *FVN* to define new routing protocols given the built-in meta-model. Detailed description and formalism is available in reference [24].

3.3.1 Background on Metarouting

Metarouting is an algebraic framework for specifying routing protocols in a restricted fashion such that the protocols are guaranteed to converge. Metarouting provides base algebras as the atomic building blocks, together with composition operators that generate complex protocol algebras from existing ones.

Metarouting uses abstract routing algebra as the mathematical model for routing. An abstract routing algebra

A is denoted by a tuple A: $A = \langle \Sigma, \preceq, \mathcal{L}, \oplus, \mathcal{O}, \phi \rangle$. Σ is the set of paths in the network totally ordered by $\preceq$. Intuitively, the preference relation is used by the routing protocol algorithm to select the most desirable path; $\mathcal{L}$ is a set of $labels$ describing links between immediate neighbors. Note that the labels may denote complicated policies associated with the corresponding link; $\oplus$ is a mapping from $\mathcal{L} \times \Sigma$ to Σ, which is the *label application operation* that generates new paths by concatenating existing paths and adjacent links; $\mathcal{O}$ is a subset of Σ called *origination* that represents the initial routes stored at network nodes; Finally ϕ is a special element in Σ denoting prohibited paths that will not be propagated.

The semantics of routing algebra is given by four axioms on *maximality*, *absorption*, *monotonicity*, and *isotonicity*. The maximality and absorption axioms describe the behavior of prohibited path as the least preferred path that is closed under path concatenation; monotonicity imposes the restriction that a path becomes less preferred when it "grows" (i.e. path concatenation occurs), and isotonicity states that the preference relation over two paths is preserved when concatenated with the same link.

Unlike previous combinatorial models [8], the routing algebra identifies and proves that the properties of *monotonicity* and *isotonicity* are sufficient conditions for network convergence. Convergence verification of routing protocols implementation are hence reduced to proofs of monotonicity and isotonicity of the routing algebra.

Based on abstract routing algebra, metarouting further defines a set of base (atomic) algebras and composition operators which serve as the building blocks in the construction of routing algebras. For instance, metarouting provides instances of base algebras for adding link costs (`addA`) during path concatenation, and for specifying local preferences (`lpA`) used in route selection. These algebras are then used by composition operators such as the lexical product operator that models lexicographical comparisons of multiple attributes in route selection.

3.3.2 Example Protocol Formalization

Given the above basic framework, a protocol designer formalizes a routing protocol in terms of the metarouting algebras, and prove that the above four axioms hold for the protocol. This is tedious work: mistakes in the hand-written proofs yield faulty designs, which defeat the purpose of formal modeling. *FVN* instead uses a theorem prover to automatically check that the protocol is correctly formalized.

Our encoding uses a module system called *theory interpretation* in PVS [21]. An analogy of module systems is the use of .h files and .c files in C. We first encode the abstract metarouting algebra as abstract signatures in a PVS theory called `routeAlgebra` (a .h file). The `routeAlgebra` theory contains the type declarations of the abstract algebra, based on the tuple A that we introduced earlier: A: `sig` (Σ), `prefRel` ($\preceq$), `label` ($\mathcal{L}$), `labelApply` ($\oplus$), `org` ($\mathcal{O}$), `prohibitPath` (ϕ). The `routeAlgebra` theory also contains additional definitions for *maximality*, *absorption*, *monotonicity*, and *isotonicity* axioms.

Next, the network designer instantiates an algebra instance as an implementation of the abstract `routeAlgebra`, similar to definition of a .c file). In order for the instantiation to be valid, the designer must carry out the proofs for the above four axioms. Using PVS, network designers are freed from such tedious low-level proof obligations. The proof obligations are automatically discharged for all the base algebras developed in [24].

Furthermore, it is straightforward to encode the composition operators provided in [9]. Again, the proofs that protocols obtained from composing two well-behaved protocols using those composition operations satisfy all the necessary axioms are automatically discharged by PVS's type checker.

To provide a flavor of the formalism, we show the code snippet of a route selection policy used in a BGP system. The route selection is defined in terms of the lexical product composition operator as follows:

```
BGPSystem: THEORY = lexProduct[LP, RC]
```

`lexProduct` is the pre-defined PVS theory that formalize the lexical product composition operation. `BGPSystem` will first compare local preference (`LP`) between two routes. In the event of a tie in the local preference values, the cost of each route will be compared next (`RC`).

The sub-components `LP` and `RC` can be further defined as PVS theories. For example, the `LP` sub-component can be specified as follows to impose route preferences based on a lower local preference value:

```
LP: THEORY =
  routeAlgebra
    {{sig=lpA.SIG, label=lpA.LABEL,
      labelApply(l:lpA.LABEL, s:lpA.SIG)=l,
      prohibitPath=4, prefRel(s1, s2:int) = (s1<=s2)}}
```

The above definition species that `LP` is an instantiation of the abstract theory `routeAlgebra`, and that `LP` inherits from another pre-defined algebra instance `lpA`. In particular `prefRel` ($\preceq$), which is used to specify a total ordering of all routes, specifies a preference for smaller local preference values. Accordingly, sub-component `RC` can be defined in terms of base algebra `addA` in a similar fashion.

By extending PVS specification logic with metarouting theory, *FVN* can leverage PVS's powerful type checker and built-in proof engine to ensure routing model consistency. As a result, the network designer can focus on high-level protocol design (i.e. customize the policies using existing base algebras, such as the sub-components `LP` and `RC` shown above) and the conceptual decomposition of their routing protocols, and shift the low-level details of ensuring consistency of the derived protocol model with respect to metarouting theory to the proof engine.

4. DISCUSSION

To conclude, we outline our ongoing research efforts.

4.1 Network Models and Implementation

FVN uses *NDlog* as an intermediary layer between high-level logical specifications of the network model and low-level implementations. However, it is still up to the network designer to define the conceptual network model. Our initial effort in Section 3.3 demonstrates the promise of using formal tools to automate the process of defining and verifying a network model. We plan to expand upon this initial effort and explore a range of alternative meta-models in *FVN*. For example, metarouting, as an idealized model for a constrained class of protocols, cannot represent well behaved converging protocols that violate monotonicity. *FVN*, by leveraging PVS's proof checker, can aid in the design and analysis of relaxed algebraic models for a wider range of routing protocols.

In Section 3.2, we observe a natural mapping between a component-based conceptual model of BGP and equivalent *NDlog* programs. However, in the case of the metarouting formalism in Section 3.3, the translation is less clearly defined. We however are optimistic that given the close logical relationships between metarouting algebraic objects and declarative networking specifications, a property-preserving translation can be achieved.

Beyond routing, we are interested in exploring network models based on component-based abstractions (e.g. [14, 10]) that are also amenable to the translation into *NDlog* programs for direct execution.

4.2 Modeling Soft-state

Soft-state [4] is central in the design of many network protocol. Declarative networking incorporates soft-state by allowing tuples in tables to timeout after a specified lifetime. To reason about protocols with soft-state, Wang *et al.* [22] utilize a rule rewrite strategy that translate soft-state to hard-state rules via the introduction of explicit timestamps and lifetime attributes to soft-state predicates. The resulting encoding is heavy-weight and cumbersome to prove, and consequently, non-ideal for reasoning about eventual consistency of protocols.

We are currently exploring the use of *linear logic* [6] as the semantics foundation for verification of *NDlog* programs with soft-state data. To this end, we are interested in extending *NDlog* specification with linear logic so that the semantics of soft-states can be explicitly modeled. The characteristics of linear logic is that logical facts will be consumed once it is used in a proof. Consequently, linear logic is known to be able to reason about state transition and resource consumption elegantly, and have wide-applicability in security protocol verification [3] and memory update models [11].

4.3 Combining Verification Techniques

An important aspect of this work is to develop a holistic understanding of various verification techniques, and develop a verification methodology that combines them in the most effective manner.

Unlike model checking, theorem proving is complete, and it is also more expressive in the types of properties that it can express and verify. In terms of proof effort, model checking typically requires less human intervention compared to theorem proving. We however note that typically two-thirds of the proof steps can be automated by the theorem prover's default proof strategies, and in some cases, can be further automated [25].

An advantage of model checking over theorem proving is its ability to simulate runs of the protocols and explore all possible states to detect automatically if some run will reach an undesirable state. The proof process can be automated via integration with model checking to explore the proof search space. Another useful method to combine the two techniques is via a *counter-example* approach. When verification fails, most model checkers provide a counter-example (that is, a trace that illustrates why the formula evaluates to false) to aid in the theorem proving process.

Extending *NDlog* with linear logic has the added benefit that it would allow us to view the declarative networking specification as a set of transition rules that determine the updates of the underlying routing tables. We can leverage such transition system representation to directly interface with model checkers, hence providing an additional verification mechanism for *FVN*.

Acknowledgments This work is partially supported by NSF CAREER CNS-0845552, NSF CCF-0820208, and AFOSR MURI 22179000-41070B.

5. REFERENCES

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
[2] Y. Bertot and P. Castéran. Interactive theorem proving and program development. Coq'Art: The calculus of inductive constructions, 2004.
[3] I. Cervesato, N. Durgin, P. Lincoln, J. Mitchell, and A. Scedrov. A meta-notation for protocol analysis. *CSFW, IEEE*, 1999.
[4] D. D. Clark. The design philosophy of the DARPA internet protocols. In *SIGCOMM*, 1988.
[5] D. Engler and M. Musuvathi. Model-checking large network protocol implementations. In *NSDI*, 2004.
[6] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 1987.
[7] T. Griffin and G. Wilfong. An analysis of BGP convergence properties. In *SIGCOMM*, 1999.
[8] T. G. Griffin, F. B. Shepherd, and G. Wilfong. The stable paths problem and interdomain routing. *IEEE Trans. on Networking*, 10:232–243, 2002.
[9] T. G. Griffin and J. L. Sobrinho. Metarouting. In *ACM SIGCOMM*, 2005.
[10] M. Handley, A. Ghosh, P. Radoslavov, O. Hodson, and E. Kohler. Designing IP Router Software. In *NSDI*, 2005.
[11] L. Jia and D. Walker. ILC: A foundation for automated reasoning about pointer programs. In *ESOP*, 2006.
[12] M. Karsten, S. Keshav, S. Prasad, and M. Beg. An axiomatic basis for communication. In *SIGCOMM*, 2009.
[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] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. *ACM TOCS*, 18(3):263–297, 2000.
[15] B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing Declarative Overlays. In *ACM SOSP*, 2005.
[16] B. T. Loo, J. M. Hellerstein, I. Stoica, and R. Ramakrishnan. Declarative Routing: Extensible Routing with Declarative Queries. In *ACM SIGCOMM*, 2005.
[17] S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. K. Srivas. PVS: Combining specification, proof checking, and model checking. In *CAV*, 1996.
[18] P2: Declarative Networking System. http://p2.cs.berkeley.edu.
[19] J. A. N. Perez, A. Rybalchenko, and A. Singh. Cardinality abstraction for declarative networking applications. In *CAV*, 2009.
[20] 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.
[21] O. Sam and S. Natarajan. Theory Interpretations in PVS. Technical report, 2001.
[22] A. Wang, P. Basu, B. T. Loo, and O. Sokolsky. Declarative network verification. In *PADL*, 2009.
[23] A. Wang, C. Liu, B. T. Loo, O. Sokolsky, and P. Basu. Verifiable Policy-based Routing with DRIVER. Technical Report MS-CIS-09-12, CIS Dept. University of Pennsylvania, 2009.
[24] A. Wang and B. T. Loo. Formalizing Metarouting in PVS. In *Automated Formal Methods (AFM)*, 2009.
[25] Yices: An SMT Solver. http://yices.csl.sri.com/.