We advocate for an approach to network modeling and analysis based on a common intermediate language. Unlike today, where each tool builds a custom model and analysis engine for its target network functionality, we argue that network functionality should be expressed in a common language. This approach makes it easier to expand formal analysis to new functionality and analyze interactions between dependent functionalities (e.g., routing and packet filtering). We demonstrate the feasibility of this approach by developing an intermediate language called Zen and three different analyses for programs in that language. For representative data plane and control plane functionalities, we find that Zen reduces the modeling effort by an order of magnitude, while providing analysis performance that matches custom tools.

CCS CONCEPTS
- Networks → Network reliability; • Theory of computation → Program verification; • Software and its engineering → Model checking; Functional languages.

ABSTRACT

Network verification today. Each tool has its own model of target functionality and its own analysis approach.

Figure 1: Network verification today. Each tool has its own model of target functionality and its own analysis approach.

Figure 2: Network verification with model composition.

However, as shown in Figure 1, each such tool today is a monolith, with its own model of the target functionality and its own analysis engine. The engine typically includes a front end that encodes the domain and a back-end solver that may be custom or standard solvers such as an SMT solver or BDDs (binary decision diagram). Any sharing between tools that exists is surface-level. For instance, Minesweeper [3] and ARC [12] use Batfish [11] to obtain a structured representation of network configuration files, but then build their own models and their own analysis engines. If Batfish is later extended to new functionality, these tools will not support that functionality unless they are updated as well.

This state of affairs makes it hard to expand verification to new network functionality because that expansion requires developing the full pipeline from scratch, a significant endeavor. Consequently, there is a substantial amount of network functionality that is not covered by any tool today, from the link layer (e.g., multiple access protocols) to the network layer (e.g., EIGRP routing protocol) to the application layer (e.g., HTTP firewalls and URL-based forwarding). The extent of unverified network functionality will only increase as cloud network providers continue to roll out new features and as engineers rapidly innovate atop programmable NICs and switches.

Making matters worse, it is not enough to have some tool to verify each piece of network functionality in isolation because the ultimate network behavior depends on the interactions of these pieces. When an individual piece is verified, it assumes that the pieces it depends upon are correct. If tools use disparate models, such assumptions can go unchecked, and bugs can lurk at the boundary of independently verified pieces.
We call for a compositional approach to network modeling and analysis. Illustrated in Figure 2, here, network functionality is modeled in a common language, and analysis engines target the language instead of a specific network functionality. This approach enables rapid expansion of verification to new functionality by separating concerns. Users need only encode domain-specific functionality in the modeling language, while authors of analysis engines need only target the modeling language and not a particular domain. It also enables holistic analysis across functional pieces. One can combine the models of multiple pieces to obtain a model of the joint behavior that can then help uncover bugs at the boundaries. These capabilities will put us on the path to fully verified networks, where all critical functionalities and interactions can be verified.

Our approach is inspired by the software analysis domain. There is a vibrant ecosystem around intermediate languages such as LLVM [25] and Boogie [27] that can encode the semantics of programs in multiple other source languages. A variety of analysis and optimization tools are available for these languages, which benefit all source languages. Boogie also became the basis for writing provably correct programs [26] and full-system verification [18] where each instruction was formally verified. Our intent is to similarly accelerate innovation for network modeling and analysis.

The success of our proposal depends on designing the right intermediate modeling language. It must be expressive enough to encode diverse and complex network functionality, restricted enough to permit efficient automated analysis, and simple enough to reduce the burden of implementing analyses.

We present the preliminary design of such an expressive yet compact modeling language called Zen. At its core, Zen is an expression-oriented language with basic types including booleans, integers, tuples, objects/structs, lists, and maps. To make it easier to both encode network functionality and author analysis tools, we embed Zen in the C# language. Users write functions that process Zen objects as they would write any C# code. We then use reflection to automatically analyze and translate objects into logic or a novel state set abstraction that we have developed.

We built several analysis engines for Zen including a simulator, bounded model checker, unbounded model checker, and test input generator. We further encoded the functionality of a number of network components such as route policies, IP GRE tunnels, and ACLs, and compare their expressiveness and performance with state-of-the-art tools. We find that Zen can often implement complex functionality in an order of magnitude less code and that its analysis is efficient.

2 MOTIVATION

To realize our vision of rapid development of network analyses, we need an intermediate verification language (IVL) that is: (1) compositional, and (2) general.

Objective #1: Compositional. Consider a setting where multiple virtual networks, called overlays, run atop a shared physical network, called the underlay. Such virtualized networks are the norm in modern data centers because they provide isolation and easy migration for overlays. Figure 3 shows one such network. Vₐ and Vₜ (virtual machines or containers) are overlay endpoints with a (virtual) link between them, and the underlay has three nodes.

![Figure 3: An example virtualized network with an illustration of how packets flow across it.](image)

There are many ways to virtualize a network, but independent of the technique used, overlay packets tend to be tunneled. So, as shown in the figure, when Vₐ sends a packet to Vₜ, this packet is encapsulated by U₁ within another header with U₃ as the destination. Then, the packet reaches U₂ via the underlay, and it is decapsulated and passed to Vₜ. The overlay and underlay have their own control and data plane systems. That is, the overlay has its own routing, forwarding, and packet filtering rules, and the underlay has its own version, though the underlay processing may be based on overlay headers as well. The two systems may be completely different, e.g., the overlay may use an SDN-style control plane and the underlay may use distributed routing protocols.

When verifying a virtualized network, it is desirable to verify the combined impact of overlay and underlay processing. Using today’s approach, of building monolithic analysis tools that do not decouple network behavior modeling from solvers, a virtualized network may be verified using one of two ways. The first is to separately verify the overlay and the underlay, using tools appropriate for each network type. Here, overlay verification must assume that the underlay provides perfect connectivity, and the underlay verification will be agnostic to overlays that run atop it. The second method is to build and validate the a combined model of the overlay and underlay.

Both methods are problematic. The first cannot find problems that manifest when the two networks interact. For instance, the underlay may have a buggy packet filter that drops some types of overlay packets. This bug will not be found if we verify the underlay and the overlay separately. The second method has high engineering complexity, as it will have to model multiple types of overlay and underlay combinations.

Both methods are also hard to evolve because the solvers are intimately tied to the network model. Assume that a user introduces new functionality into the overlay (which is easy because developers can roll their own using a software update) or the underlay (which is also easy given the advent of programmable NICs, switches, and P4). Now, we will have to painstakingly update the models as well as the solvers.

Contrast the current approach with a compositional modeling approach. Here, the overlay and the underlay will be modeled separately. Once that is done, a range of solvers become available to verify the underlay and the overlay. It will also enable the creation of different types of combined models, which can then be verified using any available solver.

Our approach will find bugs at the intersection of the overlay and the underlay and will find overlay-only and underlay-only bugs faster. It will also be easier to evolve. Supporting new overlay
We now describe how to model networks in Zen using the virtual set work verification analyses often manipulate. The second is a \( \text{function} \) that takes three parameters. The first is the forwarding encode packet forwarding, one might write the metrics by writing functions that process these objects. For example, to create ordinary classes in the host language (C#) that model objects examples for IPv4 packets (line 1) and for packets with two headers, such as packets, forwarding tables, and GRE tunnels. Figure 4 shows create ordinary classes in the host language (C#) that model objects

<table>
<thead>
<tr>
<th>Analysis</th>
<th>Rosette</th>
<th>Kaplan</th>
<th>Boogie</th>
<th>NV</th>
<th>Zen</th>
</tr>
</thead>
<tbody>
<tr>
<td>HSA [22]</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>AP [41, 42]</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>Anteater [29]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>Minesweeper [3]</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Bonsai [4]</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Shapeshifter [5]</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 1: Whether different intermediate verification languages (IVLs) can express example network analyses.

functionality only requires changing the overlay model, and the rest of the system stays as is.

While we considered virtualized networks above, the essential characteristics we discussed are universal. The full network behavior is a result of the interaction between multiple pieces (e.g., BGP plus OSPF plus access control lists), different networks have different combinations of these pieces, and existing pieces evolve and new pieces appear constantly.

**Objective #2: General.** Our IVL must be able to express a wide range of network analyses. This is where other IVLs for general-purpose software such as Kaplan, Boogie, and Rosette [24, 27, 37] fall short. They cannot express many verification analyses that have been highly effective for networks. A key reason is that network verification analyses often manipulate sets of objects (e.g., packets) whereas these IVLs work via compilation to logical constraints (which help find a counter example). There is a recent IVL for networks, called NV [13]. It too cannot express a wide range of analyses. Instead, NV bundles a few types of analyses, and modifications are needed to support more types. Table 1 shows examples of network analyses and whether different IVLs can support them. Zen achieves generality via \( i \) a new state set abstraction for reasoning about sets of objects; and \( ii \) its embedding as a library in a general purpose language, which allows users to express analyses by manipulating sets of objects using arbitrarily complex code.

### 3 MODELING NETWORKS WITH ZEN

We now describe how to model networks in Zen using the virtual network example above and then demonstrate how to analyze such models in the next section.

**Encoding a Domain.** Assume that we want to build a data plane model for a network with longest-prefix match based forwarding and access control (ACLs) in both the overlay and underlay, and IP GRE [17] for tunneling. This task can be split into two parts. We first create ordinary classes in the host language (C#) that model objects such as packets, forwarding tables, and GRE tunnels. Figure 4 shows examples for IPv4 packets (line 1) and for packets with two headers, an overlay header and an optional underlay header (line 9).

After defining the objects to model, we encode the domain semantics by writing functions that process these objects. For example, to encode packet forwarding, one might write the `Forward` function (line 12) that takes three parameters. The first is the forwarding table in which the entries are in descending order of prefix length. The second is a `Zen<Header>`, which is an IP header. More generally, the wrapper type `Zen<T>` represents a value of type `T` that is handled by the Zen library and can be either symbolic or concrete. The third parameter is the line number to start matching from.

The function evaluates the header against the forwarding table and returns a `Zen<byte>` representing the output port. It first checks if the line number is beyond the last rule in the forwarding table. If so, it returns 0 (null interface). Otherwise, it gets the current rule and makes a call through Zen to either return the rule’s port number if the rule matches the header, otherwise to continue on to rule \( i + 1 \). Note that the recursive call takes place through C# and not the Zen library.

Matching the header against the rule is similarly implemented as a simple function that checks if the header’s destination IP is matched by the forwarding rule prefix. The library overloads operators such as \&\& and \& to work seamlessly over Zen values. We can encode ACLs in a similar manner as forwarding tables since they too are a list of prioritized rules.

A final step is to encode the semantics of tunneling, whose implementation is shown in Figure 5. There are two additional functions to define the effect of encapsulating and decapsulating a packet given a GRE tunnel. The first function `Encap` adds an underlay header using the tunnel’s source and destination IP addresses while copying over all other fields. The second function `Decap` simply strips off the top header by replacing it with `Null<Header>()`. The function `Encap` adds an underlay header with a specified destination IP address, copying over all other fields. The second function `Decap` simply strips off the top header by replacing it with `Null<Header>()`.

**Composing Network Models.** Composing models of network elements with Zen is as simple as writing new functions that call functions defined in earlier models. Suppose we want to model combined (overlay and underlay) treatment of packets at a switch, accounting for forwarding, ACLs, and tunneling. We might write the functions in Figure 6. The two functions take a packet as input along with an interface, and return a value of type `Zen<Option<Packet>>`.

```csharp
public class Header {
    public Ip SrcIp;
    public Ip DstIp;
    ....
}
public class Packet {
    public Header OverlayHeader;
    public Option<Header> UnderlayHeader;
}

Zen<byte> Forward(FwdTbl t, Zen<Header> h, int i) {
    if (i >= t.Rules.Length)
        return 0; // null interface
    var r = t.Rules[i];
    return If(Matches(r, h), r.Port, Forward(h, i+1));
}

Zen<byte> Matches(FwdRule r, Zen<Header> h) {
    var mask = 0xFFFFFFFF << (32 - r.Prefix.Length);
    return (h.GetDstIp() & mask) == r.Prefix.Address;
}
```
as output, which is either null if the packet is dropped, or otherwise a new (possibly modified) packet. The first function applies any inbound policy including the ACL and decapsulation, while the second applies outbound policy, including the forwarding table, outbound acl, and any encapsulation.

4 ANALYZING MODELS WITH ZEN

Zen provides a number of ways to analyze network models.

Simulation. Since Zen models are executable—they are simply C# code—simulations performed by tools like Batfish [11] are straightforward. In particular Zen allows users to pass concrete values of type T to arguments expecting a value of type Zen<T>. For example, to simulate what happens to a given packet entering the network at a given interface, starting with that packet and interface as concrete inputs, we will repeatedly call FwdIn and FwdOut functions until the packet is dropped or exits the network (along all paths).

Finding (counter) example inputs. Many verification tasks are based on finding an input that leads to an undesirable behavior. Zen enables this primitive using its Find method. Suppose we wanted to know if a flow will be delivered along a path. We can write a function such as the one in Figure 7 to capture how a flow traverses a path through the network. Zen can then reason about it:

```csharp
var f = Function(pkt => Fwd(path, pkt));
var results = f.Find((pkt, result) => result.HasValue);
```

The first line creates a ZenFunction that the library can manipulate, and the second line asks for a packet that is delivered along the path. Packet delivery is indicated by checking that the result of f should have a value. Under the covers, Zen can leverage various forms of symbolic reasoning to find an example (if any) input.

To find if a packet can reach node A to B, along any path, we can iterate over all possible paths between those two nodes. If the execution of Find uses SMT-based reasoning, we would have implemented a verifier akin to Anteater [29], though Zen is not limited to just that reasoning method alone.

Computing with sets. Many network analyses reason about sets of objects [4, 5, 22, 41] instead of finding examples. Zen enables such reasoning via transformers that can manipulate large sets of objects. For example, we can construct a transformer for the FwdOut function for an interface i:

```csharp
var t = f.Transformer();
```

### Figure 5: Encoding IP GRE tunnels in Zen

```csharp
Zen<Packet> Encap(GreTunnel t, Zen<Packet> pkt) {
    if (t == null) return pkt;
    var oheader = pkt.OverlayHeader;
    var uheader = Create<Header>
        (Create<Ip>(t.SrcIp), Create<Ip>(t.DstIp),
            oheader.GetProtocol(), oheader.GetDstPort(),
            oheader.GetSrcPort(),
            oheader.GetProtocol());
    return Create<Packet>(oheader, Some(uheader));
}
```

### Figure 6: Modeling the combined (overlay and underlay) treatment of packets being processed at a device.

```csharp
Zen<Packet> Decap(GreTunnel t, Zen<Packet> pkt) {
    if (t == null) return pkt;
    return Create<Packet>(
        pkt.OverlayHeader, Null<Header>());
}
```

### Figure 7: Modeling forwarding along a given path.

```csharp
IEnumerable<PathSet>
HSA(Intf i, StateSet<Packet> set) {
    var q = new Queue<PathSet>();
    q.Enqueue(new PathSet(i, set));
    while (!qIsEmpty()) {
        var path = q.Dequeue();
        var forwarded = false;
        foreach (var intfOut in intfIn.Device.Nbrs) {
            var outSet = tout.TransformForward(inSet);
            var port = Forward(i.Device, p, 0);
            var encap = Encap(i.GreStart, p);
            var pktOut = If(port == i.Id, pktOut, Null<Packet>());
        }
    }
    return x;
}
```
Once a user creates a transformer, Zen can automatically compute the TransformForward set that represents the set of output objects that correspond to the given input objects and TransformsReverse set that represents the set of input objects that correspond to the given output objects. These capabilities enable users to build efficient analyses without worrying about the implementation.

Figure 8 shows an implementation of HSA [22], which computes sets of reachable packets from an interface along all paths. It uses the inbound and outbound transformers for network interfaces (built using FwdIn and FwdOut functions respectively; not shown in the figure) and pushes packet sets through the network to explore all paths. TransformForward computes the packet sets at each step.

5 LANGUAGE

The Zen language is designed to be as simple as possible without a priori limiting what users can encode. At its core it is a simple expression language, whose abstract syntax (e) is shown in Figure 9. It supports most logical, arithmetic, and bitwise operations, as well as ways to create objects, get and update their fields, and match on lists (case).

To make Zen extensible, we include a special expression type: adapt[r1, r2](e) that allows for implementing operations over new types by converting them to types that Zen knows how to handle. For instance, Zen currently implements dictionaries by representing them as lists of tuples with the most recent elements at the head of the list, and it implements options by representing them as a class with flag and value fields.

6 IMPLEMENTATION

The Zen framework is implemented in over 15K lines of C# code and available as open source software.1 Zen currently supports several backends. One is for bounded model checking that can use either an SMT (via Z3 [8]) solver or a high-performance binary decision diagram (BDD) solver. For the SMT backend, Zen encodes all primitive operations using the theory of bitvectors before bitblasting [16] the formulas to SAT. Another backend uses the transformer API to perform unbounded model checking and also leverages the BDD backend. Transform operations such as TransformForward are implemented using standard pre/post image computation via existential quantification [7]. All the backends in Zen use the reflection capabilities of C# to introspect the types of objects at runtime, and thus build efficient symbolic representations.

1https://github.com/microsoft/zen

Variable ordering heuristics. For the BDD backend, Zen uses a custom analysis, similar to alias analyses in traditional programming languages, to find a strategy for ordering variables. BDDs can often perform very well but are highly sensitive to the order of their variables [33]. For example, when two variables are compared for (inequality, Zen ensures their orderings will be interleaved, as any other ordering will result in an exponential memory blowup [1]. For instance, in the following function:

\[ \text{Function} \langle \text{int}, \text{int}, \text{int}, \text{bool} \rangle ((x, y) \Rightarrow x\text{.Item2 == y}) \]

the second component for x must be interleaved with y.

If Zen detects that different transformers have different variable ordering requirements, it performs a second optimization whereby it allocates a new set of unique variables for the second transformer rather than reusing those for the first. Instead, it converts between the sets of variables dynamically at runtime using a BDD substitution operation. When possible to efficiently order the variables, it avoids this runtime conversion. Moving this translation to runtime in many cases allows for implementing transformers that would otherwise be impossible due to state space explosion.

Composite data structures. To implement complex data structures like lists, Zen uses a variable to represent the list length and another collection of variables to represent the list elements for different sized lengths. It then employs a type-driven merging operation similar to that employed by Rosetta [38]. The maximum list length is controlled via an optional parameter to the FInd function.

7 PRELIMINARY EVALUATION

We present results from preliminary experiments that show the feasibility and promise of our language-based approach. They show that expressing a range of network functionality is easy and the performance overhead of a general solver that is not functionality specific is acceptable.

Expressiveness. To demonstrate Zen’s expressiveness, we write implementations for several networking components such as router ACLs and longest-prefix-based forwarding, as well as control plane route policies offered from commercial vendors and more. For each implementation, we report on the number of lines of code required to model the component using Zen. Table 2 shows the results. In general, we find that the implementations are straightforward and easy to express. Moreover, they compare favorably with existing implementations. For example, Minesweeper [3] implements a similar conversion of route-maps to SMT using 1K lines of code, and Bonsai [4] implements a similar conversion with BDDs using over 1K lines. We do note that our implementation is not 1:1 feature compatible with Minesweeper: we implement certain features such as the full AS path, which Minesweeper does not, and do not implement certain features such as OSPF areas that Minesweeper does.

However, the Zen encoding gives both a BDD and SMT backend in the same 75 lines of code.

Performance. We evaluate Zen’s performance on two verification tasks. The first is the time to verify an ACL (a data plane analysis), and the second is the time to verify a route map (a control plane analysis). In both cases, the verifier’s task is to find inputs (data packets or routing messages) that match the last line, which requires analyzing the complete ACL or route map. We generated ACLs and

\[ \text{e ::= c | e1 < e2 | e1 * e2 | e1 - e2 | e1 == e2 | e1 & e2 | (e1 | e2) | not e | e1 or e2 | e1 and e2 | create[r](e, \ldots, e) | e.f e1[f := e2] | if e1 then e2 else e3 | [] | e1 :: e2 | case e1 of e2 e3 | adapt[r1, r2](e) \]

\[ \text{r ::= bool | byte | short | ushort | int | uint | long | ulong | (r1, r2) | [f=r1, \ldots, f=r2] | List[r] | Option[r] \]

Figure 9: Zen abstract language syntax.
While our initial focus with Zen is analyzing network models, it has which we have already prototyped.

<table>
<thead>
<tr>
<th>Network Component</th>
<th>Zen Lines</th>
<th>Existing systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>LPM-based Forwarding</td>
<td>18</td>
<td>&gt;900 [22]</td>
</tr>
<tr>
<td>Route Map Filters</td>
<td>75</td>
<td>&gt;1000 [3, 4]</td>
</tr>
<tr>
<td>IP GRE tunnels</td>
<td>21</td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Lines of code to express common network functionalities in Zen. The third column shows lines of code for encoding similar functionality in current tools.

Figure 10: Zen microbenchmarks for random ACLs and route maps with line tracking and different solvers.

route maps of different sizes randomly, and we ran both BDD and SMT backends. For the ACL analysis, we also ran Batfish, which performs the same analysis using a hand-optimized, BDD-based encoding. Batfish currently does not support verification of route maps. All experiments were performed on a 8-core Intel i7 machine with 16GB of RAM, and each data point in the graphs is the mean value across 100 runs.

Figure 10 shows the results. For the ACL analysis, we see that Zen’s BDD backend is more efficient than the SMT backend. We also see that this backend performs comparably to the hand-optimized Batfish implementation despite having its encoding generated automatically. Thus, general solvers have the potential to match the performance of custom ones.

For the route map analysis, unlike ACL analysis, we see that the SMT backend performs better than the BDD backend. In general, we have found the SMT backend better for reasoning about data structures such as lists. These results show the value of having access to multiple backends, so users can pick the one that is best for their domain and network. This goal would be almost impossible to achieve with custom encodings as one would have to develop multiple different backends for each functionality.

8 BEYOND MODEL ANALYSIS

While our initial focus with Zen is analyzing network models, it has other important use cases. We briefly discuss two such use cases which we have already prototyped.

Testing implementations. Zen models can become the basis for testing the implementations that they model. Given a Zen function `f.GenerateInputs()` produces test inputs with a high-degree of coverage based on symbolic execution [14]. We can test that these inputs are handled by the implementation as expected. For instance, if we have a model for an ACL, we can generate test packets that match on every single rule in the ACL, and then validate that the implementation processes each packet as expected. This model-based testing approach has been successfully used before [30, 39]. Zen enables a more modular and systematic way to expand such testing to a broad set of network functionalities.

Synthesizing implementations. Zen models are executable, allowing us to directly generate implementations from them. We can compile any Zen function to a real implementation by simply writing: `f.Compile()`. This instructs Zen to generate C# IL instructions, using the System.Reflection.Emit API, which will then be just-in-time compiled to assembly at runtime. The resulting implementation runs efficiently. An implementation extracted in this manner will be in sync with the verified model. This property then becomes the foundation for networks whose implementations are provably correct (modulo compiler bugs).

9 RELATED WORK

Zen builds on two prior threads of research:

Network verification. There has been a long line of research on network verification. These works differ in terms of verification algorithms used as well as the network functionality targeted: stateless dataplanes [19, 21–23, 29, 40, 41], stateful dataplanes (e.g., middleboxes) [32, 43], programmable dataplanes (e.g., Click, P4) [9, 28, 35], distributed routing protocols (e.g., BGP, OSPF) [3–5, 10–12], and centralized control planes [6, 15]. While different domains come with their own challenges, they commonly employ translations to standard verification technologies. Zen aims to abstract away this translation. Even tools that use non-standard or domain-optimized solvers (e.g., HSA [22]) can incorporate such solvers as new backends in Zen, allowing for many models to reap their benefits. Beyond simplifying tool development, Zen also allows for easy composition of network models which is challenging or impossible when different tools are implemented using disparate technologies and APIs.

Intermediate verification languages. Zen draws on prior work on IVLs [24, 27, 34, 37] that aim to simplify verification tasks. It shares many common technologies with these languages. For example, its bounded model checker uses a type-aware merging strategy pioneered by Rosette [37]. However, as shown in Table 1, prior IVLs cannot express many common network analyses. To address this limitation, Zen introduces a new state set abstraction that allows for directly manipulating sets of values in user code.

Zen shares the linguistic modeling approach of NV [13]. While NV provides high-level abstractions for encoding certain network functionalities (e.g., distributed routing) and analyses, Zen’s abstractions are lower-level and more general. Consequently, it can be used to model a wide range of network functionalities and analyses.

10 CONCLUSION

Verification tools today are implemented as monoliths, mixing together domain semantics, analysis engines and solver technologies. We argue for a compositional approach to network modeling and analysis based on a common intermediate language for expressing domain functionality. This approach can enable rapid construction, composition, and verification of domain-specific models for new network functionalities and pave the way for fully-verified networks.