VeriLucid: A Verification-aware Data-plane Programming Language

John Sonchack
Princeton University
Princeton, NJ, USA
jsonch@princeton.edu

Pamela Zave
Princeton University
Princeton, NJ, USA
pamela@pamelazave.com

Jennifer Rexford
Princeton University
Princeton, NJ, USA
jrex@princeton.edu

Abstract

Correctness is important in data-plane programs, which run on critical infrastructure connecting millions of users. Verification helps programmers build correct software, but current data-plane tools can only check simple properties or require immense programmer effort. As a solution, this paper introduces the first verification-aware data-plane language: VeriLucid. The core idea is to unify programming and specification in one high-level language, with built-in proof automation. Integration makes it natural for programmers to use verification continuously throughout development, like unit testing but with strong guarantees. In evaluation, we show that VeriLucid requires 10X less programmer effort, in terms of lines of code, than other verification tools with comparable expressiveness.

CCS Concepts

• **Networks** → **Programmable networks**; **Data path algorithms**; **Programming interfaces**; • **Software and its engineering** → **Domain specific languages**; **Formal software verification**.

Keywords

data-plane programming, verification

ACM Reference Format:
John Sonchack, Pamela Zave, and Jennifer Rexford. 2026. VeriLucid: A Verification-aware Data-plane Programming Language. In *ACM SIGCOMM 2026 Conference (SIGCOMM '26), August 17–21, 2026, Denver, CO, USA.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3789240.3829171

1 Introduction

Data-plane programming is important but challenging. Customizing a network's packet forwarding and processing has many benefits, for example a 100X improvement in the scalability of DDoS defense by offloading from an end host to a programmable switch [40]. But data-plane programs are difficult to write, and difficult to get right. The difficulties of complex hardware constraints, low-level languages, and aggressive optimization vex data-plane programmers. Bugs inevitably arise, and their cost in real networks is high—unnecessary failures and uncertain behavior, in critical components for networks serving thousands or millions of endpoints.

The only known comprehensive solution is verification that a program satisfies a high-level specification—one that clearly relates the program to the networking problem that it solves. Verification is most useful when it supports a data-plane programmer's favorite strategy, which is to start with a readable reference implementation, then gradually refine it to satisfy hardware constraints. When applied continuously throughout refinement,verification can help programmers work through design problems, catch bugs early when they are easiest to fix, and make changes and optimizations with confidence. Unfortunately, today's data-plane programmers cannot follow this method because the necessary tools do not exist, particularly for the high-performance platforms that are hardest to program. There are three critical unmet needs:

Specifications as expressive as code. Many specification languages for data-plane programs are limited to simple program properties (e.g., unparsed header fields are never read [20], or transport-layer headers remain unchanged by switch processing [34]). Yet, reasoning about the correctness of a data-plane program requires reasoning about the traffic it processes, about the switch that runs it, about the data structures it uses, and about its behavior over time (e.g., a programmer may wish to prove that a sliding-window Bloom filter never returns a false negative). Specifying these properties requires no less expressiveness than implementing them does. Other specification languages used for data-plane programs [3, 36], such as Rocq [33], have the expressive power, but are designed for specialists in verification. Using them requires learning a whole new language, plus writing intricate encodings of data-plane concepts into general-purpose verifiable representations.

High-level languages. The most widely used language for data-plane programming is P4 [5, 7]. There are many tools for verifying basic properties of P4 programs [12, 13, 20, 32, 34, 42, 43] because P4 is a low-level language where simple safety violations, such as reading from uninitialized variables, are common hazards. A higher-level data-plane language [14, 25, 30], with more built-in structure and modularity would make verification of these basic properties unnecessary, and also allow programmers to write far less code.

Continuous, mostly-automated verification. Manual verification is tedious and difficult, but modern verification tools are far advanced along the path to automating it. Almost always, when the tool cannot find a proof of something true, it just needs more helpful facts documented as specifications. The benefits of such tools are enhanced further when they are active participants in development—running continuously in the background, checking that code and specifications agree, and providing source-level feedback when they do not.

In working toward meeting these needs, we can build on the prior art of Lucid [21, 30], a higher-level language that compiles to optimized low-level P4 for the Intel Tofino and Tofino II switches, and handles automatically the allocation of program storage and

SIGCOMM '26, Denver, CO, USA

ACM ISBN 979-8-4007-2467-1/2026/08
https://doi.org/10.1145/3789240.3829171

computation to pipeline stages. Yet Lucid still falls far short of our vision, for two major reasons: (i) there are still many challenges in data-plane programming that Lucid does not eliminate, and (ii) contrary to our vision for data-plane programming, there is no support for verification of Lucid programs.

Verification is not easy, either to build into tools or to use. Decades ago, researchers found that the best way to offer verification is a language that builds it in from the ground up. Verification-aware languages like Dafny [23], carefully designed to support programming, specification, and proof automation, evolved from the problems of trying to layer verification atop languages like C and Java [8]. Dafny is widely used and has a production-ready verifier; it was built over many years and has been battle-tested in large real-world systems [8, 15]. Mindful of this history, we decided to leverage all the work that has been put into Dafny, rather than attempt to extend Lucid with a custom verifier.

In this paper we report on our progress toward realizing our vision of expressive specifications, high-level programming, and continuous, mostly-automated verification—using the basic approach of embedding Lucid in Dafny. The result is VeriLucid (short for Verifiable Lucid), a domain-specific dialect of Dafny supported by a programming framework with verified libraries and a hardware-simulation environment. There are overviews of the language, libraries, and hardware environment in §3 through §5, respectively.

Despite these strong foundations, we had to overcome many engineering and research challenges to bring the idea to fruition. The engineering challenges are everywhere: to preserve and improve the usability of Lucid and Dafny in our context, including many design decisions affecting expressiveness and efficiency as well as usability. In addition, our research solved three specific problems:

- We determined how much of the expressiveness of object-oriented programming in Dafny can be preserved in a data-plane language.
- Some of the stringent data-access constraints inherent in the PISA chip architecture (see §2.1) are independent of the chip resources. For checking those constraints, we designed and automated methods that can be used throughout refinement of VeriLucid programs. These constraints guarantee that stateful programs can be compiled safely to a PISA pipeline with sufficient resources.
- We laid the fundamental groundwork for specifying and verifying timing behavior in a program's operating environment and algorithms. This involved processor modeling, analysis of typical requirements and the reasoning needed to verify them, and discovery of a new constraint on the design of data-plane hardware.

The first two contributions are presented in §2 through §4. The third contribution is presented in §5.

VeriLucid has been evaluated (§6) with a suite of diverse applications. Two case studies, in particular, demonstrate its benefits.

- We verify a core property of a sliding-window Bloom filter. This moderately complicated task takes thousands of lines of verification code in Verifiable P4 [36], the only other tool expressive enough for the task. We show that it takes $\frac{1}{10}$ as many lines of code in VeriLucid, with *no* user-written proofs.

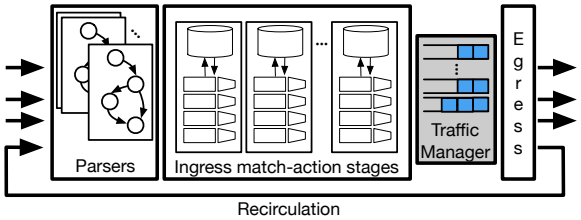

Figure 1: Stateful PISA architecture.

- We use VeriLucid to check the correctness of TurboFlow [29]. Like many recent data-plane systems [29, 31, 40], TurboFlow has a hybrid architecture splitting work between a hardware frontend and a software backend, to balance performance and flexibility. We proved that the published implementation of TurboFlow is incorrect, and have designed an efficient fix.

Our evaluation demonstrates the expressiveness and practicality of VeriLucid. §7 discusses related work, while §8 summarizes limitations and future work.

2 VeriLucid Overview

2.1 Abstracting PISA

VeriLucid targets PISA (Protocol Independent Switch Architecture) [1, 6, 28] processors because they support a wide range of applications with strong performance guarantees. Supporting many applications means supporting *stateful* programming, in which packet histories are not merely stored in passive data structures, but also affect subsequent packet processing. Strong performance guarantees mean line-rate processing at one packet per clock, with operation at 1+ Ghz (i.e., billions of packets per second) in commercially available switches [1].

Figure 1 shows the relevant architectural details. Packets from the network enter an array of parsers, with each parser serving a fixed subset of ports. Parsers extract a packet's headers into memory and submit them to a shared header-processing pipeline. The pipeline has a throughput of one packet per clock. In the pipeline, packets only move forward. Each stage contains parallel exact and ternary match engines, ALUs, O(1MB) of local memory, and a small number of stateful ALUs (sALUs) to operate on the memory. An sALU can perform one atomic read-modify-write operation to a single memory address once per packet. After the pipeline, a traffic manager copies packets to per-port egress queues. A recirculation port carries packets back to the ingress, for processing too complex for a single pass.

Higher-level languages and compilers have done much to make PISA programming more like conventional programming. P4 introduces constructs such as parsers and externally-configured match-action tables. Domino [28] and Lyra [14] invent portable abstractions for stateful operations. Lucid integrates all of these ideas into a language with a type-safe module system, reusable data structures, and an execution model that abstracts a PISA switch as an event processor [21, 22, 30]. In Lucid, parsing converts packets into events that automatically invoke handler functions. This abstraction unifies packet processing, recirculation operations, and communication between distributed components, and allows the compiler to automatically configure the parser, traffic manager, and egress pipeline. Higher-level abstraction also allows Lucid to be portable, e.g., there is also an interpreter [16] and C compiler.

2.2 Program Design and Verification

Even with all this help, PISA programming is still challenging because *the architectural restrictions complicate program design*. For example, consider a stateful firewall tracking IP flows using a set data structure, and needing to perform either a `set.insert(key)` or `set.query(key)` operation on every packet. The PISA architecture imposes many constraints here:

(1) Both operations need to guarantee worst-case constant-time operation, because the pipeline has a fixed number of stages.
(2) The set's internal state needs to be partitioned into multiple smaller data structures, because each sALU can only access a single word while processing each packet, and can only perform limited computation on it.
(3) All control flows, handling all events, need to access the data from each stage in the same order, because packets only move forward in the pipeline.
(4) Finally, in many scenarios, the data structure will not be able to track full flow keys, because there is not enough memory in the pipeline.

These restrictions are well worth the problems they cause, because they make it possible to write stateful programs and execute them at line rate. Because handling of each packet accesses the data from each stage in the same order, program semantics is the same as if handling of each packet was atomic, even though it is actually highly parallelized.

Nevertheless, satisfying these constraints can turn a conceptually simple program into a convoluted puzzle. It may be necessary to rewrite the algorithms so that variables are accessed in a radically different order. If this is not possible, it may be necessary to break handling of some events into two independent pieces, where the second piece is executed by handling a new type of recirculation event. Or it may even be necessary to weaken requirements. For example, going back to the `set` data structure, there may be no way to implement an exact set that satisfies all of these constraints. Then the program must have an *approximate* set, such as a Bloom filter or a cache. Although Bloom filters, caches, and exact sets all have the same interface (i.e., `insert(key)` and `query(key)` operations), they all have different behaviors (i.e., specifications). Bloom filters have false positives, caches have false negatives, while exact sets have neither error.

In all these cases, the programmer will have to reason carefully about what properties the application requires, and how the program ensures that they hold. This is the purpose of verification. But even before that the programmer must know whether the four PISA constraints above are satisfied, to know whether redesign is necessary.

Lucid checks all four constraints at compile time, and in fact (1) and (4) are resource-dependent, and can *only* be checked during Lucid compilation. (2) and (3) have a big effect on program structure, and should be checked early in development, but Lucid's checks depend on hard syntax and type-system rules in a lower-level program—meaning that a Lucid program must already be designed for PISA before it can be checked. In this paper we show how VeriLucid programs can be checked for compliance with (2) (§4.1) and (3) (§4.2) even at early stages of refinement.

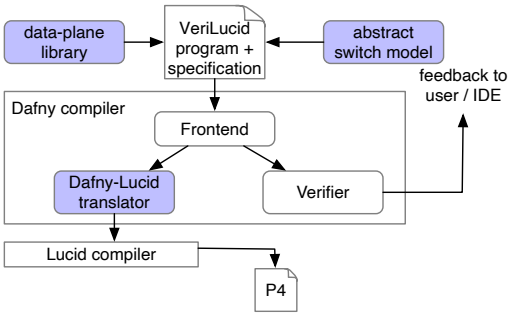

Figure 2: VeriLucid (in blue) extends Dafny and Lucid.

2.3 The VeriLucid Programming Framework

VeriLucid is a verification-aware data-plane language that helps with the kinds of design choices and optimizations described above. It lets programmers write high-level code for both implementation *and* specification, in a single unified language. As the programmer codes, a verifier checks the program against its specification. If it cannot automatically prove that they agree, it gives feedback directly to the programmer, like a type checker. Prior data-plane verification tools fall far short of this. For example, the only prior work we found that supports data structure invariants like false negatives in an approximate set is Verifiable P4 [36], which requires detailed user-written proofs (§6.1).

VeriLucid is implemented as a domain-specific dialect of Dafny that translates to Lucid. Figure 2 shows the components that implement VeriLucid's programming framework, and how they interact with Dafny and Lucid tools.

As the figure shows, a VeriLucid program (with specification) is fed to the Dafny compiler for frontend processing and verification. A verified program then passes through the Dafny-Lucid translator, which has the job of enforcing the restrictions on Dafny making VeriLucid compilable to hardware. §3 explains the challenges of choosing the right subset of Dafny. For ease of programming, we wanted to retain popular features such as objects, if possible. We needed to choose data representations for efficient verification as well as runtime efficiency. And although we cannot choose for the programmer how to satisfy hardware constraints, we must be able to check that the constraints have been satisfied.

2.4 Library-based Memory Models

Stateful operations are the most complex part of a data-plane program. Prior data-plane verification tools use simple memory models that are hard-coded into the verifier. For instance, rather than modeling the instruction-level capabilities of sALUs for real [1] or proposed [28] architectures, prior tools only support read and write operations [20], or have no capability to reason about memory at all [13, 20]. Equally important, prior work has no way to reason about state modifications that occur on recirculation, which enable much more sophisticated programs [30]. So the VeriLucid design must meet three challenges: making it easy for programmers to control precisely how they will meet PISA constraints on memory access, making it possible for the translator to enforce PISA constraints, and enabling effective reasoning about memory.

All this is accomplished with a data-plane library in Dafny as shown in Figure 2 and presented in §4. The library models memory

operations and includes a verified specification of their complete semantics. Additional library helpers let programmers reason about whether their memory operations are compatible with the instruction set of a particular PISA platform. Programmers can also define and verify their own data structure on top of the primitives, such as a Bloom filter.

2.5 Modeling Switch Execution

Data-plane programmers often need to reason about program behavior over time, as a sequence of packets is processed. State-of-the-art data-plane verification tools [43] only support coarse-grained temporal properties expressed in a simple first-order language. They encode temporal semantics by expanding the program into a graph of its state space, which inevitably has limitations on scale.

Instead, VeriLucid's programming framework includes an abstract model of its runtime environment, which is the nonprogrammable part of the PISA processor (Figure 2, §5). This "switch model" includes its internal clock, input and recirculation queues, output queues, and its dispatch algorithm. Library methods simulate processor actions and external events such as packet arrival and the passage of time. A verified specification of the switch model supports precise, flexible reasoning about the timing properties of a VeriLucid program—and the reasoning is unaffected by scale.

As §5 explains, developing this part of our programming framework entailed particular research challenges, including some struggle with the features of Dafny, and a deep dive into processor behavior resulting in a strong recommendation about how programmable processors should be designed.

Overall, the VeriLucid framework surrounds the programming activity with guardrails in the form of templates as well as powerful libraries. Although mostly out of the scope of this paper, these guardrails are especially important for VeriLucid usability because verification is a subtle activity and can be prone to pitfalls and fragile behavior. The more a programmer can rely on the guardrails of a well-worn framework, the less frustration there will be.

3 Embedding Lucid in Dafny

Like Lucid, VeriLucid is event-driven: input packets and other external stimuli are defined as events, and handler functions both process and generate events. VeriLucid represents events and handlers in a carefully selected subset of Dafny, to balance programmer convenience, verifier friendliness, and hardware compatibility.

3.1 Events

An event is usually an abstraction of a packet. Each event type has a name and typed parameters representing the data it carries. Event types are declared as constructors of a single algebraic datatype (i.e., a tagged union). For example, an IP router might use a `Pkt` event to represent data-plane packets, and a `PortStatus` event to represent messages from the control plane about the state of a particular port, for fault-tolerance:

```
datatype Event = Pkt(ip:iphdr, pl:payload)
               | PortStatus(port:nat8, up:bool)
```

Programs can read event parameters like any other data type, and can create new event values from their constructors. Event values are immutable, which simplifies verification. Unifying packets and other external stimuli in a single `Event` type allows us to model switch internals as generic components that operate on any queued event.

3.2 Handlers

Each event type has exactly one handler. Structuring a program as a collection of handlers, rather than a monolithic function that processes all packets, as in P4, improves modularity and prevents operations on unparsed headers.

A handler is a method whose parameters match its corresponding event. For example, here is a handler for `Pkt`:

```
method HandlePkt(ip:iphdr, pl:payload) {
  if (ip.ttl>0) {
    var port:nat8 := GetOutPort(ip);
    var ip':iphdr := UpdateIpHdr(ip);
    switch.generateOutput({port}, Pkt(ip', pl));}
}
```

The handler checks the `ttl` field, calls helper methods (written by the programmer) to find an output port and update the header, then generates an output event using `generateOutput`, a method from the VeriLucid switch model.

The key design choice for handlers is which Dafny features to allow. Dafny is very expressive, with many constructs that are too complex for line-rate processing. The primary restrictions are: objects must be created on initialization, numbers must be bounded natural numbers whose bounds are powers of 2, arrays must have fixed sizes, and loops must have constant bounds (see Appendix A for a grammar and details).

3.3 Local Variables

Statements in a handler declare, assign, and modify local variables. Programmers define types, in addition to bounded naturals and booleans, using Dafny's constructs for record and subset types. A record type is an immutable struct, for example, an IP header:

```
datatype iphdr = iphdr(v_ihl:nat8,...,dst:nat32)}
```

The compiler ensures that all record types used in the compiled part of a program are ultimately composed only of VeriLucid primitive types. A subset type places restrictions on the set of values a type may take. These restrictions are expressed using any Dafny expression, and enforced by its verifier. For example, we can represent port identifiers as 8-bit non-zero naturals: `type port = x:nat8| x!=0`.

The representation of primitive types significantly impacts verification performance. We initially represented fixed-width integers as bit vectors, Dafny's native type for this purpose. However, verification was so slow as to be nearly unusable, even for simple programs. The problem is that program *specifications* often use unbounded natural numbers, e.g., to represent the real passage of time. Frequent conversion between natural numbers and bit vectors made verification slow, but replacing bit vectors with subtyped natural numbers resolved all performance issues.

3.4 Specification and Verification

In Dafny, there are two main ways to connect a specification with code: *assertion statements*, which are interspersed with other statements, and *requires (pre-) or ensures (post-) conditions*, which are the external specifications of methods.

The syntax of specifications is much the same as the syntax of code, except that specifications can make use of some non-executable language features. A *ghost variable* is a variable that never appears

in a compiled program; it is updated by (non-compiled) program statements to capture some of the history of program execution, and then read by specifications. Also, in an *ensures* clause, `old(varName)` refers to the value of `varName` before method execution began.

The key fact about verification in Dafny is that it is modular, meaning that every method must be self-contained with respect to verification, so that it can be verified using only its own code, its own specifications, and the specifications of other methods. More precisely, verification of a method can assume the method's *requires* clauses and code. Verification of the method must satisfy three proof obligations: (i) whenever the code calls another method, the *requires* clauses of the called method are satisfied; (ii) at any control point in the code where there is an *assert* statement, the asserted boolean condition is always true; and (iii) assuming that the *ensures* clauses of called methods are true upon their returns, the method being verified satisfies its own *ensures* clauses.

Modularity in Dafny is extremely important because it makes verification reliably fast and scalable. It also has disadvantages, which will be discussed in §5.

As an example of an assertion, we may wish to specify that a handler never generates output in its `ttl == 0` branch:

```
1 method HandlePkt(ip:iphdr, pl:payload) {
2   if (ip.ttl>0) { /* normal processing */ }
3   else {
4     incrementDropCount();
5     assert (switch.outputQueue == old(switch.outputQueue));
6 } }
```

The assertion here asks the verifier to prove that, for every execution of `HandlePkt` that reaches the assertion, the switch's output queue at that point has the same contents it did at the start of execution. If verification fails, there is an error highlighting the assertion.

Whereas a conventional language only allows programmers to describe the shape of input and output data (using types), pre- and post-conditions are used to specify when a method is legal to call and what guarantees it returns to the caller. Consider the specification of a helper method called in `Pkt`:

```
1 method UpdateTTL(ip : IpHdr) returns (rv : IpHdr)
2   requires ip.ttl > 0
3   ensures rv.ttl == ip.ttl-1
4   ensures rv.src == ip.src && rv.dst == ip.dst
```

The *requires* clause on line 2 specifies that `UpdateTTL` can only be called on an input with non-zero ttl. If the verifier cannot prove that this holds when the `Pkt` handler calls `UpdateTTL`, it will raise an error at the call site. Inside of `UpdateTTL`, the verifier uses the *requires* clause as an assumption. Similarly, the *ensures* clauses on lines 3 and 4 guarantee that `UpdateTTL` will decrement the ip header's ttl value and leave its source and destination fields unchanged. The specification benefits the implementer of `UpdateTTL`, who does not have to worry about underflow in the decrement operation.

4 A Library for Memory Access

Because the representation of memory in a PISA processor is written in Dafny, it can be fully specified and integrated into program verification.

PISA constraints on memory access typically require significant optimization and refactoring to satisfy, which motivates VeriLucid's introduction of *optional* checks that allow a programmer to verify and test a program even if it is not yet PISA compatible. Memory operations are also constrained by the instruction set of the target PISA architecture, which requires an extensible solution for compatibility checking in VeriLucid.

4.1 Memory Primitives

VeriLucid's memory primitives are scalar and array objects with methods for atomic read-modify-write operations, and match-action tables that the data-plane program can read, but not write. This section focuses on the data-plane-modifiable scalars and arrays because they present the most challenges; Appendix B describes the match-action primitive as an extension that uses the same general representation techniques.

Declaring a scalar or array in a program is simple, but reveals important design choices that reflect the data-plane domain.

```
1 const flag : StateVar<bool>
2 const ctr : VarArray<nat32>
```

In Dafny, objects are allocated on the heap and can be created at runtime. But in a data-plane program, there is no heap and allocation must be done at compile time. This leads to VeriLucid's first state-related restriction: state objects may only be declared as fields of the program object, outside handlers and other methods.

Objects in Dafny are references, mutable by default. But in a data-plane program, there is no such thing as a mutable reference. To reflect this, VeriLucid requires state objects to be declared as *const*s. This has a major impact on verification because a substantial challenge for the verifier is figuring out whether or not it is possible for a reference to change.

To maintain compatibility with Dafny, VeriLucid keeps allocation and initialization separate from declaration:

```
1 ctr := new VarArray.Create(N, 0);
```

This allocates the `VarArray` of `N` cells and initializes them all to 0 by calling `VarArray`'s `Create` constructor. VeriLucid enforces three restrictions on state object construction that allow it to statically allocate all memory objects: allocation (`new`) and initialization (the call to `Create`) must always happen in the same statement; constructors can only be called inside of other constructors; and constructor calls can only be passed constant value-typed arguments.

VeriLucid handlers and methods can read and update the data in a state object with `Get`, `Set`, and `GetSet` methods, e.g.:

```
1 function add(memVal : nat32, incrBy : nat32) {memVal+incrBy}
2 ctr.Set(i, add, 1); //ctr.cells[i] := add(ctr.cells[i], 1);
3 var oldCtr := ctr.GetSet(i, nocalc, 0, add, 1);
```

Here, `add` is a *memcalc* function describing how to update a value in memory; it is an abstraction of an sALU instruction. In line 2, `Set` applies the memcalc to an indexed counter cell. Any memcalc's first argument is the value retrieved from memory, its second argument is passed from the third argument of the method call, and its return value is written back to memory. Line 3 is an alternative way to update the counter cell; it does the same update as line 2, and also returns the cell's previous value as the method value. Note that `nocalc` is a built-in memcalc yielding its first argument, but another memcalc might have altered the value before returning it.

A memcalc is a *function* rather than a method because a function can be passed as an argument to a method, and because a function cannot update any non-local state—both of which support the abstraction. Functions can also be type-restricted, and memcalcs have

the type `memcalc<!t> = (t, t) -> t`. The `<!t>` notation restricts memcalcs to value types, ensuring that state objects cannot be passed as memcalc arguments.

Model implementation. Internally, the cells of a `VarArray` are an immutable sequence. Sequences can be used directly in specifications and are more efficient to verify. They also allow array methods to have very compact specifications:

```
1 class VarArray <!t> {
2  var cells : seq<t>
3  method GetSet (idx: nat, mget: memcalc<t>, garg: t,
4                 mset: memcalc<t>, sarg: t) returns (oldVal: t)
5   requires 0 <= idx < |cells|
6   ensures oldVal == mget(old(cells)[idx], garg)
7   ensures cells == old(cells)[idx:=mset(old(cells[idx]),sarg)]
8  {  oldVal := mget (cells[idx], garg);
9     cells := cells[idx := mset(cells[idx], sarg)]; }
```

In the specification for `GetSet`, line 6 requires that the index argument is within bounds of its `cells` sequence. Line 7 ensures that the returned value is the result of the appropriate call to the getter memcalc, and line 8 ensures that the final value of the array's cells is equal to the initial value with the cell at index replaced by the output of the setter memcalc. The disadvantage of storing `VarArray` state as an immutable sequence is that every update requires reconstructing the entire sequence (line 9), but this does not matter because the code is only a simulation of the real hardware.

Checking memcalc compatibility. There are many restrictions on a `memcalc` function because it must compile to a single instruction for an sALU in a PISA pipeline. Refactoring a data-plane program to meet these restrictions typically requires significant effort, and different variants of PISA have different sALU instructions [28].

As a solution, VeriLucid introduces a library-based approach for sALU compatibility checking that is optional and extensible. When programmers want to be certain that a `memcalc` can compile to an sALU instruction, they can use VeriLucid's `SALU` libraries to prove that their `memcalc` is functionally equivalent to an sALU instruction. For example:

```
1 import opened SALU8
2 function incr(stored:nat8, added:nat8): nat8
3  ensures incr(stored, added) == eval(
4  Ret(Op(x:=MemVal(stored),op:=Plus,y:=ArgVal(added),const_op:=None)))
5 { (stored + added) % 256 }
```

`SALU8` is a conservative model of the Tofino's 8-bit sALU instruction set. The library includes datatypes to define an instruction, such as `Ret`, `MemVal`, and `ArgVal`); and an eval function to interpret the instruction. In the above example, the ensures clause uses `eval` to prove that the `incr` memcalc is equivalent to the sALU instruction defined by the `Ret(...)` constructor.

The `SALU` library models the same instruction-level constraints that Lucid checks with syntactic rules in its front end. The major benefit of VeriLucid's approach is that if a program contains a memcalc that is not yet hardware-compatible, programmers can still verify and test code. VeriLucid's approach is also extensible. Writing a model, for example of a future PISA chip's sALU, is conceptually straightforward and similar to defining a small ISA. VeriLucid's current library, detailed more in Appendix C, serves as a template.

Checking ordering constraints. Conceptually, each `StateVar` or `VarArray` object is assigned to and consumes one stage of a PISA pipeline. [1]

[1] PISA chips can support multiple parallel objects in each stage; Lucid (and other PISA compilers) pack multiple objects into the same stage as an optimization when resources (e.g., memory) and constraints allow.

```
1 class BloomFilter {
2  const cols : seq<BoolArray>
3  ghost var exactSet : set<nat32>
4
5  ghost predicate inFilter (key : nat32)
6   {∀ i :: 0<=i<N ==> cols[i].cells[hash(seeds[i], key)]}
7  ghost predicate noFalseNegatives()
8   {∀ k :: k in exactSet ==> inFilter(k)}
9
10 method insert(key : nat32)
11  requires noFalseNegatives()
12  ensures  noFalseNegatives()
13  ensures  exactSet == old(exactSet) + {key}
14 { for i := 0 to N
15     { var h := hash(seeds[i], key);
16       cols[i].Set(h, swapcalc, true); }
17   exactSet := exactSet + {key};
18 } }
```

Figure 3: A Bloom filter in VeriLucid.

When a stage executes, it can perform at most one access to its `StateVar` or a single cell in its `VarArray`. Since stages are in a pipeline, all control flows must access all objects in the same order and may only access each object at most once (see §2.2).

VeriLucid re-uses Lucid's type system to check ordering. A well-typed Lucid program is guaranteed to always access objects in the order they are declared; Lucid programs that violate this constraint cannot be interpreted or compiled. To begin with, a VeriLucid programmer can ignore the issue and verify or test a reference implementation. After refinement to satisfy the PISA constraint, the programmer can put variable declarations in the intended order. Translation from VeriLucid to Lucid preserves the order of declarations and variable access, so successful Lucid compilation guarantees that the program is constraint-compliant as well as verified.

4.2 User-defined Data Structures

VeriLucid lets programmers define their own data structures as classes that use `StateVars`, `VarArrays`, and other user-defined data structures internally. Programmers use data structures the same way as `StateVar` and `VarArray` primitives: they declare instances as constant objects, initialize them inside of constructors, and call data structure methods in event handlers and functions.

A data structure class translates into a Lucid module. Programmers can use Lucid's type system to check that each module respects ordering constraints internally, independent from the rest of their code. VeriLucid maintains a program's object hierarchy in translation to Lucid. Translation does not flatten composite data structure objects, so that ordering violations are reported in terms of the objects involved rather than the underlying primitives. Appendix A.1 explains translation from VeriLucid classes to Lucid modules.

In VeriLucid, programmers can prove strong properties about data structures. For example, consider Bloom filters. A programmer can specify and implement a multi-column (i.e., partitioned [2]) Bloom as shown in Figure 3. Here we use a boolean version of `VarArray`, a built-in memcalc `swapcalc` that simply returns its second argument, and show only the insert method. `insert` iterates over a sequence of arrays, setting the appropriate position in each one to true.

The specification is more complicated. *This is common in verified programming*, and why it is important for verification tools to support high-level specifications. The specification focuses on an essential Bloom filter property, *no false negatives*: everything inserted will be correctly identified as in the represented set by a subsequent

query. This property is natural to describe with ghost variables. The `BloomFilter` uses the ghost variable `exactSet` to represent the exact set of inserted keys. The body of `insert` updates `exactSet` (line 17) as well as the data structure.

4.3 Recirculation

In PISA processors some state updates must be asynchronous, often to work around pipeline ordering requirements. For example, a handler may need to read a "control" variable near its beginning to find out how to process a packet, and then update the control variable after main processing has determined whether an update is needed. A handler cannot access the same variable twice, so, to update it, the handler must generate an update command as a recirculation event. This event will propagate through the processor and appear in a recirculation queue, where it can be dispatched and handled just like an input event. Appendix D shows an example.

5 Time, Simulation, and Testing

The correctness of a program depends on the environment in which it runs. This is particularly true of data-plane programs, which frequently track and measure time. To support both environment simulation and reasoning about time, the VeriLucid framework includes a simulation model and many features for specifying, verifying, and testing timing properties. This section reveals previously-unknown subtleties that can have a major effect on program verifiability.

5.1 Switch and Environment Simulation

The abstract model of a switch and its environment is a Dafny program representing the non-programmable parts of the switch. Its state includes the current time on the switch clock, the time `lastDispatch` of the last event dispatched, and a variety of event queues, including a set of input queues (one for each port), a recirculation queue, and an output queue.

This model omits several important parts of the switch: the pipeline stages, the traffic manager, and multiple pipes. The pipeline stages are not necessary because Lucid provides a programming abstraction in which they do not exist; the Lucid compiler allocates program resources to pipeline stages automatically (if it is possible). Furthermore, the constraints on PISA architecture (§2.2) ensure that the stages do not interact semantically. Although the traffic manager and multiple pipelines are outside our scope, this section lays the groundwork for verifying them.

In the input and recirculation queues, events are tagged with their arrival times in two forms: a bounded `timestamp` and an unbounded `time` for specification purposes. In the output queue, events are tagged with one or more output ports. Various methods called by application programs generate and enqueue recirculation events, generate and enqueue output events, and dequeue both input and recirculation events for dispatch to the application program.

The `simulateArrival` method simulates the switch environment by adding a sequence of events to an input queue. For testing, it can be called with a specific event sequence. For verification only its specification matters, and its specification—which just guarantees that the input queue remains time-ordered—means that the application program will be verified *for any possible sequence of input events.*

The other major environment phenomenon affecting programs is hardware failure. Most hardware failures reset the entire switch including all queues, so their effect is simulated simply by making the initial state of a program the same as its reset state. If the processor has "softer" failure modes, these can be signaled to the program as control events, and programmers can write application-specific handlers for them.

Obviously the switch model and an application program are separate parts: the program is written by programmers, while programmers need never even see the model. At the same time, the two parts run like coroutines and influence each other heavily in verification. Here the disadvantages of modular verification in Dafny come to the fore: with respect to larger program structures such as objects and program modules, the Dafny features that make modular verification possible (by separating methods logically and specifying all the properties of each method that other methods rely on) are complex and notoriously difficult to use. So separating the application program from the switch model was a challenging problem that took deep knowledge of Dafny to solve.

Fortunately we were able to compartmentalize this solution so that programmers can use it straightforwardly. Even though it is natural for the switch model to dispatch packets by calling event handlers, the problem can only be solved by having the program call the switch model instead. The solution entails having a template-generated simulation method `dispatch` in each application program that calls the switch model to get the next event, and then matches it with its event handler. Here is a slightly simplified `dispatch` method for the events defined in §3.1:

```
1  method dispatch ()
2    requires switch.switchInvariant ()
3    requires stateInvariant (lastTime, lastTime % T)
4    ensures switch.switchInvariant ()
5    ensures stateInvariant (e.time, e.timestamp)
6  {
7    e := switch.pickNextEvent ();
8     assert e.timestamp == e.time % T;
9     assert e.time == switch.lastDispatch;
10    assert switch.lastDispatch >= old (switch.lastDispatch);
11   if (operatingAssumptions (e) && ! stopped) {
12     if { case e.event.Pkt? =>
13               { processPkt (e.event.ip, e.event.pl); }
14          case e.event.PortStatus? =>
15            { processPortStatus(e.event.port,e.event.up);}
16          case e.event.Non? => { }
17     }
18   }
19   else { stopped := true; }
20   lastTime := e.time;
21 }
```

Each execution of `dispatch` calls `pickNextEvent` in the switch model to dequeue the next event, and then dispatches it to its matching packet handler (`processPkt` or `processPortStatus`). Other aspects of this code will be discussed in the next section.

5.2 Reasoning About Time

In reasoning about time in the real world, there are two things we know for sure: (i) time never moves backward (earlier), and (ii) time moves forward (later) regardless of what happens in any particular machine or program. A data-plane program can observe time only through event timestamps, however, which means that the program's view of time can be distorted in several ways. First and most obvious, switch timestamps are bounded numbers—when they roll over, time appears to move backward. And this can have a significant effect on the measurement of time: if nanosecond timestamps are stored in 32-bit registers, then timestamps roll over in 4.3 seconds!

Another distortion is that within an event handler, time appears frozen at the timestamp, and never moves forward. Thus, to reason about the passage of time, it is necessary to consider the implicit loop in which the switch dispatches an event and a handler processes it, over and over again. In terms of a program, the implicit loop occurs because `dispatch` is executed repeatedly.

In the VeriLucid framework there are many built-in features to facilitate reasoning about this loop. For example, `switch.switchInvariant` is a predicate defined in the switch model to include all its state properties that programs rely on. Also, `stateInvariant` is a predicate that the programmer must define as the application-specific invariant. Both appear in requires clauses of `dispatch` (lines 2 and 3) so that event handlers can rely on them. Both appear in ensures clauses (lines 4 and 5) so that code cannot interfere with the switch model, and must preserve the application invariant. All event handlers must have these and other framework-defined clauses in their specifications.

In the framework, `switch.lastDispatch` is a built-in ghost variable in which the switch model keeps the time of the last event it dispatched, and `lastTime` is a built-in ghost variable in which the program keeps track of the last event it processed. The three assertions (lines 8-10) simply document what verification of the switch model guarantees: the `timestamp` of an event is its `time` modulo the timestamp bound `T`, `time` of the event is the same as `lastDispatch` in the switch, and dispatch times increase monotonically.

We will now illustrate the importance of these and other features by showing how to verify a basic timer, which measures the time elapsed since the timer was set, and checks when the elapsed time reaches a limit. Measuring elapsed time is fundamental to many data-plane programs, which may calculate flow rates, evict expired entries from caches, or detect faults elsewhere in the network. For readability this is a reference implementation, in which we use ordinary state variables that have not yet been refined into `StateVar` primitives. When an event sets the timer, `timeSet` gets its `time`, `timestampSet` gets its `timestamp`, and `limit` gets the time limit:

```
1 ghost var timeSet : nat
2 var timestampSet : nat32
3 var limit : nat32
```

The `stateInvariant` ensures the sanity of the variable values:

```
1 ghost predicate stateInvariant (time:nat, timestamp:nat32)
2 {     ( timestampSet == timeSet % T )
3    && ( timeSet <= time )
4 }
```

If an event handler depends on whether the timer has expired, these cases can be used:

```
1 if (e.timestamp - timestampSet) % T >= limit {
2    // code for case when timer expired
3       assert (e.time - timeSet) >= limit;
4       . . .
5    } else {
6    // code for case when timer not expired
7       assert (e.time - timeSet) < limit;
8       . . .
9 }
```

Checking whether the timer has expired (line 1), using modulo arithmetic, is easy and efficient. However, this program will not verify, because the correctness condition at line 7 does not hold in general. The problem is that line 1 does not calculate the elapsed time since the timer was set, but rather the elapsed time modulo T. If the elapsed time is less than T, the calculation is correct. As the elapsed time grows greater than T, the condition on line 1 will oscillate between true and false. In other words, once the timer is set, one timestamp rollover is acceptable, but more than one is not.

This is a typical situation in which a program relies on an environment assumption, in this case about inter-packet gaps. It is especially valuable to verify timing properties so that assumptions like this can be identified, checked for validity in their real context, and documented to future-proof the code. Verification of this program requires the following predicate:

```
1 predicate operatingAssumptions (e: TimedEvent)
2    {   e.time < timeSet + T }
```

How is this predicate used within the VeriLucid framework? This is a bit tricky: operating assumptions cannot go in the switch model because they are program-dependent, but if we put them in the program then they cannot prevent an event that violates the assumptions from being picked by the switch. The solution is shown in the `dispatch` code. If an event arrives and does not satisfy `operatingAssumptions`, it is not processed, but rather sets the ghost variable `stopped` to true. After that, execution ceases, because no event will ever be processed again.

This program behavior may seem odd, but it is a standard convention in specification and verification—and lines 11 and 19 will not be compiled! The convention embodies the semantics that nothing can be proved about a program if its operating assumptions are violated. Its benefit is that this execution trace will not cause a spurious verification error, while all execution traces whose events satisfy the operating assumptions will be verified. We were happy to see that Dafny verifies the complete program automatically, which means that we did not have to fuss with the axioms of modulo arithmetic.

PISA processors often have a 48-bit nanosecond clocks. In our programs we usually store the high-order 32 bits of timestamps. This means that a clock tick in the program is 2^{16} nanoseconds, so our timers can operate within any interval less than 3.25 days, instead of 4.3 seconds.

5.3 Another Way Time Can Move Backward

There is a third way in which a program's view of time can be distorted, specifically by going backward. This problem does not seem to have been noticed or analyzed previously.

The heart of the switch model is the method that picks the next event to be dispatched to event handlers. If it picked from only one queue, it would be trivial. There are, however, multiple input queues and a recirculation queue, and PISA switches allow the setting of priorities among them. With priorities, it is possible for the switch to select an event with a timestamp earlier than one that has been dispatched already.

This is not a trivial problem, even if the discrepancy is small, because the calculation of elapsed time with bounded timestamps assumes we know which of two timestamps is later. For example consider what happens in the timer code when `timestampSet` is set to n at timestamp n, and the very next event has timestamp $n-1$. The elapsed time will be calculated to be the maximum timestamp, this will certainly equal or exceed the `limit` (which has the same bound), and the timer will expire—even though it has just been set!

Whether a situation like this can cause trouble in a real program might be difficult to assess (see below). What we know for sure is that time running backward—except for the well-understood large-scale phenomenon of rollover—prevents verification of almost all

timing properties. For example, once a specific waiting period has elapsed, typical invariants state that it remains elapsed, regardless of subsequent events. These invariants will not hold if time can move backward and then forward again.

In VeriLucid the switch model corrects this problem in a simple way: when `pickNextEvent` picks an event whose time is earlier than `lastDispatch`, it alters the time/timestamp to `lastDispatch`/(`lastDispatch` mod T) before returning the event for handling in `dispatch`. This preserves the crucial assertions at `dispatch` lines 8-10. The fact that time does not move forward in this case, but only stays the same, is not a problem, because all of the axioms in VeriLucid are designed in expectation of this case.

Although the algorithm in the switch model uses unbounded variables, it is easy to implement the same solution in the dispatch hardware of a switch. To check the next timestamp `next` against a bounded version of `lastDispatch`, the hardware evaluates this expression:

```
1    (next - lastDispatch) % T
```

If between the two timestamps the clock moved m ticks forward, for $0 <= m < T$, then the value will be m. If between the two timestamps the clock moved m ticks backward, for $1 <= m < T$, then the value will be $T - m$. Both hold regardless of rollover during the move. The maximum forward move is determined by inter-event gaps, while the maximum backward move is determined by the maximum queueing delay within the input queues. Because of the typical values of these measures within a switch, the maximum movement M will be very much smaller than T, making backward movement reliably detectable and correctable.

What if your programmable switch does not correct the problem? To assess possible solutions, you must consider the factors above in the context of your program. How long is a clock tick in your program, and how big is T? (Both depend on how timestamps are saved.) What are the minimum and maximum inter-event gaps? What will be the maximum queueing delay in your input and recirculation queues? (This depends on their priorities, as well as overall traffic.) If the answers are favorable, time as observed by your program is extremely unlikely to run backward. But if the answers are unfavorable, time might run backward. Worse yet, if there might be both very small and very large inter-event gaps, the correction algorithm described above might not work for your program, so that you cannot build your own solution.

Given all these complexities and uncertainties, we strongly recommend that future programmable processors be designed with this problem in mind, and that the correction above be built into the non-programmable hardware.

5.4 Testing VeriLucid Programs

Although verification mostly supersedes testing, testing VeriLucid programs can still be useful. For one reason, testing is concrete and familiar. More importantly, programmers who are new to verification might not see immediately how to write the strong specifications that guarantee correctness. With testing, programmers can gain program insight and the necessary specification experience gradually.

Figure 4 shows a template for a simple test. Note that Dafny will not compile this program until it is verified, and the verifier requires verifiable invariants on all loops. The invariant on a dispatch loop

```
1  method Main() {
2    var maxSteps := 5;
3    var prog := new Program ();      // initializes switch
4    prog.switch.simulateArrival (0, fiveInputEvents());
5                                     // input port 0
6    while (maxSteps > 0)
7        invariant prog.switch.switchInvariant ()
8        invariant prog.stateInvariant
9          (prog.lastTime, prog.lastTime % T)
10   {
11     prog.dispatch ();
12     prog.switch.printOutputQueue ();
13     maxSteps := maxSteps - 1;
14   }
15 }
```

Figure 4: A simple test template.

is already present in the specification of `dispatch`, but the verifier does not know that, so the template copies it as a loop invariant. As the program specification gets stronger, the `dispatch` specification will not change, but the *definition* of the predicate `stateInvariant` will include more and more information.

Our framework includes several testing aids. As shown, tests can call the switch model to print queues, as well as relying on print statements in event handlers. Tests can also call `switch.simulateClockTick(j)` to move the processor clock j ticks forward.

6 Implementation and Evaluation

The components of VeriLucid (shown in Figure 2) are implemented entirely in Dafny.[2] The largest component is the translator (approximately 4000 LoC), implemented as a standard backend module in Dafny's compiler.

Applications. Table 1 summarizes the applications implemented in VeriLucid so far. Each application is specified, verified, and compiled to the 3.2 Tb/s Tofino. We summarize findings below, then focus on two case studies: one to compare with Verifiable P4 [36], the only prior PISA-compatible verification tool capable of supporting VeriLucid's level of expressiveness, and another to show a subtle bug discovered in a previous publication.

Programming. All programs used VeriLucid's high-level abstractions for a degree of modularity and compactness that would be impossible in P4. The sliding-window Bloom filter uses Bloom filters internally, the MAC learner uses recirculation events for new flow installations, and almost all applications use `VarArrays` and general-purpose functions for structure. None of these constructs are supported in P4, hence the orders-of-magnitude increase in code size versus P4.

Specification. Specifications used functions/predicates for structure, ghost variables to model state, and abstract tests to prove temporal properties (in the SBF and MAC learner). Specifications were around 2X longer than implementations, but short in absolute lines of code. In comparison, P4 verifiers offer few of these abstractions, so when equivalent specifications are possible, they are much longer as §6.1 will show.

State and temporal properties. All of the examples have specifications relating to data-plane modifiable state. The sliding-window BF in §6.1 includes a temporal data structure invariant. The MAC learner, discussed more in Appendix E, shows how to operate the switch model by hand to prove temporal properties related to event recirculation. Finally, a more sophisticated DNS reflection defense

[2] VeriLucid is available at https://github.com/PrincetonUniversity/verilucid.

Program	Specification Description	Verif. Time	Dafny LoC,Spec	Lucid LoC	P4 LoC	Stages
Bloom filter (BF) [36]	no false negatives, insert/query correct	1.06s	19, 63	27	364	1
Sliding-window BF [36]	no early timeouts, insert/query correct	7.98s	46, 78	67	2429	11
Count Min [16]	count/query correct, queries return min	1.43s	21, 36	50	547	5
MAC Learning [22]	new flow installation at first packet gap	1.05s	27, 49	34	526	5
Privacy-safe Telemetry [18]	samples are anonymized	0.97s	8, 15	9	331	3
TurboFlow [29]	metrics converge with centralized model	1.61s	31, 69	39	545	8
ML Parameter Agg. [27]	correct aggregation and broadcast trigger	1.23s	26, 50	29	1836	9

Table 1: VeriLucid benchmark program statistics. Stages are physical stages of a 12-stage Intel Tofino.

that was co-developed with VeriLucid but is beyond the scope of this paper illustrates the full use of temporal properties expressed as event dispatch invariants (see §8).

Verification time. Most verification tasks took around a second, fast enough to provide developer feedback without disruption. Verification was fast due to Dafny's modular approach to programming and verification, which allows the verifier to check each method independently and in parallel, and also the carefully selected representational choices and domain restrictions as described throughout the paper.

Compiling to hardware. All programs fit within the constraints of the Tofino. This often required us to optimize program control flow. For example, to fit the Sliding-window Bloom filter in 11 stages, we had to optimize the Bloom filters it uses internally. We discovered that refactoring the BF's query method to return early (from inside of a for loop) allowed Lucid's compiler to fully parallelize the operation; VeriLucid helped us prove that this optimization was safe.

Automation of verification. Although Dafny verification is only semi-automated in theory, *within the VeriLucid framework* it has never failed to verify a true theorem automatically. What this means for the programmer is that, if verification fails, it is simply necessary to add more specification—usually more or stronger requires and ensures clauses. In other words, you just have to tell the tool more about what you are assuming to be true, and soon you will either find your error or get an automated proof. We have emphasized working within the VeriLucid framework because its careful design and built-in specifications have squeezed out many verification pitfalls.

6.1 Comparison with Verifiable P4

The comparison with Verifiable P4 uses their driving example: a sliding-window Bloom filter (SBF). A plain Bloom filter does not support deletions, which are necessary for applications such as firewalls. An SBF handles deletions by removing set elements after a fixed timeout. We will be concerned with specifying and verifying the property *no early timeouts,* which is really a timed version of *no false negatives*, allowing a query to return `false` for a once-inserted key, once enough time has elapsed.

An SBF uses multiple Bloom filters (at least 3) as ordered panes that slide (or tumble) over each other at regular intervals. The first filter is hot, containing newly-inserted keys. The second filter is warm, containing recently-inserted keys. Queries check for elements in the hot and warm filters. The last filter is cold, holding only keys old enough to delete. For timeouts, the SBF rotates the filters periodically, say every I clock ticks: hot becomes warm, warm becomes cold, and cold gets reset and recycled into hot. The reset operation does not have to be performed all at once—it can be done incrementally over the cold period.

```
1  class SBF {
2   var clearIdx : StateVar<Idx>
3   var panes : seq<BloomFilter> // 4 BloomFilters
4
5   function curPane (timestamp: nat32) : Pane
6    {  ( timestamp / I ) % numPanes }
7   predicate noFalseNegs(bf : BloomFilter)
8    { ∀ k :: k in bf.exactSet ==> bf.inFilter(k) }
9
10  method insert (key : nat32, ts : nat32)
11   requires ∀bf in panes :: noFalseNegs(panes[bf])
12   ensures ∀ bf in panes :: noFalseNegs(panes[bf])
13   ensures panes[curPane(ts)].exactSet ==          // hot pane
14        old(panes[curPane(ts)].exactSet) + {key}   // updated
15   ensures unchanged(panes[(curPane(ts)-1)%4])     // warm panes
16   ensures unchanged(panes[(curPane(ts)-2)%4])     // unchanged
17  {  var j := clearIdx.GetSet(nocalc,0,incr,1);   // mod col length
18     match curPane (ts) {
19       case 0 => {panes[0].insert(key); panes[1].clearCell(j);}
20       case 1 => {panes[1].insert(key); panes[2].clearCell(j);}
21       case 2 => {panes[2].insert(key); panes[3].clearCell(j);}
22       case 3 => {panes[3].insert(key); panes[0].clearCell(j);} }
23   }
24 } // End class SBF
```

Figure 5: Specification and implementation of an SBF's insert in VeriLucid.

Figure 5 shows part of an SBF in VeriLucid. Here, we use four panes instead of three (with two warm ones) and power-of-two values for I, so that it can calculate the current pane from the current timestamp. The match statement on lines 18-23 gets the index of the current hot pane, inserts the key into the corresponding Bloom filter, and clears one cell of the cold pane (which includes removing from the exact set any keys it represents). The index to clear is read and updated with an atomic operation on line 17. Every `query` also clears a cell of the cold pane, to get the pane cleared completely during I clock ticks.

The specification of `insert` on lines 11-16 of Figure 5 is limited to properties necessary to prove *no early timeouts*. Lines 11 and 12 establish that none of the internal Bloom filters have false negatives when `insert` is called or returns. Lines 13 and 14 say that when `insert` returns, the hot pane stores all the same keys it did when it was called, plus the new key. Finally, lines 15 and 16 guarantee that `insert` never changes either of the warm panes. In practice, a programmer is likely to figure out the right specification for insert iteratively, e.g., while working out the proof for *no early timeouts*.

```
1  const T : nat := max32 // max 32-bit timestamp
2  method noEarlyTimeoutsTest () {
3    var sbf := new SBF ();
4    var key: nat32 := *; var tInsert:nat := *; //random values
5    sbf.insert (key, tInsert%T);
6    var found := sbf.query (key, tInsert%T);
7    var inserted : nat := tInsert;
8    while (inserted <= tInsert + 2*I)
9      invariant ∀bf in sbf.panes :: sbf.noFalseNegs(panes[bf])
10     invariant key in sbf.panes[sbf.curPane(tInsert%T)].exactSet
11     invariant found == true
12   {
13     var doInsert:bool := *; var k:nat32 := *;//random vals
14     if (doInsert) { sbf.insert(k, inserted%T); }
15     else { var kFound := sbf.query(k, inserted%T); }
16     found := sbf.query (key, inserted%T);
17     if (found == false) { print "BUG: key not found!"; }
18     var jump : nat :| jump > 0; //random val greater than 0
19     inserted := inserted + jump;
20   }
21 }
```

Figure 6: A generalized test for no early timeouts.

Component	Language	Lines of code
	Verifiable P4	
Implementation	P4	416
Functional model	Gallina (Rocq)	480
Functional spec.	Verifiable P4	1258
Proofs	Gallina (Rocq)	3120
Total	3 languages	4,013 LoC
	VeriLucid	
Implementation	VeriLucid	65
Specification	VeriLucid	141
Total	1 language	206 LoC

Table 2: Lines of code for SBF.

A generalized test. There are many ways to prove this property. Figure 6 demonstrates how to use verification to prove general properties in a test method. The code creates a new SBF, inserts a random key at a random time (lines 4 and 5), runs a loop that performs a random insert or query operation, checks to see if the original key is still found, and then jumps ahead by a random non-zero unit of time until $2 * I$ time units (the timeout for a 4-pane filter) have elapsed (lines 8-20). The loop invariant on line 9 requires all Bloom filter panes to maintain *no false negatives*; 10 requires that the original key can still be found in the pane where it was inserted; and 11 requires that the query for the key in each iteration returns true. The above code *looks* like a test case, and executing it once will test a single trace, as expected. But by *verifying* the method, we prove that the loop invariant `found == true` holds for any possible choice of the randomly-initialized variables shown in the function, which directly implies *no early timeouts*.

Programmer effort. Table 2 compares the SBF in VeriLucid with equivalent Verifiable P4. The VeriLucid version was under 250 lines of code—less than $\frac{1}{10}$ as many as in Verifiable P4. Further, all code was in one language, rather than separate implementation and specification languages with different tools and levels of abstraction.

```
1  const      keys       : VarArray<FlowKey> //in this example,
2  const      lens       : VarArray<nat32>   //keys are IP addrs
3  ghost const standalone : ExactCollector
4  ghost const backend    : ExactCollector
5  ghost predicate stateInvariant()
6  { ∀ k: FlowKey :: standalone.getLen(k) ==
7                    backend.getLen(k) + getCachedLen(k) }
8  // if key in cache then getCachedLen is cached len else 0
9
10 method ProcessPacket(ip : Ip)
11   requires stateInvariant()
12   ensures stateInvariant()
13 {
14   standalone.Update(toKey(ip), ip.len);
15   var key : FlowKey := toKey(ip);
16   var idx : FlowIdx := hash(seed,key);
17   var oldKey := keys.GetSet(idx, nocalc, 0, swapcalc, key);
18   if (oldKey != key) {
19    var oldLen := lens.GetSet(idx, nocalc, 0, swapcalc, ip.len);
20    backend.Update(oldKey, oldLen);
21    switch.generateOutput(collecPort, FlowRec(oldKey, oldLen));
22   } else {
23    lens.Set(key, add, ip.len); // verification failure
24   } }
```

Figure 7: TurboFlow data-plane cache in VeriLucid.

Clearly, VeriLucid requires significantly less programmer effort than Verifiable P4, but the two are complementary rather than competing. Verifiable P4 leaves fewer unverified components in the compilation chain because it is at a lower level of abstraction, which motivates multi-level verification [38], or verifying the VeriLucid translator itself. Also, proof assistants can be better for complex code, which motivates research on hybrid approaches [26].

6.2 Verifying a Hybrid System

We discovered a subtle bug in TurboFlow [29], a flow-monitoring system with a hybrid architecture. The data-plane component (Figure 7) uses a replace-on-collision cache. When a new packet collides with an existing flow record, the existing flow record is exported to a backend collection server. Caching reduces the backend's workload while (in principle) allowing it to compute the same values for commutative metrics such as flow length (what we are tracking here).

To prove correctness, we formalized a model of an exact flow collector—having no size limit—that could operate in either a standalone mode or as a backend for TurboFlow. Our goal was to prove the `stateInvariant` on lines 5-8: the length of a flow in the standalone collector is always equal to the length of the flow in the backend collector, plus the length of the flow in TurboFlow's internal cache.

Dafny's verifier cannot prove `stateInvariant`, raising an error on line 23 and providing a counterexample: a packet that causes `lens` to overflow. The subtle aspect of this bug is that simply preventing overflow with a saturating counter is not enough. If `incr` saturated the counter, `TurboFlow` would still lose information for long flows. Instead, `TurboFlow` must detect an overflow condition before it happens and export a record. One fix is replacing the increment on line 23 with:

```
var oldLen := lens.GetSet(idx,nocalc,0,safeIncr,ip.len);
if (oldLen >= max32 - maxPktLen) {
 backend.Update(oldKey, oldLen);
 switch.generateOutput(collecPort, FlowRec(key, oldLen));
}
```

where `safeIncr` checks for potential overflow, and if detected behaves like `swapcalc`. This fixes the program at the cost of adding 1 more stage.

7 Related Work

VeriLucid is related to a number of research areas focused on making it easier to build correct data-plane and network programs.

Other P4 verifiers. We have already said much about the limitations of other P4 verification efforts. Vera [32], p4v [20], ASSERT-P4 [13], Aquila [34], and Π4 [12] focus on safety and stateless application properties of P4. P4LTL [43] and P4inv [42] add simple stateful properties. These tools lack VeriLucid's higher-level specification constructs. Scalability is also a challenge because automatic verifiers (besides Π4) operate monolithically. They translate the program and its specification into an intermediate verification language [4] and attempt to verify everything at once. This scales poorly, especially in programs with state, in contrast with Dafny's modular approach. Interactive verifiers (Verifiable P4 [36] and HOL4P4 [3]) are more expressive and scalable, but at the cost of high user effort.

Formalizing hardware semantics. Several efforts have formally modeled data-plane hardware components at a low level, including the Tofino [37] and eBPF VM [3]. VeriLucid is complementary, providing a higher-level model for verification and demonstrating how Dafny can be used for such modeling in the future. Separately, other verification tools focus on modeling parsers [10] or match-action tables [11]. Custom parser functions and match-action tables are both supported in Lucid and reasonable to model in Dafny, but not yet supported by VeriLucid's compiler.

Higher-level data-plane languages. In addition to Lucid, there are Lyra [14] and NAP [25]. We chose Lucid over NAP because it is more general, and over Lyra because it is higher-level (as well as having an available implementation). From the programmer's perspective, all of these languages change the role of verification somewhat: they build in more safety properties, and make straightforward algorithms easier to understand, so verification of simple properties is less useful. At the same time, higher-level languages enable more complex and sophisticated programs, which makes verification of application-level properties even more important.

Portability. An important topic in data-plane research is portable languages [14] and compilers [19, 39]. VeriLucid inherits portability from Lucid, which can also compile to C for general-purpose processors (GPPs). Although GPPs have fewer restrictions, getting high and deterministic performance out of them requires careful program design and many of the same considerations as on PISA processors. Thus, we believe that both the motivation and approach demonstrated by VeriLucid are broadly useful, regardless of hardware platform.

8 Limitations and Future Work

VeriLucid is the first attempt to combine high-level data-plane programming, expressive specification, and automated verification into a language for programming hardware-accelerated data planes. There are several limitations and many directions of future work.

Usability. The verification research community has worked for decades to understand what makes a verification language usable, and their progress is reflected in Dafny and preserved by VeriLucid. Nevertheless, neither specification nor verification is easy, especially if—as is commonly the case with data-plane programming—programmers are not already familiar with the abstract concepts on which specification and verification rely. This paper has only weak evidence of usability, based on a variety of data-plane programs written and verified by the authors.

Future work will address this limitation, first by developing a domain-specific method to guide data-plane programmers in thinking about their application and its operating assumptions, specifying its absolute requirements and flexible goals, and refining a reference implementation with the approximations, trade-offs, and optimizations necessary for PISA execution. The method will be explained through our previously-mentioned case study of a defense against DNS reflection attacks. Once we have a method, even if preliminary, it becomes meaningful to see how new users react to the combination of VeriLucid and its method.

Instruction set. VeriLucid's sALU instruction model represents a conservative subset of the Tofino and Lucid's sALU capabilities. It omits instructions for paired arrays (which can be updated based on each others' values), single-bit arrays, meters, and the Tofino sALU's math mode. The sALU instruction set modeled here has been shown useful for a wide range of applications [30]. Future work can use the current library as a template to represent more intricate instruction sets. These might include a more complete model of the Tofino's sALUs, or instruction sets of future PISA chips.

Hardware modeling. Our processor model does not include the traffic manager, multiple pipelines, configurable queues, and other hardware details. Lucid abstracts a subset of this machinery that is expressive enough for a range of programs. Modeling a PISA switch at a lower level of abstraction entails significant target-specific reverse engineering because manufacturers in general do not provide precise hardware specifications. VeriLucid's approach of modeling components as libraries in a high-level verification language can accelerate future work in this area and make formal modeling of hardware more tractable for non-experts in verification.

New chip architectures. VeriLucid targets only PISA processors, while there are now several new programmable chips on the market with a variety of architectures [9, 44]. Even though other architectures may be preferable for stateless tasks, PISA embodies the restrictions necessary for executing stateful programs at line rate. As programmable hardware expedites more and more tasks, stateful programs will become more common. Given the fundamental performance advantages of pipeline architectures like PISA [45], we believe that many future chips for stateful programming at line rate will rely on on some version of the PISA constraints, so it is important to know how to work with them.

Trustworthy tools. VeriLucid trusts three unverified compilers: the VeriLucid-Lucid compiler; the Lucid-P4 compiler, and finally the P4-Tofino compiler. It is an open question how much of these compilers can be verified. The VeriLucid-Lucid compiler is already implemented in Dafny, so we expect verification of that stage to be possible. Much of Lucid's compiler is largely based on syntax-tree transformations, which are also typically amenable to verification. Deeper stages of the Lucid and P4 compiler, involving complicated search procedures, would be more challenging.

Ethics. This paper does not raise any ethical concerns.

Acknowledgements. We thank the anonymous reviewers and shepherds for their feedback. This research was partially funded by DARPA Grant HR0011-20-C-0160 and National Science Foundation Grants 2223515 and 2429485.

References

[1] Anurag Agrawal and Changhoon Kim. 2020. Intel Tofino2–A 12.9 Tbps P4-programmable Ethernet switch. In *IEEE Hot Chips 32 Symposium (HCS)*. IEEE Computer Society, 1–32.

[2] Paulo Sérgio Almeida. 2023. A Case for Partitioned Bloom Filters. *IEEE Trans. Comput.* 72, 6 (2023), 1681–1691. doi:10.1109/TC.2022.3218995

[3] Anoud Alshnakat, Didrik Lundberg, Roberto Guanciale, and Mads Dam. 2024. HOL4P4: Mechanized small-step semantics for P4. *Proceedings of the ACM on Programming Languages* 8, OOPSLA1 (2024), 223–249.

[4] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K Rustan M Leino. 2005. Boogie: A modular reusable verifier for object-oriented programs. In *International Symposium on Formal Methods for Components and Objects*. Springer, 364–387.

[5] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: programming protocol-independent packet processors. *SIGCOMM Computer Communication Review* 44, 3 (July 2014), 87–95. doi:10.1145/2656877.2656890

[6] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. 2013. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. *ACM SIGCOMM Computer Communication Review* 43, 4 (2013), 99–110.

[7] Mihai Budiu and Chris Dodd. 2017. The P416 Programming Language. *SIGOPS Oper. Syst. Rev.* 51, 1 (Sept. 2017), 5–14. doi:10.1145/3139645.3139648

[8] Aleks Chakarov, Jaco Geldenhuys, Matthew Heck, Michael Hicks, Sam Huang, Georges-Axel Jaloyan, Anjali Joshi, K. Rustan M. Leino, Mikael Mayer, Sean McLaughlin, Akhilesh Mritunjai, Clement Pit-Claudel, Sorawee Porncharoenwase, Florian Rabe, Marianna Rapoport, Giles Reger, Cody Roux, Neha Rungta, Robin Salkeld, Matthias Schlaipfer, Daniel Schoepe, Johanna Schwartzentruber, Serdar Tasiran, Aaron Tomb, Emina Torlak, Jean-Baptiste Tristan, Lucas Wagner, Michael W. Whalen, Remy Willems, Tongtong Xiang, Tae Joon Byun, Joshua Cohen, Ruijie Fang, Junyoung Jang, Jakob Rath, Hira Taqdees Syeda, Dominik Wagner, and Yongwei Yuan. 2025. Formally Verified Cloud-Scale Authorization. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. 2508–2521. doi:10.1109/ICSE55347.2025.00166

[9] Sharad Chole, Andy Fingerhut, Sha Ma, Anirudh Sivaraman, Shay Vargaftik, Alon Berger, Gal Mendelson, Mohammad Alizadeh, Shang-Tse Chuang, Isaac Keslassy, Ariel Orda, and Tom Edsall. 2017. dRMT: Disaggregated Programmable Switching. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (Los Angeles, CA, USA) *(SIGCOMM '17)*. Association for Computing Machinery, New York, NY, USA, 1–14. doi:10.1145/3098822.3098823

[10] Ryan Doenges, Tobias Kappé, John Sarracino, Nate Foster, and Greg Morrisett. 2022. Leapfrog: certified equivalence for protocol parsers. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 950–965. doi:10.1145/3519939.3523715

[11] Dragos Dumitrescu, Radu Stoenescu, Lorina Negreanu, and Costin Raiciu. 2020. bf4: towards bug-free P4 programs. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication* (Virtual Event, USA) *(SIGCOMM '20)*. Association for Computing Machinery, New York, NY, USA, 571–585. doi:10.1145/3387514.3405888

[12] Matthias Eichholz, Eric Hayden Campbell, Matthias Krebs, Nate Foster, and Mira Mezini. 2022. Dependently-typed data plane programming. *Proc. ACM Program. Lang.* 6, POPL, Article 40 (Jan. 2022), 28 pages. doi:10.1145/3498701

[13] Lucas Freire, Miguel Neves, Lucas Leal, Kirill Levchenko, Alberto Schaeffer-Filho, and Marinho Barcellos. 2018. Uncovering Bugs in P4 Programs with Assertion-based Verification. In *Proceedings of the Symposium on SDN Research* (Los Angeles, CA, USA) *(SOSR '18)*. Association for Computing Machinery, New York, NY, USA, Article 4, 7 pages. doi:10.1145/3185467.3185499

[14] Jiaqi Gao, Ennan Zhai, Hongqiang Harry Liu, Rui Miao, Yu Zhou, Bingchuan Tian, Chen Sun, Dennis Cai, Ming Zhang, and Minlan Yu. 2020. Lyra: A Cross-Platform Language and Compiler for Data Plane Programming on Heterogeneous ASICs. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication* (Virtual Event, USA) *(SIGCOMM '20)*. Association for Computing Machinery, New York, NY, USA, 435–450. doi:10.1145/3387514.3405879

[15] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. 2015. IronFleet: proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California) *(SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 1–17. doi:10.1145/2815400.2815428

[16] Mary Hogan, Devon Loehr, John Sonchack, Shir Landau Feibish, Jennifer Rexford, and David Walker. 2025. Automated Optimization of Parameterized Data-Plane Programs With Parasol. *IEEE Transactions on Networking* 33, 2 (2025), 526–540. doi:10.1109/TNET.2024.3499833

[17] Andrew Johnson, Ryan Beckett, Xiaoqi Chen, Ratul Mahajan, and David Walker. 2024. Sequence abstractions for flexible, line-rate network monitoring. In *Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation* (Santa Clara, CA, USA) *(NSDI'24)*. USENIX Association, USA, Article 88, 28 pages.

[18] Hyojoon Kim and Arpit Gupta. 2019. ONTAS: Flexible and Scalable Online Network Traffic Anonymization System. In *Proceedings of the 2019 Workshop on Network Meets AI & ML* (Beijing, China) *(NetAI'19)*. Association for Computing Machinery, New York, NY, USA, 15–21. doi:10.1145/3341216.3342208

[19] Jiaxin Lin, Zhiyuan Guo, Mihir Shah, Tao Ji, Yiying Zhang, Daehyeok Kim, and Aditya Akella. 2025. Enabling Portable and High-Performance SmartNIC Programs with Alkali. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*. 107–126.

[20] Jed Liu, William Hallahan, Cole Schlesinger, Milad Sharif, Jeongkeun Lee, Robert Soulé, Han Wang, Călin Caşcaval, Nick McKeown, and Nate Foster. 2018. p4v: practical verification for programmable data planes. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (Budapest, Hungary) *(SIGCOMM '18)*. Association for Computing Machinery, New York, NY, USA, 490–503. doi:10.1145/3230543.3230582

[21] Devon Loehr and David Walker. 2022. Safe, modular packet pipeline programming. *Proc. ACM Program. Lang.* 6, POPL, Article 38 (Jan. 2022), 28 pages. doi:10.1145/3498699

[22] Devon Kennedy Loehr. 2024. *Lucid: A high-level, easy-to-use dataplane programming language*. Ph. D. Dissertation. Princeton University.

[23] K. Rustan M. Leino. 2017. Accessible Software Verification with Dafny. *IEEE Software* 34, 6 (2017), 94–97. doi:10.1109/MS.2017.4121212

[24] Md Rakib Hossain Misu, Cristina V. Lopes, Iris Ma, and James Noble. 2024. Towards AI-Assisted Synthesis of Verified Dafny Methods. *Proc. ACM Softw. Eng.* 1, FSE, Article 37 (July 2024), 24 pages. doi:10.1145/3643763

[25] Mengying Pan, Hyojoon Kim, Jennifer Rexford, and David Walker. 2023. NAP: Programming Data Planes with Approximate Data Structures. In *Proceedings of the 6th on European P4 Workshop* (Paris, France) *(EuroP4 '23)*. Association for Computing Machinery, New York, NY, USA, 33–39. doi:10.1145/3630047.3630196

[26] Wojciech Różowski, Georges-Axel Jaloyan, and Sean McLaughlin. 2025. Lean on Dafny: Exploring Interactive Verification of Dafny Programs in Lean. In *Dafny Workshop*.

[27] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtárik. 2021. Scaling distributed machine learning with In-Network aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 785–808.

[28] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. 2016. Packet Transactions: High-Level Programming for Line-Rate Switches. In *Proceedings of the 2016 ACM SIGCOMM Conference* (Florianopolis, Brazil) *(SIGCOMM '16)*. Association for Computing Machinery, New York, NY, USA, 15–28. doi:10.1145/2934872.2934900

[29] John Sonchack, Adam J. Aviv, Eric Keller, and Jonathan M. Smith. 2018. Turboflow: information rich flow record generation on commodity switches. In *Proceedings of the Thirteenth EuroSys Conference* (Porto, Portugal) *(EuroSys '18)*. Association for Computing Machinery, New York, NY, USA, Article 11, 16 pages. doi:10.1145/3190508.3190558

[30] John Sonchack, Devon Loehr, Jennifer Rexford, and David Walker. 2021. Lucid: a language for control in the data plane. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference* (Virtual Event, USA) *(SIGCOMM '21)*. Association for Computing Machinery, New York, NY, USA, 731–747. doi:10.1145/3452296.3472903

[31] John Sonchack, Oliver Michel, Adam J Aviv, Eric Keller, and Jonathan M Smith. 2018. Scaling hardware accelerated network monitoring to concurrent and dynamic queries with *Flow. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 823–835.

[32] Radu Stoenescu, Dragos Dumitrescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. 2018. Debugging P4 programs with vera. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (Budapest, Hungary) *(SIGCOMM '18)*. Association for Computing Machinery, New York, NY, USA, 518–532. doi:10.1145/3230543.3230548

[33] The Rocq Development Team. 2025. *The Rocq Prover*. doi:10.5281/zenodo.17473943

[34] Bingchuan Tian, Jiaqi Gao, Mengqi Liu, Ennan Zhai, Yanqing Chen, Yu Zhou, Li Dai, Feng Yan, Mengjing Ma, Ming Tang, Jie Lu, Xionglie Wei, Hongqiang Harry Liu, Ming Zhang, Chen Tian, and Minlan Yu. 2021. Aquila: A practically usable verification system for production-scale programmable data planes. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference* (Virtual Event, USA) *(SIGCOMM '21)*. Association for Computing Machinery, New York, NY, USA, 17–32. doi:10.

1145/3452296.3472937

[35] Aaron Tomb. 2023. *Identifying Specification Problems in Dafny Programs*. https://dafny.org/blog/2023/10/27/proof-dependencies/

[36] Qinshi Wang, Mengying Pan, Shengyi Wang, Ryan Doenges, Lennart Beringer, and Andrew W Appel. 2023. Foundational verification of stateful P4 packet processing. In *International Conference on Interactive Theorem Proving*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 32–1.

[37] Shengyi Wang, Mengying Pan, and Andrew W Appel. 2024. Comprehensive Verification of Packet Processing. *arXiv preprint arXiv:2412.19908* (2024).

[38] Xiwei Wu, Yueyang Feng, Tianyi Huang, Xiaoyang Lu, Shengkai Lin, Lihan Xie, Shizhen Zhao, and Qinxiang Cao. 2025. VEP: A Two-stage Verification Toolchain for Full eBPF Programmability. In *22nd USENIX Symposium on Networked Systems Design and Implementation, NSDI 2025, Philadelphia, PA, USA, April 28-30, 2025*. USENIX Association, 277–299.

[39] Yihan Yang, Haifeng Sun, Antoine Kaufmann, and Jialin Li. 2026. NutCracker: A Compilation Framework for Hybrid DPU Architectures. In *Proceedings of the 21st European Conference on Computer Systems* (McEwan Hall/The University of Edinburgh, Edinburgh, Scotland UK) *(EUROSYS '26)*. Association for Computing Machinery, New York, NY, USA, 1759–1779. doi:10.1145/3767295.3803618

[40] Sophia Yoo, Xiaoqi Chen, and Jennifer Rexford. 2024. SmartCookie: Blocking Large-Scale SYN Floods with a Split-Proxy Defense on Programmable Data Planes. In *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*, Davide Balzarotti and Wenyuan Xu (Eds.). USENIX Association. https://www.usenix.org/conference/usenixsecurity24/presentation/yoo

[41] Liangcheng Yu, John Sonchack, and Vincent Liu. 2022. OrbWeaver: Using IDLE Cycles in Programmable Networks for Opportunistic Coordination. In *19th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2022, Renton, WA, USA, April 4-6, 2022*, Amar Phanishayee and Vyas Sekar (Eds.). USENIX Association, 1195–1212.

[42] Delong Zhang, Chong Ye, and Fei He. 2024. P4Inv: Inferring Packet Invariants for Verification of Stateful P4 Programs. In *IEEE INFOCOM 2024-IEEE Conference on Computer Communications*. IEEE, 2129–2138.

[43] Delong Zhang, Chong Ye, and Fei He. 2025. On Temporal Verification of Stateful P4 Programs. In *22nd USENIX Symposium on Networked Systems Design and Implementation, NSDI 2025, Philadelphia, PA, USA, April 28-30, 2025*. USENIX Association, 219–235.

[44] Hao Zheng, Xin Yan, Wenbo Li, Jiaqi Zheng, Xiaoliang Wang, Qingqing Zhao, Luyou He, Xiaofei Lai, Feng Gao, Fuguang Huang, Wanchun Dou, Guihai Chen, and Chen Tian. 2025. When P4 meets run-to-completion architecture. In *Proceedings of the 22nd USENIX Symposium on Networked Systems Design and Implementation* (Philadelphia, PA, USA) *(NSDI '25)*. USENIX Association, USA, Article 79, 19 pages.

[45] Hesam Zolfaghari, Haseeb Mustafa, and Jari Nurmi. 2021. Run-to-Completion versus Pipelined: The Case of 100 Gbps Packet Parsing. In *2021 IEEE 22nd International Conference on High Performance Switching and Routing (HPSR)*. 1–6. doi:10.1109/HPSR52026.2021.9481797

Appendices are supporting material that has not been peer-reviewed.

A The Dafny-Lucid Translator

```
program    ::= decl*
decl       ::= EventDecl(id, param*)
             | GlobalDecl(id, type, expr)
             | ConstDecl(id, type, value)
             | HandlerDecl(id, param*, stmt*)
             | FunctionDecl(id, type, param*, stmt*)
             | MemopDecl(id, type, param*, stmt*)
             | ParseDecl(id, param*, stmt*)
type       ::= Int(width) | Bool | Event | Void
             | Global(id, args)
value      ::= Int(width, val) | Bool(val)
expr       ::= Var(id) | Val(value)
             | Event(id, expr*) | Op(op, expr*)
stmt       ::= If(expr, stmt*, stmt*)
             | Local(id, type, expr)
             | Assign(id, expr)
             | Return(expr?)
             | Call(id, expr*, id*)
op   ::= Add | Sub | BitOr | BitAnd | BitXor
             | ShiftL | ShiftR | Eq | Neq | Lt | Lte
             | Gt | Gte | Or | And | Not | Neg
             | Cast(width)
param      ::= (id, type)
```

Figure 8: The core of the Lucid-Dafny translator's IR.

Dafny	Lucid
`x mod C` $\mid x : nat\, i, \mathrm{C}{==}2^i$	`x`
`x mod C` $\mid C == 2^i$	`x & C-1`
`x[/ *] C` $\mid C == 2^i$	`x [» «] i`
`x<mod, /, *>C` $\mid C \neq 2^i$	invalid
`x <arith,cmp,bool op> y`	`x <arith,cmp,bool op> y`

Table 3: Rules for arithmetic operation translation.

The Dafny-Lucid translator is a compiler backend for Dafny. It runs after Dafny's frontend type checks and verifies a program, deletes its ghost code and specifications, and translates it into a simpler intermediate representation (IR). The Dafny-Lucid translator converts Dafny's IR into a Lucid IR (Figure 8) and checks for unsupported Dafny constructs in the process. It runs several translation and normalization passes before printing the program as Lucid. The compiler is approximately 4000 lines, implemented itself in Dafny.

As described in the paper, we carefully chose the representation of Lucid in Dafny so that a VeriLucid program can be translated to Lucid without losing structure or readability. Statements, expressions, and types map almost directly, with restrictions. One of the more subtle restrictions is on modulo, multiplication, and division operations. These operations are only supported if they can be reduced to bit-shift or mask operations with a constant second argument. This special case of modulo operations is important because a common use in Dafny is to explicitly represent the rollover that occurs in fixed-width nats for verification, e.g., `z := (x-y) mod T`, where z is a nat bounded from above by T. The compiler recognizes this common case and simply deletes the modulo operation, relying on the rollover semantics of the underlying hardware.

A.1 Translating Classes to Modules

In Dafny, user-defined data structures are commonly abstracted as a class, whereas in Lucid, they are abstracted as modules. The VeriLucid translator converts Dafny classes into Lucid modules as illustrated below.

```
class Obj {
  const i : IObj
  constructor Create() {
   i:=new IObj.Create();
  }
  method foo() {
    i.bar();
  }
}
```

```
module Obj {
  global type t =
    { i : IObj.t }
  constr Create() =
    {i = IObj.Create();}
  fun void foo(t self) {
    IObj.bar(self.i);
  }
}
```

The translation follows four rules:

- Instance variables become fields of a record type `t` representing the object's data.
- Instance methods become static methods with a `self:t` parameter.
- Field accesses are prefixed with `self` (e.g., `i` becomes `self.i`).
- Method calls replace the object with its class name and pass the object data as the first argument (e.g., `i.bar()` becomes `IObj.bar(self.i)`).

B Match-action Tables

A match-action table maps keys to actions that compute over packet headers and metadata. The entries (or rules) in a match action table are managed by an external controller. From the perspective of an event handler, a match-action table is immutable during a single execution, but may change between events.

VeriLucid models Lucid's match-action tables, which follow a correct-by-construction design. Actions are pure functions: they have no side effects and only compute results from input arguments. Tables must have default actions, and all actions in a table must share the same type. This interface prevents a number of subtle bugs by construction compared to P4's match-action tables where keys and actions may reference undefined variables, actions can update non-local variables, and default actions are optional.

Actions. Actions are functions typed `(iarg, arg) -> ret`. `iarg` is the action's *install-time argument*, provided by the control plane when it installs the action and stored in the entry along with the action. `arg` is the action's *lookup-time argument*, provided by the handler that calls the table lookup function. For example, a forwarding table that selects a packet's output port and decrements its ttl may define `fwd` and `drop` actions:

```
1 function fwd(p : port, a : ttl) : (port, ttl) {
2     (p, if a > 0 then a - 1 else 0) }
3 function drop(p : port, a : ttl) : (port, ttl) { (0, a) }
```

Both actions return `(port, ttl)` tuples. The port, `p`, is the install-time argument used by the control plane to set the output port of packets matching each rule; ttl is the lookup-time argument—the current packet's ttl.

Table interface. Match-action tables are declared and initialized in almost the same way as `VarArrays` and user-defined data structures. For example, a forwarding table that uses the above arguments:

```
1 const fwdTbl : Table<key, iarg, arg, ret>
2 constructor Create() {
3   fwdTbl := Table.Create(1024, [fwd, drop], (0, drop)); }
```

`Table` has four type arguments: (1) its key and (2 - 4) its action arguments and return values. The `Table.Create` constructor's arguments are table size, an actions list, and a default (iarg, action) tuple. The difference between `Table` and `VarArrays` initialization is that the `new` keyword is not used before `Table.Create`, because tables are modeled as datatypes rather than classes, for reasons discussed below.
Tables provide one function to use from the data plane, `Lookup`:

```
var (out_port, new_ttl) := tbl.Lookup(ip, ttl);
```

Table model. Since a `Table` is immutable from the data-plane's perspective, we model it as a `datatype` rather than a `class`, as we use for mutable `VarArrays`. Datatypes are immutable and generally easier for verification.

```
datatype Table<Key(==), !IArgs, !Args, Ret> = Table(
  size : nat,
  actions : seq<(IArgs, Args) -> Ret>,
  rules: map<Key, (IArgs, (IArgs, Args) -> Ret)>,
  dflt: (IArgs, (IArgs, Args) -> Ret))
{
  static function Create(size : nat,
    actions : seq<(IArgs, Args) -> Ret>,
    def : (IArgs, (IArgs, Args) -> Ret)) : Table<Key, IArgs, Args, Ret>
  {
    Table(size, actions, map[], def)
  }
  function Lookup(k: Key, a: Args) : Ret {
    var entry := if k in rules then rules[k] else dflt;
    var (param, action) := entry;
    action(param, a)
  }
}
```

The core of the table is a `map` from keys to tuples containing an install-time argument and an action. Since `Table` is a datatype rather than a class, it cannot define constructors or methods, only functions that take (or possible return) `Table` values. Instead of a constructor, we define the `static function Create` that constructs an empty table. The `Lookup` function implicitly takes a table as its first argument, because it is a non-static datatype member. `Lookup` finds the entry in the table matching the key, taking the default entry if there is none, and calls the corresponding action passing it the install-time parameter also retrieved from the entry.

Note that `Table` does not need to define its semantics in its specification, like `VarArray` did. Dafny uses the implementation itself as a specification, which is possible precisely because tables are defined as an immutable datatype with pure functions, rather than a class with methods.

The `(==)` restriction on `(Key(==))` means it must be comparable (i.e., define an equality function), a requirement to use it as a type argument to a map. All `nats` and user-defined `datatypes` are comparable by default. The argument types are non-invariant(`!`), which allows `IArg` and `Arg` to be used as action function arguments.

Specification. To verify properties about programs that use match-action tables, programmers need to specify the configuration (i.e., rules) of a table relevant to the properties they wish to prove. The direct way to do this is by annotating handlers with pre-conditions specifying the assumed table configuration abstractly. For example:

```
method HandlePkt(ip : key, ttl : arg)
  requires ∀x :: tbl.Lookup(x, 0).0 == 0
  ensures ttl == 0 ==> (switch.outputQueue == old(switch.outputQueue))
  {
    var (out_port, new_ttl) := tbl.Lookup(ip, ttl);
    if (out_port != 0) {
      switch.generateOutput({out_port}, PktOut(ip, new_ttl));
    }
  }
```

Here, the `requires` clause specifies that `tbl` is configured to return an output port of 0 whenever it is given a `ttl` of 0, regardless of the `ip` argument. This allows Dafny to prove the ensures clause, which specifies that no outputs are generated for packets with `ttl == 0`, always holds.

Vacuous configurations. A programmer can specify a table's configuration directly as in the example above. However, there is a hazard: a programmer can place contradictory requirements on the table that are *vacuous* and permit the verifier to prove anything. For instance: `tbl.Lookup(1, 0).0 == 0 & tbl.Lookup(1, 0).0 == 1` — a table lookup for `ip` address 1 and `ttl` 0 returns output port of both 0 and 1. Vacuous assumptions allow verifiers to prove *anything*, e.g., `0 == 1`. This is not unique to VeriLucid or even Dafny and there are a number of solutions. First, Dafny has built-in checks to identify contradictory specifications [35]. Second, the programmer can write a `lemma` to prove that a table specification is not vacuous. This typically involves providing a witness or example of a table that satisfies the constraints. Automated generation of such lemmas is an active area of research, e.g., with interactive theorem provers [26] or LLMs [24], that future work can also leverage.

Last, it is also possible to use define helpers that make it impossible to specify contradictory table configurations. For example:

```
ghost function Rule(k: Key, acn: (IArgs, Args) -> Ret, p : IArgs) :
      Table<Key, IArgs, Args, Ret>
requires k !in rules
{
  Table(this.size, this.actions, this.rules[k := (p, acn)], this.dflt)
}
```

When `Rule` is used in a specification, the verifier will fail the specification itself if it accidentally defines two rules that map one value to different outputs. For example, the following will fail verification:

```
requires tbl == Table<...>.Empty().Rule(1, fwd).Rule(1, drop)
```

Translation to Lucid. Translation of table construction and lookup is syntactic, similar to `VarArrays`, because Lucid's tables have the same interface, arguments, and semantics as VeriLucid's model.

C Checking sALU Instruction Compatibility

A `memcalc` function abstracts an instruction for a stateful ALU (sALU), the units in a PISA pipeline that perform atomic updates to state. SALUs have many restrictions that can differ across PISA chips [28]. To check that a `memcalc` respects particular sALU restrictions in VeriLucid, a programmer proves that it is equivalent to an instruction for an sALU model.

VeriLucid's model is a library that represents a conservative subset of the Tofino's sALU capabilities and matches Lucid's restrictions. The full model library is approximately 50 lines of Dafny. Future work may use it as a template for more precise sALU models, or models for different targets.

The sALU instruction model. An sALU instruction executes in two stages. The first stage performs a comparison operation, the second stage executes one of two arithmetic operations depending on the result of the comparison. The comparison operation is optional. The model represents this with `MemInstr`:

```
datatype MemInstr =
  | Ret(ret:ArithExp)
  | Branch(b:CmpExp, btrue:ArithExp, bfalse:ArithExp)
```

`Ret` evaluates an arithmetic expression and returns its result, while `Branch` performs both the comparison and arithmetic operations in each branch.

Arithmetic and comparison units have different operations, so are represented by different nodes. An `ArithExp` is:

```
datatype ArithExp =
 | MemExp(x:MemVal)| ArgExp(y:ArgVal)
 | OpExp(x:MemVal, op:Op, y:ArgVal, const_op: Option<(Op, ConstVal)>)
datatype Op = Plus | Minus // | ...
```

It returns either its memory operand, its argument operand, or an arithmetic operation of the form `MemVal <op> ArgVal (<op> ConstVal)`, where the constant operation is optional.

A `CmpExp` has only one constructor, for an operation of the form `MemVal <cmp> ArgVal (<op> ConstVal)`:

```
datatype CmpExp =
  CmpExp(x:MemVal, cmp:Cmp, y:ArgVal, const_op: Option<(Op, ConstVal)>)
datatype Cmp = Lt | Gt | Eq | Neq | // ...
```

An sALU takes operands from 3 sources: memory (i.e., the array cell being updated), an argument from the caller, or a constant. An instruction (comparison and arithmetic) can only take one operand from each input source. The model represents this with separate `MemVal`, `ArgVal`, and `ConstVal` datatypes:

```
datatype MemVal = MemVal(v:word)
datatype ArgVal = ArgVal(v:word)
datatype ConstVal = ConstVal(v:word)
```

Each datatype represents one operand source. The inner values of all 3 operand source types use the same `word` type.

The model is implemented as an abstract module that is generic to word size, with a concrete module for each sALU word size available in Lucid / Tofino:

```
abstract module SALU {
  type pos = x:nat | x > 0 witness 1
  const MAXVAL: pos
  type word = x:int | 0 <= x < MAXVAL witness 0
  // datatype ArithExp = ...
}
module SALU8 refines SALU { const MAXVAL := 256 }
module SALU16 refines SALU { const MAXVAL := 65536 }
module SALU32 refines SALU { const MAXVAL := 4294967296 }
```

`MAXVAL` is a constant declared, but not defined, in the model's abstract module (`SALU`); `SALU8`, `SALU16`, and `SALU32` are the concrete modules used to model 8-, 16-, and 32-bit sALUs.

Instruction evaluation. The second half of the model defines functions to evaluate an instruction, exactly as one would write in an interpreter. There is an evaluation function for each of the `MemInstr`, `ArithExp`, and `CmpExp` datatypes:

```
function eval(mi:MemInstr): word {
  match mi {
    case Ret(ret)        => evalArith(ret)
    case Branch(b, bt, bf) => if evalTest(b) then evalArith(bt)
                              else evalArith(bf)
}
function evalArith(ae:ArithExp): word {
  match ae {
    case Op(x, op, y, None) =>
      match op {
        case Plus => (x.v + y.v) % MAXVAL
        case Minus => (x.v - y.v) % MAXVAL
      }
    case Op(x, op, y, Some((const_op, c))) =>
      match (op, const_op) {
        case (Plus, Plus) => (x.v + y.v + c.v) % MAXVAL
        case (Plus, Minus) => (x.v + y.v - c.v) % MAXVAL
        case (Minus, Plus) => (x.v - y.v + c.v) % MAXVAL
        case (Minus, Minus) => (x.v - y.v - c.v) % MAXVAL
      }
    case MemExp(x) => x.v
    case ArgExp(y) => y.v
  }
}
function evalCmp(t:CmpExp): bool {
  var rhs:word := match t.const_op {
    case Some((Op, ConstVal)) =>
      match Op {
        case Plus => (t.y.v + ConstVal.v) % MAXVAL
        case Minus => (t.y.v - ConstVal.v) % MAXVAL
      }
    case None => t.y.v
  };
  match t.cmp {
    case Lt => t.x.v < rhs case Gt => t.x.v > rhs
    case Eq => t.x.v == rhs case Neq => t.x.v != rhs
  }
}
```

Using the model. A programmer uses the model to prove that a `memcalc` can be compiled to a valid sALU instruction by: 1) defining an instruction using the instruction datatypes; 2) asserting that the memcalc is functionally equivalent to the result of interpreting the sALU instruction:

```
import opened SALU8
function incr(stored:nat8, added:nat8): nat8
 ensures incr(stored, added) == eval(
 Ret(Op(x:=MemVal(stored),op:=Plus,y:=ArgVal(added),const_op:=None)))
{ (stored + added) % 256 }
```

Limitations. `VeriLucid`'s approach to checking sALU compatibility is flexible and *optional*, so programmers can gradually enforce sALU compatibility across their program as necessary in refactoring to fit PISA constraints. There are a number of limitations of VeriLucid's approach.

- **Manual effort.** VeriLucid's approach to sALU compatibility checking requires programmers to find and sALU instruction that implements a `memcalc` by hand. In our experience, this is not a major burden because by the time a program is being optimized to fit sALU restrictions, thinking about instruction-level constraints is unavoidable. Further, the instruction equivalence proofs can be attached directly to `memcalcs`, as shown above, and verified-compatible `memcalcs` can be implemented as libraries re-used across programs. SALU instructions can be synthesized from higher-level requirements [17], thus we believe future work will be able to synthesize sALU instruction proofs from `memcalcs` automatically.
- **Optional verification.** A fundamental drawback of making verification of `memcalc` compatibility optional is that programmers can try to compile programs with `memcalcs` that are too complicated for the hardware. This is not a major drawback: in the worst case compilation will fail.
- **Portability.** VeriLucid can represent instruction sets from different architectures, but Lucid's static analysis only accepts memory functions that can translate to the Tofino. Future work can explore several ways forward: disabling Lucid's static checks and relying on `VeriLucid` to prove compatibility; extending Lucid to support user-defined syntax-based sALU instruction definitions, like those introduced in VeriLucid; or compiling VeriLucid (or a different subset of Dafny) to other platforms.

D Recirculation Event Interface

In VeriLucid, event recirculation enables asynchronous state updates. The program generates an event to update data structures, which

gets recirculated and interleaved with incoming packet events. For example, a program could count asynchronously:

```
datatype Event =
| Pkt(doCount : bool)
| Count()
const switch : Switch<Event>
const ctr : StateVar<nat32>
method HandlePkt(doCount : bool)
 { if (doCount){ switch.generateRecirc(Count());} }
method HandleCount()
 { ctr.Set(saturatingAdd, 1); }
```

Instead of updating `ctr` directly, `HandlePkt` calls `switch.generateRecirc(event)` to generate a `Count()` event, whose handler performs the update later. The switch model maintains separate queues for recirculation and output events, so the programmer can verify properties about the sizes, arrival times, or even the contents contents of queues. For example:

```
method HandlePkt(doCount : bool)
 ensures doCount ==>
 (|switch.recircQueue| == old(|switch.recircQueue|) + 1) &&
 (switch.recircQueue[|switch.recircQueue|-1].event == Count())
```

This specification on `HandlePkt` ensures that when `doCount` is true, the switch's recirculation queue length increases by 1 and the last event in the recirculation queue is a `Count()` event. Programmers can also prove properties about a program's execution over time externally, from a test case with assertions and loop invariants about queue state.

The VeriLucid switch model also allows programmers to prioritize either input events from the network or recirculated events. When input events are prioritized, the switch will not process a recirculated event until the input queue is empty. This models configurable priorities in the arbiter between a switch's parsers and its match-action pipeline, a feature of PISAs including the Tofino [41].

E Temporal Properties of a MAC Learner

This appendix briefly describes checking several temporal properties of a MAC learner using the switch model. The goal is to connect temporal reasoning, recirculation events, and the VeriLucid switch model. To illustrate the switch model's machinery, we dispatch events and reasons about queues manually in a generalized test. We believe a more elegant version of these properties could be written as a specification using the dispatch method + state invariant pattern describe in Section 5.

The MAC learner has two events: one for packets and a second for new entry installations. The packet event carries its ingress port for illustration purposes; its handler generates an install event when a packet arrives from a previously unseen source MAC address, which recirculates and updates the MAC entries.

```
datatype Event =
  | pkt(src : macAddr, dst : macAddr, ingress_port : port)
  | install(src : macAddr, port : port)
```

There are many temporal properties we may wish to prove. Here, we focus on several related to event ordering. We assume that the switch is configured to prioritize network packets over recirculated events. Our goal is to prove that, when a burst of packets arrives back-to-back at a switch with empty queues, if the first packet's source address has not yet been learned, the switch running the MAC learner will:

(1) process the first packet event and generate a recirculated `install` event while handling it;

```
1  method EventualMACLearn(events : seq<Event>, p : MacLearnerProg)
2    requires |events| > 0
3    requires ∀e :: e in events ==> e.pkt?
4    requires p.switch.validQueue(events)
5    requires p.switch.lastTime == events[0].time
6    requires ∀i :: 0 <= i < |events| ==>
7             events[i].time == events[0].time + i
8    requires p.switch.switchInvariant()
9    requires |p.switch.inputQueue| == 0 && |p.switch.recircQueue| == 0
10   requires p.learned.cells[hash(p.seed, [events[0].src])] == 0
11   requires p.switch.inputFirst
12 {
13   p.switch.simulateArrivals(events);
14   var e := p.switch.pickNextEvent();
15   ghost var fstEvent := e;
16   p.HandlePkt(e.src, e.dst, e.ingress_port);
17   ghost var fstInstall:=install(fstEvent.src, fstEvent.ingress_port);
18   assert p.switch.recircQueue[0] == fstInstall;
19   while |p.switch.inputQueue| > 0
20    decreases |p.switch.inputQueue|
21    invariant p.switch.baseInvariant()
22    invariant |p.switch.recircQueue| > 0
23    invariant p.switch.recircQueue[0] ==
24                install(fstEvent.src, fstEvent.ingress_port)
25    invariant ∀i :: 0 <= i < |p.switch.inputQueue| ==>
26                p.switch.inputQueue[i].pkt?
27   {
28      var e := p.switch.pickNextEvent();
29      match e.event {
30          case pkt(src, dst, ingress_port) =>
31            p.HandlePkt(src, dst, ingress_port);
32      }
33   }
34   e := p.switch.pickNextEvent();
35   assert e.event == fstInstall;
36 }
```

Figure 9: A generalized test method to prove temporal properties of a MAC learner.

(2) process all the remaining packet events; and
(3) process the recirculated `install` event.

Figure 9 shows the generalized test method to prove the above temporal properties. Lines 2—11 set up the assumptions: the input event list is non-empty (2), consists entirely of `pkt` events (3), and is a valid sequence of events to enqueue at the switch (4); the first event arrives at the switch's last event dispatch time (5) and events are spaced 1 time unit apart(6 - 7); the switch's basic internal invariants hold (8) and its input and recirculation queues are empty (9); the cell tracking whether the first packet event has been learned is set to 0/false (10); and finally, the switch is configured to prioritize input events over recirculated events (11).

Line 13 loads the events into the switch model, then lines 14—18 dispatch the appropriate handler manually and prove that the handler has generated a recirculation event to install the first `pkt` event's entry. Next, the while loop manually picks and dispatches all the events from the input queue. The loop's decreases clause (line 20) proves the switch decrements its input queue on every iteration, while its invariants (lines 21—26) maintain that the recirculation queue always begins with the install event generated by the first `pkt` and all events in the input queue are `pkt` events.

After all the input events are drained, lines 34 and 35 assert that the next event to be processed will be the recirculated `install` event.