P4 Language Evolution (draft)

The P4.org language consortium

First version: May 26, 2015  
Revised: June 9, 2015: added hardware/software interface specification

# An Invitation to Contribute

P4 is a language for programming the data plane of network devices. This document creates a discussion framework for evolving P4 from its current version (v.1.0.2), by taking into accounts lessons learned by implementing and deploying the first version of the language. Please contribute to this document by supplying comments, critiques, suggestions and improvements.

# P4 Overview

The P4 programming language has been recently introduced as a high-level domain-specific programming language targeted for programming the forwarding plane of programmable packet processors. The intent of P4 is to cover a broad range of targets, from software switches through FPGAs, NPUs and reconfigurable hardware switches. The language specification and a reference implementation, including several P4 programs have been published using an open-source license at http://P4.org. The current version of P4 is 1.0.2.

The P4 language is protocol independent: it allows programmers to express a rich set of data plane behaviors and protocols. The core P4 abstractions are:

* **Header definitions** describe the format (the set of fields and their sizes) of each header within a packet.
* **Parse graphs** describe the permitted header sequences within received packets and the headers and fields to extract from packets.
* **Tables** associate keys to actions. P4 tables generalize traditional switch tables; they can be used to implement routing tables, flow lookup tables, access-control lists, or other user-defined table types.
* **Actions** describe how packet header fields and metadata are manipulated.
* **Match-action units** stitch together tables and actions, and perform the following sequence of operations:
  + Construct lookup keys from packet fields or computed metadata,
  + Use the constructed lookup key to index into tables, choosing an action to execute,
  + Finally, execute the selected action.
* **Control flow** is expressed as an imperative program describing the data-dependent packet processing within a switch pipeline, including the data-dependent sequence of match-action unit invocations.

P4 is not a Turing-complete language. In fact, (assuming a unit cost for a table lookup operation), all P4 programs should provably execute a constant O(1) number of operations for each byte of an input packet processed (i.e. each header byte analyzed). In other words, the computational complexity of a P4 program depends linearly only on the header sizes, and never depends on the size of the state accumulated while processing data (e.g., the number of flows, or the total number of packets processed). These guarantees are necessary (but not sufficient) for enabling fast packet processing.

# Evolving P4

## Good features of P4

Based on the accumulated experience with P4 so far, the consensus is that it strikes a good balance between (1) providing a high-level abstract view of the programmed device, and (2) making it possible to write precise programs describing a wide variety of data-plane protocols. More specifically, P4 exhibits the following good features:

* The core abstractions in P4 are familiar to network programmers and operators. New users quickly learn to write programs in P4.
* Physical resources are abstracted using symbolic terms; in P4 there are no references to registers, byte offsets in packets, memories, or crossbars. Programs are written in terms of data structures, table look-ups and actions; a compiler allocates low-level resources.
* Despite the high-level abstract forwarding model, the programmer can accurately control how data is laid out in packets, for example using bit-level manipulations.
* P4 is composed of several separate sub-languages, tailored for different network functions: the parser language is based on *types* for describing packet layout and a state-machine model for describing protocol sequences. The match-action language, together with imperative control-flow, describes how packet fields are manipulated. The two languages are tied together by data types (the “metadata” in P4 terminology) that enable parsers and match-action units to communicate.
* P4 provides a restricted computational model. There are no loops, no recursive computations (except recursive parser state-machines), no pointers, no floating-point. These restrictions are designed to ensure that high-performance implementations of P4 are feasible (targeting multi-terabit/second devices).
* P4 describes the processing of each network packet in isolation. There are no constructs for modeling concurrency, scheduling, arbitration, locking or multi-threading. However, note that this does not preclude highly concurrent implementations.
* P4 primarily aims to describe how packets are forwarded: it does not attempt to model complex stateful packet-processing, such as control-plane protocols (e.g. routing protocols). P4 does not attempt to provide a complete model of a switch: P4 assumes the table entries are managed by an external control plane, and data (including complete packets) can be exchanged with the control-plane. P4 is designed to provide essentially a description of the “fast-path” data path of packet processing, but not a universal language for manipulating packets. P4 does not provide many facilities for deep packet inspection.

## Shortcomings of P4 v1

The process of writing complex P4 programs has uncovered several shortcomings in the P4 v1 language definition. In our opinion, there is one major shortcoming, and several relatively minor ones. (A shortcoming is defined “minor” if it can be addressed by relatively incremental changes to the language; fixing some minor shortcomings can have significant impact on the ease-of-use of the language.) The major shortcoming is that the P4 language mandates a specific abstract forwarding model, which in turn assumes a specific target architecture:

* **(Major)** The P4 language specification defines an abstract switch architecture model, composed of a parser, and two match-action pipelines (ingress and egress). It is arguable that embedding this model into the language definition is a barrier to widespread adoption of P4 on platforms with different architectures. This point is expanded in the next section.
* **(Minor)** P4 contains a large number of primitive actions; in contrast, it does not contain an assignment operator or a simple expression language (e.g. a plus operator). Many of the primitive actions could be modeled more concisely and as precisely using a simpler traditional expression language (e.g., in the add\_field action it is not obvious without reading the documentation which of the arguments is the destination).
* **(Minor)** P4 has no support for writing modular programs. All P4 variables have global scope, and there is no lexical scoping, functions, local variables, or other information-hiding mechanisms. This makes writing large programs difficult. There are no mechanisms for extending programs or contributing program fragments as libraries.
* **(Minor)** The control-flow constructs in P4 are limited to if/then/else; there is no exceptional exit, no break/return statement. Coupled with the absence of local variables this makes it tedious to handle complex control-flow patterns.
* **(Minor)** The language has a large number of keywords (more than 60) and a large number of built-in actions. One can argue that many of these constructs belong to libraries rather than to the core language (e.g., a library of checksum algorithms).
* **(Minor)** The language assumes that when an action is evaluated, all the component primitive actions execute concurrently. This precludes writing actions that are composed from sequential operations, (e.g., appending a value to a header stack, which entails incrementing a position and writing a value at the resulting index). Moreover, concurrent semantics are harder for traditional programmers to understand.
* **(Minor)** The language does not clearly delineate resource *declaration* from resource *instantiation*. For example, what happens when a table is accessed twice in a match-action pipeline: is such a program illegal, is it implemented using a single table or two separate tables, or is this choice target-dependent?
* **(Minor)** The semantics of P4 are not always completely specified, for example:
  + Bit-manipulation operations: what happens when decrementing a TTL value that is zero? What happens when adding a 16-bit signed and a 32-bit unsigned values? How can the user enforce type conversion using casts?
  + The semantics of operations acting on header stacks is incompletely specified. Can a header stack contain “holes”? The handling of header stacks within parsers is missing from the specification.
  + The semantics of uninitialized values is under-specified (what is the value obtained when reading an uninitialized variable?). Related to this is the semantics of using in computations fields from headers with valid bits not set.
  + The behavior of the program under runtime exceptions is unspecified. How are the parser exceptions exposed to the match-action pipeline (some of this information is marked as TBD in the P4 specification)? How can the parser program verify certain expected invariants (e.g., the fact that the IPv4 IHL field has a value larger than 5)? How can user programs write exception handlers?
  + The deparser cannot be specified explicitly. The P4 specification mandates deparsers to be constructed starting from the parser specification. However, this assumes that the exact same protocols are present in all input and output packets. Moreover, some parser graph topologies (e.g., including cycles) do not lead to a precise definition of the obtained deparser behavior.
  + The semantics of accessing the intrinsic metadata is under-specified or unclear (see for example the specification of egress\_spec). What happens if some egress metadata is accessed in the ingress pipeline?
  + The semantics of some actions is unclear or overly complex. (See for example the specification of the drop action.)

## Desirable features for P4

Before revising the language, the P4 community should agree on a set of goals. To initiate the process, this document propose the following set of goals for a revised version:

* Make P4 a simple and clear language. There should be no room for ambiguities or constructs with overly complex semantics. Ideally, semantics should be compositional: the meaning of a well-formed program fragment should depend only on the program fragment (contrast this to the semantics of the “drop” action in P4 v1, which depends on the entire program: even after the drop action is invoked, the packet continues to flow through the ingress pipeline and its processing could cause side-effects).
* Completely specify the language. Given a P4 program, the entire contents of the tables, and a set of input bits in a packet, the value of the bits in the program output (including output packets) should be completely defined. This means providing clear operational semantics for all P4 constructs.
* The language should be strongly-typed. No operations between incompatible types should be allowed. A program that type-checks should never produce type-related runtime errors.
* The language should provide error-handling constructs.
* The language should be as small as possible. In general, any language can evolve only by growing, and not by shrinking, because it is very difficult to remove features when legacy software depends on them. New features should only be added to the core language when they are useful across a wide variety of targets. Ideally all changes that are target-specific should be confined to target-specific libraries, and should not require amending the language definition.
* The language should provide some extensibility mechanisms without requiring extensions to the core language. (Similar to pragmas, Java annotations or C# attributes.)
* Finally, the scope of P4 should be expanded to cover a large variety of packet-processing engines with a broad range of architectures.
* The P4 language should maintain the “look and feel” of P4 v1 as much as possible. It should rely on the same core abstractions described in the overview section.

## Broadening P4’s scope

One important observation is that P4 is actually much more than just a switch datapath description language. The P4 v1 specification mandates that P4 be used to program switches implementing a fixed abstract forwarding model (Figure 1 in the P4 v1 specification).

Several problems are caused by baking the forwarding model into the language specification:

* Architectures that cannot emulate this model cannot be targeted by P4. For example, a packet-filter cannot be mapped to the P4 v1 abstract forwarding model.
* Architectures cannot expose additional features (e.g., multiple parallel processing pipelines). This makes the abstract switch model a “lowest-common denominator” architecture, which does not encourage switch manufacturers to differentiate by exposing additional functionality.
* Evolving the abstract forwarding model entails making changes to the language. This aspect is plainly visible in the P4 v1 specification, which contains a large number of constructs that were added to enhance the original forwarding model, which was deemed to be too restrictive. Such constructs include the rich set of “standard intrinsic metadata”, but also constructs such as field\_list\_calculation, counters, meters, registers, a large number of primitive actions (cloning, recirculation, resubmission, digest generation, set\_field\_to\_hash\_index), action profiles, mirroring, etc.

All these tensions, which pull the language in opposite directions, can be resolved by cutting the Gordian knot that unites the architecture with the language.

This proposal argues for broadening the definition of the P4: P4 is a language that targets the data place of general ***Packet Processing Engines*** (PPEs). Such engines include, but are not restricted to, programmable switches.

Second, this proposal argues for completely removing the abstract switch model from the P4 specification. P4 should be able to target a wide variety of PPE models. Note that this does not preclude the community from adopting one (or a small number of) standard PPE models. For example, the standard switch architecture from P4 v1 could become a baseline PPE models, allowing programmers to write portable code among switches that support the baseline model. However, the specification of the PPE models should be separate from the language specification. Most importantly, the P4 language should evolve separately from the PPE specifications; *no additions or changes to P4 should be needed when a PPE specification changes*.

## Decoupling P4 from the switch architecture

Figure 1 is an abstract view of how P4 might interact with the data-plane of a packet-processing engine. The PPE data plane is a fixed-function device (shown in cyan) that provides several programmable “holes” (shown in white). The user writes a P4 program to specify the behavior of each hole.. The PPE manufacturer describes the ***interfaces*** between the white blocks and the surrounding fixed-function blocks. These interfaces are target-specific.

Note that the fixed-function part can be software, hardware, firmware, or a mixture of all of the above. The programmable parts can be implemented by interpreting, or running compiled versions of P4 programs.

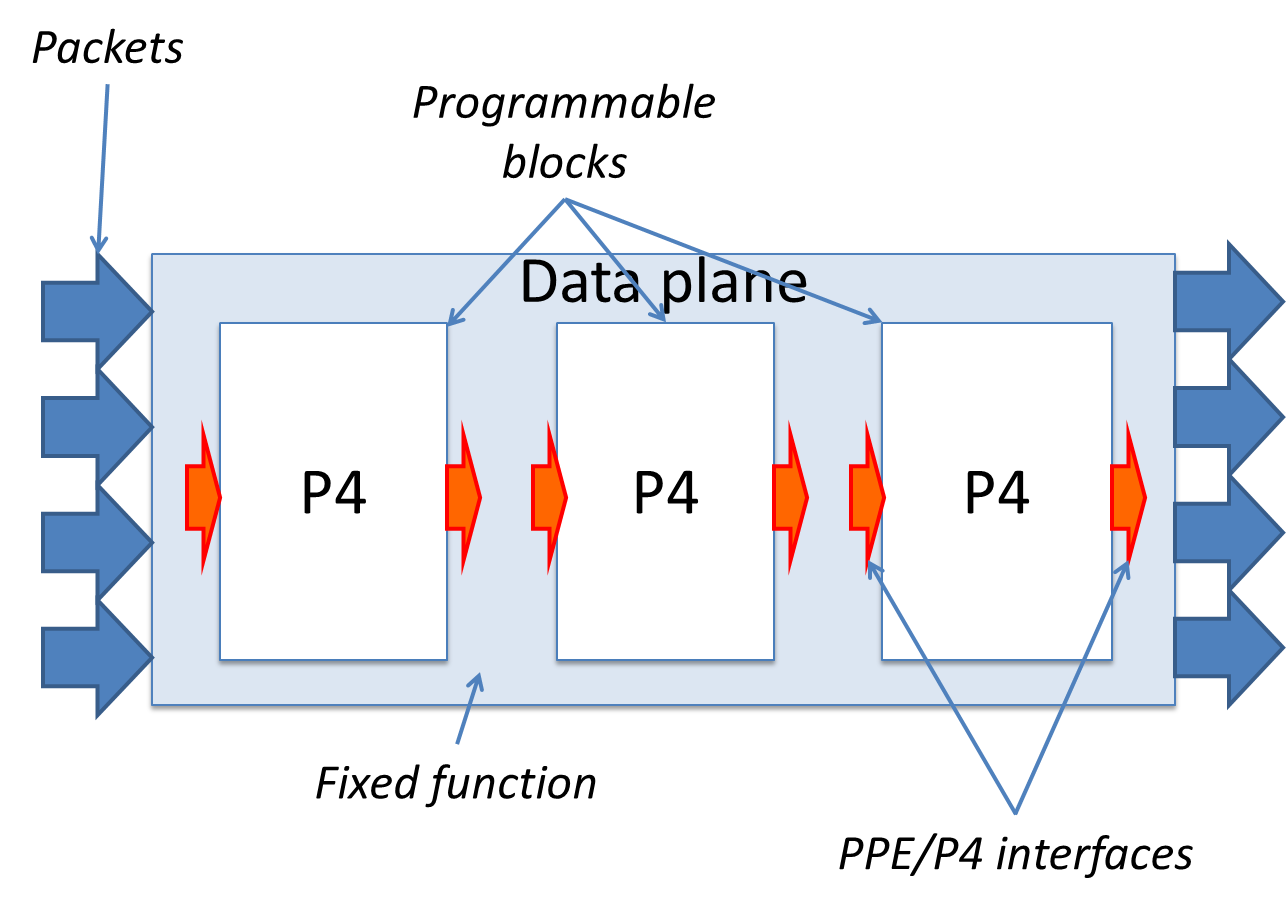


Figure : Generic abstract packet-processing engine (PPE) programmable in P4.   
The white blocks are programmable in P4. The blue box is fixed-function logic.

The interfaces allow user-defined data structures to be passed between the programmable blocks, as illustrated in Figure 2. Some of the data is passed unchanged between programmable blocks (using general-purpose registers[[1]](#footnote-1)). Each programmable P4 block also interacts with control registers at its input and output. The input control registers describe signals that are passed from the fixed-function blocks to the programmable blocks, while the output control registers are used by the programmable blocks to elicit behaviors from the fixed-function blocks.

Examples of input control signals could be: the current timestamp, the input port, metadata associated with a packet received from the control plane, the size of the queues a packet has traversed, error signals received from hardware. Examples of output control signals could be: the output port that is the destination of a packet, indications to drop a packet, to replicate a packet, to multicast a packet, packet priorities to use for various arbiters following the block, signals to recirculate a packet, to mirror a packet, etc.

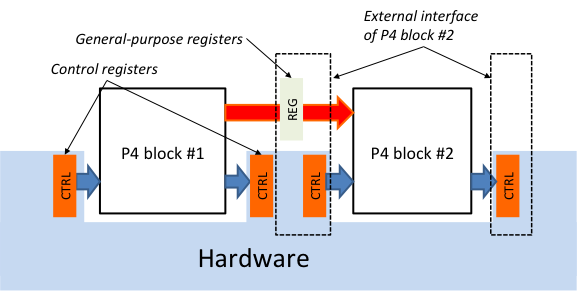


Figure : Interfaces between blocks should include user-defined data and control signals.

The semantics of each P4 program is completely described as a set of transformations mapping the input structures from the input registers to the output structures stored in output registers. Only *the PPE architecture attaches a meaning to the bits in a control register.* For example, in order to cause a network packet to be forwarded on a specific output port, a P4 program may need to write the output port index into a dedicated control register. Similarly, in order to cause a packet to be dropped, a P4 program may need to set a “drop” bit into another dedicated control register.

P4 programs should be able to invoke the services of fixed-function blocks (e.g., checksum units, meters, counters, etc.), as illustrated in Figure 3.

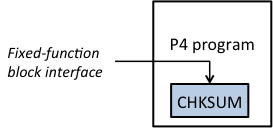


Figure : Invoking services of fixed-function blocks from P4 programs.

The hardware-software interface between P4 and the PPE is exposed through an abstract ***PPE architecture model,*** which identifies the PPE’s P4-programmable blocks and their data interfaces (e.g., parser, ingress pipeline, egress pipeline, deparser, etc.). The PPE architecture can be thought of as a contract between hardware and software equivalent to the Instruction Set Architecture (ISA) of a traditional microprocessor. Each PPE manufacturer must provide a PPE architecture model for their target, and a P4 compiler for that architecture model. It is likely that each manufacturer will define PPE architecture models specific to their target (this is illustrated in Figure 4). The PPE architecture model does **not** have to expose the entire controllable surface of the real data plane– a manufacturer may even choose to provide multiple architectures for one hardware device, each with different capabilities (e.g., with or without support for multicast).

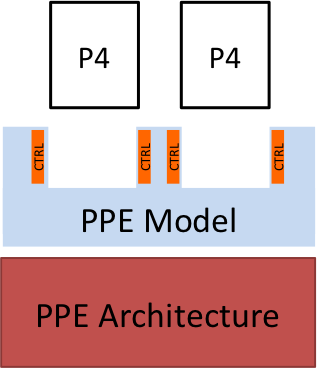


Figure : Relationship between architecture, model and P4.   
The model exposes an abstract view of the architecture,  
 describing the programmable surfaces.

The PPE hardware-software interface should be contrasted with a traditional ISA, such as x86. x86 programs can use general-purpose registers in a manner that is transparent to the hardware. However, to interact with hardware, x86 programs manipulate control registers directly or indirectly: e.g., segment registers, page table entries, and I/O registers. Usually user-level programs are insulated from the interaction with the hardware by software layers, such as the operating system or system libraries.

## Example architectures

To illustrate the benefits of decoupling the switch architecture model from the language specification this section sketches several scenarios that become feasible and which are impossible using the P4 v.1 specification.

### The P4 v1 packet switch

The current switch model from the P4 v1 specification could be defined as a “legacy” PPE architecture, enabling easy portability of programs written in the first version of the language. Applications written for this PPE model would run on all PPE architectures that support the legacy model.

### A packet filter

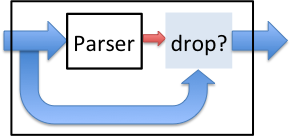


Figure : A packet filter PPE model.   
The parser computes a Boolean value, which   
is used to decide whether the packet is dropped.

A very weak PPE model is a packet filter based only on header data (no look-up tables needed). As shown in Figure 5, such a packet filter can be modeled as a P4 parser block computing a Boolean value (instead of a header vector).

This model could be used to program packet filters running in the Linux kernel. For example, P4 could be used to replace the TCP dump language with the much more powerful P4 language; P4 can support seamlessly new protocols, while providing complete “type safety” during packet processing. For such a target the P4 compiler could generate an EBPF (Extended Berkeley Packet Filter) program, which is injected by the TCP dump utility into the Linux kernel, and executed by the EBPF kernel JIT compiler/runtime.

In this case the target is the Linux kernel, and the PPE abstract model is a packet filter.

(Note that this simple packet filter PPE model of the Linux kernel can be generalized in many ways, by using richer PPE models for the Linux kernel; for example, P4 could subsume many forwarding functions in the Linux kernel using a suitable forwarding PPE model of the Linux kernel, that allows P4 code to be hooked in the right places in the packet processing datapath.)

### A complex switch

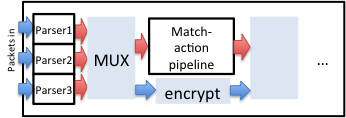


Figure : A complex switch with multiple parsers and a payload encryption engine.

Our last example in Figure 6 sketches a complex PPE architecture, which provides one parser for each input port (to allow for increased parsing bandwidth) and also provides a packet payload encryption engine. The architecture exposes multiple programmable surfaces interconnected in a complex topology.

## P4 program portability

A major downside of decoupling the architecture model from the language definition is that P4 programs are no longer guaranteed to be portable. It is arguable that this trade-off is an improvement, and will prove healthy for the long-term evolution of P4.

In the revised P4 model, P4 programs are no longer the same across all targets. Instead, P4 programs written for a given PPE architecture model are portable across all targets that faithfully implement said architecture model (assuming that enough resources are available to implement the program). In general P4 programs are not expected to be portable across different PPE architecture models. For example, executing a P4 program that controls packet broadcast by writing into a custom control register will not work on a target that provides no such control register.

The behavior of P4 programs should be defined using an abstract machine with infinite resources. However, each target has a limited set of resources and capabilities. Rather than being the least-common denominator supported by all platforms, P4 contains powerful constructs that may be unavailable on some targets. *It is to be expected that not all P4 programs can be supported by all targets.* Particular targets may not support fully some P4 language constructs (for example, some targets may not support arbitrary-length stacked protocol headers). Ideally the restrictions on the P4 language imposed by a specific target should be clearly documented by the target manufacturers; at the very least, restrictions have to be conveyed to P4 programmers using clear compiler error messages when attempting to compile programs that use unsupported features. However, on most targets, P4 programs will have an escape route for processing complex packets: they can delegate the processing of such packets to the switch control plane (i.e., the “slow path”).

***P4 conformance*** of a target is defined as follows: if a specific target T supports only a subset of the P4 programming language (let’s call it P4T), the programs written in P4T executed on the target should provide the exact same behavior as P4 programs executed on an abstract machine with infinite resources.

With these caveats, compared to the state-of-the-art method of programming microcode on top of custom hardware, P4 provides significant value:

* **Flexibility**: P4 makes many packet-forwarding policies expressible as programs; contrast this to the fixed-function forwarding engines in traditional network forwarding elements.
* **Expressiveness:** P4 programs may express sophisticated hardware-independent packet processing algorithms using solely general-purpose registers and table look-ups. Such programs will be portable between hardware architectures that implement similar PPE architecture models.
* **Resource management:** P4 programs express resource usage in symbolic terms (e.g., IPv4 source address); compilers map such user-defined fields to available hardware resources (e.g., stage5.register128) and manage resource allocation and scheduling.
* **Software engineering:** P4 programs provide important benefits such as type checking, information hiding and software reuse.
* **Standard libraries:** Manufacturer-supplied standard libraries may be used to wrap target-specific functions into portable high-level P4 constructs.
* **Decouple hardware and software evolution:** target manufacturers may use abstract PPE architectures to further decouple the evolution of low-level architectural detail from high-level processing.
* **Debugging:** Manufacturers can provide their customers software models of PPE architectures to aid in the development and debugging of P4 programs.

## P4 Evolution Roadmap

This document is not yet proposing a language to describe the PPE architecture model. It is clear that such a language is highly desirable, but the space of PPE models is not yet well-enough understood to standardize a language. Moreover, previous research in this direction indicates that such a language can be significantly more complex than P4 itself.

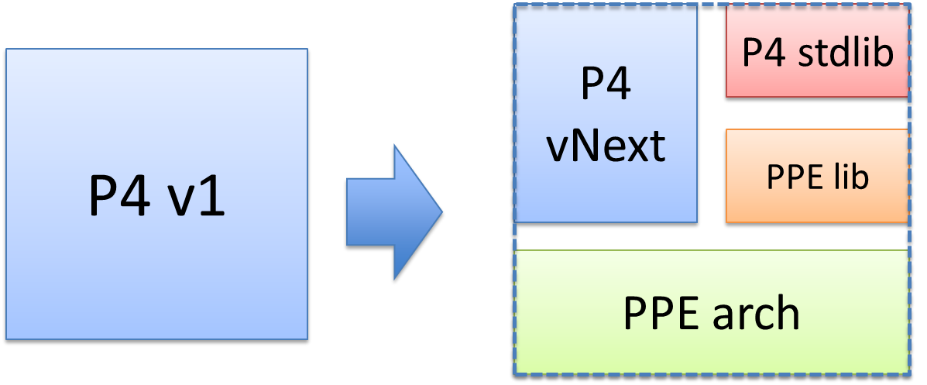


Figure : P4 language evolution roadmap

Figure 7 illustrates the core of our proposal for evolving P4:

* The core language elements should be kept in the P4 definition (parsers, deparsers, match-action units, actions, control-flow, headers, etc.).
* The PPE architecture specification should be relegated to a separate document. One can foresee a broad ecosystem of PPE architectures, provided by vendors, researchers and other interested parties. This ecosystem should evolve independently from the P4 language definition.
* Some elements should be migrated into a standard library. Possible examples include: standard error codes, standard interfaces (e.g., a network packet), standard utility functions, standard checksum computation algorithms.
* Some elements should be migrated into target-specific libraries. Examples include: custom tables (e.g., hash-tables, TCAMs, tries – all these table implementations provide the same service while having widely different implementations and parameters), custom accelerators (e.g., encryption engines), custom target-specific actions (e.g., broadcast) The boundary between the standard library and the target-specific libraries is not clear-cut; elements may migrate into the standard library if they have broad adoption across many PPE architectures.

# Specifying the hardware-software interfaces

We propose extending P4 with the following syntactic constructs:

* **Prototypes**: Constructs similar to C function prototypes are used to describe programmable block interfaces.
* **Type variables**: to indicate user-defined types in programmable blocks.
* **Implementations**: Constructs similar to C functions are used to describe the programmable block implementations.

A PPE program is thus composed of two separate parts:

1. The PPE declaration, written by the PPE manufacturer (the equivalent of “header” files), consisting of a series of prototypes.
2. The implementation of the PPE dataplane, written by the PPE user, consisting of a series of implementations and **bindings** of implementations to the prototypes.

## Prototypes

(Proposed P4 keywords shown in **bold**.)

To describe a functional block that can be programmed in P4 the PPE manufacturer provides a PPE prototype. For example, the PPE manufacturer could write a declaration as follows:

**pipeline** matchActionPipe<H>(**in** **bit**<4> inputPort,   
 **inout** H parsedHeaders,   
 **out** **bit**<4> outputPort);

This declaration describes a type of programmable block named matchActionPipe that performs match-action processing (shown by the **pipeline** P4 reserved keyword). The hardware-software interface between the matchActionPipe block and the surrounding hardware can be read from this declaration:

* The syntax **bit**<4> is a type indicating a 4-bit value,
* The in, inout, and out keywords indicate the direction of parameters
* Input #1: is a 4-bit value named inputPort, received in a read-only (**in**) control register
* Input #2: an object of type H, named parsedHeaders.It is both an input and an output, shown by **inout**.
* The type H is given by a type variable <H> in the declaration (the syntax is similar to Java). The type H indicates a type that will be defined later by the PPE programmer. Since the type is a variable, the value parsedHeaders must be stored in general-purpose registers.
* Output #1: the same parsedHeaders is and output, also stored in general-purpose registers. The user mutates this value in-place in the pipeline implementation, and the value of the parsedHeaders at the completion of pipe is the result of the computation.
* Output #2: a four-bit value named outputPort. This value is written into a control register.

We envision at least four different kinds of prototypes, corresponding to the core P4 constructs:

* parser prototypes
* match-action pipeline prototypes
* deparser prototypes
* a module block, which can be used to tie together multiple prototypes

Only the prototypes can use type variables. When PPE users write implementation programs, they can only use concrete types. So the type variables are only used in very restricted ways – e.g., they are not equivalent to generic types as in Java. Type variables are only a mechanism used to defer the declaration of a type until a later time.

### Implementations

To program a PPE, users have to provide *implementations* for all manufacturer-declared prototypes.

For example, a PPE user can implement a concrete version of the matchActionPipeline by writing the following code:

**header** Header\_h { ... } // header declarations

**pipeline** pipeImplementation(**in** **bit**<4> inputPort,  
 **inout** Header\_h parsedHeaders,  
 **out** **bit**<4> outputPort)  
{  
 // concrete P4 pipeline code here  
}

The user can the **bind** the declaration with the implementation, e.g. using a syntax such as:

matchActionPipe = pipeImplementation;

The binding step is necessary because a complex PPE model may expose multiple programmable blocks with similar interfaces (e.g., the switch with many parsers from Figure 6); the binding makes it clear how implementations are associated to declarations. An implementation may be used for multiple declarations with matching signatures.

### Modularity

Each separate PPE implementation module behaves in many respects like a function in a traditional programming language. For example, the parameters inputPort, parsedHeaders, and outputPort are only accessible locally, within the pipeImplementation block. This mechanism provides support for writing modular PPE specifications and implementations. Each programmable block becomes a separate “function” in the implementatation.

## A Complete model example declaration

To make the discussion concrete we give in this section a model for a simple switch, which we call the “Simple Switch” (SS). Figure 8 is a diagram of this architecture model. SS is a simplified version of the P4 v1 Standard architecture (it only has an ingress pipeline, it omits most of the intrinsic metadata and some packet data paths, such as cloning, resubmission, mirroring, etc.).

SS receives the packet through one of 8 input Ethernet ports, or through a recirculation channel. SS has one single parser, feeding into a single match-action pipeline, which feeds into a single deparser. After exiting the deparser, packets are emitted through one of 8 output Ethernet ports or one of 3 “special” ports:

* Packets sent to the “CPU port” are sent to the control plane
* Packets sent to the “Drop port” are discarded
* Packets sent to the “Recirculate port” are re-injected in the switch through a special port

Packets may be received from one of three sources:

* One of 8 input ports
* Through recirculation
* From the control plane CPU

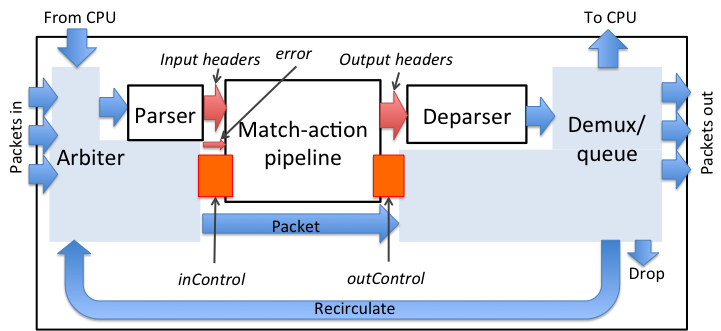


Figure : The Simple Switch (SS) architecture model

The white blocks in the figure are programmable, and the user must provide a corresponding P4 module to drive each such block. The cyan blocks are fixed-function. The orange blocks are hardware-software interfaces used to convey information between the fixed-function blocks and the programmable blocks. The red arrows show the flow of user-defined data.

### Simple Switch architecture declaration

The following P4 program provides a complete declaration of SS. The declaration contains several type declarations, constants, and finally declarations for the programmable blocks. P4 keywords are shown in bold. The programmable blocks are described by function prototypes.

// File “simple.p4”  
 // Simple switch P4 declaration

/\* Various constants and structure declarations \*/

/\* ports are represented using 4-bit values \*/  
 **typedef** bit<4> PortId;

/\* only 8 ports are “real” \*/  
 **const** PortId REAL\_PORT\_COUNT = 4:8;

/\* metadata accompanying an input packet \*/  
 **struct** InControl {  
 PortId inputPort;  
 }

/\* special input port values \*/  
 **const** PortId RECIRCULATE\_INPUT\_PORT = 4:0xD;  
 **const** PortId CPU\_INPUT\_PORT = 4:0xE;

/\* metadata that must be computed for outgoing packets \*/  
 **struct** OutControl {  
 PortId outputPort;  
 }

/\* special output port values for outgoing packet \*/  
 **const** PortId DROP\_PORT = 4:0xF;  
 **const** PortId CPU\_OUT\_PORT = 4:0xE;  
 **const** PortId RECIRCULATE\_OUT\_PORT = 4:0xD;

/\* Prototypes for all programmable blocks \*/

/\*\*  
 \* Switch parser.  
 \* @param <H> type of headers; defined by user  
 \* @param b input packet  
 \* @param parserHeaders headers constructed by parser  
 **/  
 parser** Parser<H>(**in** packet\_in b, **out** H parsedHeaders);

**/**\*\*  
 \* Match-action pipeline  
 \* @param <H> type of input and output headers   
 \* @param headers headers received from the parser  
 \* and sent to the deparser  
 \* @param parseError error that may have surfaced during parsing  
 \* @param inCtrl metadata accompanying input packet  
 \* @param outCtrl metadata for output packet  
 \*/  
 **pipeline** MAP<H>(**inout** H headers,  
 **in** error parseError, // parser error  
 **in** InControl inCtrl, // input port  
 **out** OutControl outCtrl); // output port

/\*\*  
 \* Switch deparser.  
 \* @param <H> type of headers; defined by user  
 \* @param b output packet  
 \* @param outputHeaders headers for output packet  
 **/**   
 **deparser** Deparser<H>(**in** H outputHeaders,   
 **in** packet\_out b);

Let us describe some of these elements:

* The syntax 4:0xF indicates the value 15 represented using 4 bits. An alternative notation is 4:15.
* Next follows the declaration of a parser:   
  **parser** Parser<H>(**in** packet\_in b, **out** H parsedHeaders);  
  This declaration describes a type of parsers. An implementation of such a parser will have to be provided by the user. The parser reads its input (**in** keyword) from a packet\_in, a pre-defined P4 abstraction that represents an incoming network packet. The parser writes its output (the **out** keyword) into the parsedHeaders argument. The type of this argument is H, yet unknown – it will also be provided by the user.
* The declaration   
  **pipeline** MAP(**inout** H headers,   
   **in** error parseError,   
   **in** InControl inCtrl,   
   **out** OutControl outCtrl);  
  describes the interface of a Match-Action pipeline named MAP.   
  The pipeline receives 3 inputs: the headers headers, a parser error parseError, and the inCtrl control data. Figure 8 indicates the different sources of these pieces of information.   
  The pipeline writes its outputs into outCtrl, and it must update in place the headers to be consumed by the deparser.
* In this program the inCtrl and outCtrl structures represent control registers. The contents of the headers structure is stored in general-purpose registers.

1. Note that here “register” implies a PPE storage element, which is different from the P4 v1.x notion of a “register”. [↑](#footnote-ref-1)