The P4 Language Specification

Version 1.0.5

Спецификация языка P4, версия 1.0.5

May 31, 2018

The P4 Language Consortium

Оглавление

1 Введение

P4 - это декларативный язык описания обработки пакетов в сетевых элементах пересылки, таких как коммутаторы, сетевые адаптеры (NIC), маршрутизаторы и др. Язык основан на абстрактной модели пересылки, включающей синтаксический анализатор (parser) и набор табличных ресурсов «сопоставление-действие» (СД, match+action), разделённых на входной и выходной конвейер. Анализатор идентифицирует заголовки, имеющиеся в каждом входящем пакете. Каждая таблица СД выполняет поиск по подмножеству полей заголовка и применяет действия (action), соответствующие первой подходящей (совпадающей) записи в таблице. Графическое представление этой модели дано на рисунке 1.

P4 сам по себе не зависит от протокола, но позволяет задавать протоколы плоскости пересылки. Программа P4 задаёт для каждого элемента пересылки:

- определения заголовков - формат (набор полей с их размерами) каждого заголовка в пакете;
- граф анализа - разрешённые в пакетах последовательности заголовков;
- определения таблиц - тип выполняемого поиска, входные поля для применения, выполняемые действия, размер каждой таблицы;
- определения действий - композитные операции, состоящие из примитивов действий;
- схему конвейера и поток управления - набор таблиц в конвейере и порядок прохождения пакетов через конвейер.

P4 настраивает конфигурацию элементов пересылки. После настройки таблицы могут заполняться правилами обработки пакетов. Такие операции называются в этом документе операциями в процессе работы (run time). Эти операции не препятствуют изменению конфигурации работающего устройства.

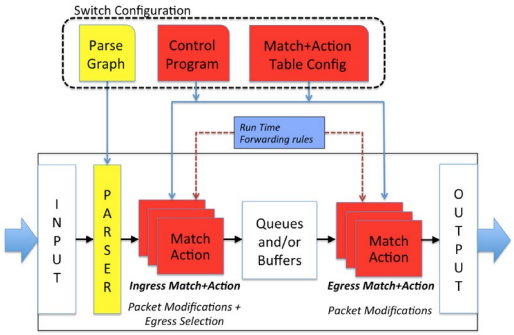

Рисунок 1. Абстрактная модель пересылки.

1.1 Абстрактная модель P4

На рисунке 1 представлена высокоуровневая абстрактная модель P4. Язык P4 работает лишь с небольшим числом простых правил.

- Для каждого пакета анализатор формирует разобранное представление (Parsed Representation), с которым работают таблицы СД.
- Таблицы СД во входном конвейере (Ingress Pipeline) создают выходную спецификацию (Egress Specification), определяющую набор портов (и число экземпляров пакета для каждого порта), куда нужно переслать пакет.
- Блок очередей (Queuing Mechanism) обрабатывает выходную спецификацию, создавая требуемые экземпляры пакета и представляя каждый из них выходному конвейеру (Egress Pipeline). Выходные очереди могут буферизовать пакеты при перегрузке (over-subscription) выходного порта, хотя P4 этого не требует.
- Физический адресат (порт) экземпляра пакета определяется до попадания пакета в выходной контроллер. После передачи пакета в Egress Pipeline этот адресат не может быть сменен (хотя возможно отбрасывание пакета и дополнительные изменения заголовков).
- По завершении обработки в выходном конвейере формируется заголовок экземпляра на основе Parsed Representation (с изменениями в процессе обработки СД) и полученный пакет передаётся в порт.

P4 поддерживает рециркуляцию и клонирование пакетов, которые на рисунке 1 не показаны (см. раздел 14 Рециркуляция и клонирование).

Язык P4 сосредоточен на задании анализатора, таблиц СД и управлении потоком пакетов через конвейеры. Программисты задают эти операции в программах P4, которые определяют конфигурацию коммутатора, как показано на рисунке 1.

Машина, на которой может работать программа P4, называется целевой платформой (target). Хотя платформа может напрямую выполнять программу P4, в документе предполагается компиляция программы в соответствующую конфигурацию платформы. В текущей версии P4 не раскрывает, например, функциональности блока очередей (Queuing Mechanism) и не задаёт семантику Egress Specification сверх упомянутой выше. В настоящее время они считаются зависящими от платформы, указанной на входе компилятора, и раскрываются вместе с другими интерфейсами, которые обеспечивают настройку конфигурации и управление в процессе работы системы. В будущих версиях P4 спецификация этих механизмов может быть раскрыта, что позволит согласованно управлять этими ресурсами из программ P4.

1.2 Пример mTag

В первой статье о P4 [1] содержится пример, названный mTag. В этой спецификации данный пример используется для разъяснения базовых функций языка. Полный код этого примера, включая run-time API, доступен на сайте P4 [2].

> Рассмотрим пример сети L2 со стоечными коммутаторами (ToR[1]), связанными через двухуровневое ядро. Предположим, что число хостов возросло и таблицы L2 переполняются. … P4 позволяет описать решение с минимальными изменениями архитектуры сети. … Маршруты через ядро представляются 32-битовыми метками, состоящими из 4 однобайтовых полей. Такая метка может указывать маршрут, заданный отправителем (source route) … Каждому коммутатору в ядре нужно проверить лишь один байт метки и выполнить коммутацию не его основе. [1]

В примере приведены две программы P4, одна из которых предназначена для краевых коммутаторов (ToR), другая - для агрегирующих (коммутаторы ядра). Эти программы используют общие определения заголовков пакетов, синтаксических анализаторов и действий.

1.3 Абстракции P4

P4 обеспечивает перечисленные ниже абстракции, экземпляры которых включаются в программы P4.

- Тип заголовка (Header type) - спецификация полей в заголовке.
- Экземпляр заголовка (Header instance) - конкретный экземпляр заголовка пакета или метаданных.
- Функция состояния анализатора (Parser state function) определяет заголовки, идентифицируемые в пакете.
- Функция действия (Action function) состоит из примитивов действий, применяемых совместно.
- Экземпляр таблицы (Table instance) задаёт поля для сопоставления и выполняемые действия.
- Функция управления потоком (Control flow function) - императивное описание порядка применения таблиц.
- Память состояний (Stateful memories) - счётчики, измерители и регистры, где учитываются пакеты.

В дополнение к этим абстракциям высокого уровня используются абстракции более низких уровней.

- Для экземпляра заголовка.
  - Метаданные хранят на уровне отдельного пакета состояние, которое не выводится из данных пакета. Иногда трактуются так же, как заголовки пакета.
  - Стек заголовков - непрерывный массив экземпляров заголовков.
  - Зависимые поля - поля, значения которых зависят от расчётов, применяемых к другим полям или константам.
- Для анализатора.
  - Набор значений (Value set) - обновляемые в процессе работы значения, определяющие смену состояний анализатора.
  - Расчёты контрольных сумм - возможность применить функцию к набору байтов из пакета и потом проверить соответствие поля повторному расчёту.

1.4 Структура языка P4

В этом параграфе (работа ещё не завершена) представлен краткий обзор структуры языка P4. Этот язык использует плоскую структуру типизации, выводящую типы для большинства параметров функций. В этом случае на каждом уровне P4 используется своё пространство имён, хотя возможны некоторые неоднозначности, упомянутые в спецификации.

Значения констант могут быть представлены в P4 двоичными (префикс 0b), десятичными и шестнадцатеричными (префикс 0x) значениями. Иногда требуется указывать число битов, которые следует использовать для представления значений. P4 позволяет сделать это путём добавления размера к базовой спецификации (1.5.1 Спецификации значений).

P4 поддерживает выражения с операторами, которые могут быть вычислены в момент компиляции.

1.5 Используемые соглашения

В этом документе грамматические конструкции P4 представлены в формате BNF с использованием приведённых ниже соглашений:

- для представления BNF используется зелёный цвет;
- терминальные узлы выделены *курсивом*;
- узлы, имена которых имеют суффикс _name, являются неявно строками, начинающимися с буквы (не цифры;
- узлы, за которыми следует символ +, указывают 1 или несколько экземпляров;
- узлы, за которыми следует символ *, указывают возможность наличия экземпляров возможно 0);
- символ | разделяет опции, из которых нужно выбрать одну;
- квадратные скобки [] служат для группировки узлов; группа является необязательной, если за ней не указан символ +; группа может указываться с символом * после неё, указывающим возможность наличия экземпляров;
- символы специального назначения (например, [ ] * + |) можно применять в качестве терминальных узлов путём размещения их внутри кавычек, например, "*";

[1]

- остальные символы являются литералами (например, фигурные скобки, точка с запятой, круглые скобки, точка);
- если правило не помещается в строку, новая строка с продолжением начинается сразу после символов ::=, а описание завершается пустой строкой;
- для представления примеров используется синий цвет;
- для выделения мест, где указан размер поля (этот размер может иметь значение), применяется узел field_value (он является синонимом для любого постоянного значения const_value).

Типы заголовков и определения таблиц задаются объявлениями, которые обычно состоят из набора пар атрибут-значение, разделённых двоеточием. Анализаторы, действия и поток управления задаются императивно с нетипизованными параметрами (если они имеются) и ограниченным набором операций.

1.5.1 Спецификации значений

Как отмечено в параграфе 1.4 Структура языка P4, язык P4 поддерживает базовые значения с возможностью указания их размера в битах. Формальное представление приведено ниже.

```
const_value ::= [ "+" | - ] [ width_spec ] unsigned_value
unsigned_value ::= binary_value | decimal_value | hexadecimal_value

binary_value ::= binary_base binary_digit+
decimal_value ::= decimal_digit+
hexadecimal_value ::= hexadecimal_base hexadecimal_digit+

binary_base ::= 0b | 0B
hexadecimal_base ::= 0x | 0X

binary_digit ::= _ | 0 | 1
decimal_digit ::= binary_digit | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
hexadecimal_digit ::=
       decimal_digit | a | A | b | B | c | C | d | D | e | E | f | F

width_spec ::= decimal_digit+ '
field_value ::= const_value
```

За спецификацией размера следует апостроф (не обратная галочка). Размер должен указываться десятичным числом.

Отметим, что константы всегда начинаются с цифры, в отличие от идентификаторов. Элемент const_value можно читать как значение константы (constant value). Элемент field_value используется в спецификации для того, чтобы указать возможность влияния размера представления. В остальных случаях он является синонимом const_value. Пробельные символы завершают спецификацию константы.

Символы подчёркивания допускаются в значениях для группировки цифр. Например, можно задать 78_256_803 или 0b1101_1110_0101 для удобства восприятия.

Битовый размер значения может быть указан с помощью bit_width. Нулевой размер в этом случае равносилен незаданному размеру. Если размер не указан перед значением, он выводится. Для положительных значений выведенный размер равен минимальному числа битов, требуемых для представления значения, для отрицательных - на 1 больше. Отрицательные числа представляются в форме дополнения до 2 (15.7 Преобразование значений полей[1]).

Ниже даны примеры представления некоторых значений.

Таблица 1. Примеры представления значений.

Обозначение	***Десятичное значение***	***Число битов***	***Примечания***
42	42	6	По умолчанию используется десятичная форма.
16’42	42	16	То же значение с явным указанием размера (16 битов).
0b101010	42	6	Двоичное представление того же числа без явного указания размера.
0’0x2a	42	6	Нулевой размер равносилен неуказанному размеру и реальный размер выводится из значения.
12’0x100	256	12	Пример указания числа битов и шестнадцатеричной формы.
7’0b1	1	7	Двоичное значение с явным указанием размера.
-0B101	-5	4	Знак не применяется до вычисления остальной части.

Выражения, включая выражения со скобками, поддерживаются для чисел и полей в предположении их вычисления компилятором. Ниже приведён список двоичных операторов для целочисленных значений, синтаксис, семантика, порядок и ассоциативность которых следуют соглашениям языка C.

```
+ - * / % « » | & ^ ~
```

2 Заголовки и поля

2.1 Объявления типов заголовков

Типы заголовков описывают схему полей и задают имена для ссылок на поля данных. Типы заголовков служат для обновления экземпляров заголовков и метаданных, как описано в следующем параграфе.

Типы заголовков задаются, как показано ниже (в формате BNF).

```
header_type_declaration ::=
       header_type header_type_name { header_dec_body }
header_dec_body ::=
```

[1] *В оригинале ошибочно указано отсутствующее Приложение 17.5. Прим. перев.*

```
        fields { field_dec + }
        [ length : length_exp ; ]
        [ max_length : const_value ; ]
field_dec ::= field_name : bit_width [ ( field_mod ) ];
field_mod ::= signed | saturating | field_mod , field_mod
length_bin_op ::= "+" | - | "*" | "<<" | ">>"
length_exp ::=
        const_value |
        field_name |
        length_exp length_bin_op length_exp |
        ( length_exp )
bit_width ::= const_value | "*"
```

При определении заголовков используются перечисленные ниже соглашения.

- Типы должны иметь атрибут fields.
  - Список отдельных полей упорядочивается.
  - Поля по умолчанию не имеют знака и не насыщаются (т. е. операции сложения и вычитания учитывают переход между максимум и 0).
  - Смещения битов поля от начала заголовка определяются суммой размеров предшествующих полей списка.
  - Байты упорядочиваются в соответствии с заголовком пакета.
  - Биты упорядочиваются от старшего к младшему. Таким образом, если первое поле в списке заголовка имеет размер 1, это будет старший бит первого байта в заголовке.
  - Все биты заголовка должны помещаться в то или иное поле.
  - Не более одного поля в типе заголовка может быть указано а форме *, обозначающей переменный размер.
- Если все поля имеют фиксированный размер (нет полей *), заголовок также будет иметь фиксированный размер. В противном случае размер заголовка будет переменным
- Атрибут length задаёт выражение, значением которого определяет размер заголовка в байтах (для заголовков с переменным размером).
  - Этот атрибут является обязательным для полей переменного размера (имеется поле *).
  - При наличии этого атрибута у заголовка фиксированного размера компилятор должен выдавать предупреждение.
  - Поля, указанные в атрибуте length, должны размещаться до поля переменного размера.
- Атрибут max_length указывает максимально разрешенный размер заголовка (переменного размера) в байтах.
  - Если в процессе работы рассчитанный размер превысит заданное значение, это считается особым случаем синтаксического анализа (см. параграф 4.6 Исключительные случаи при анализе).
  - Атрибут max_length применяется лишь для заголовков переменного размера.
  - При наличии этого атрибута у заголовка фиксированного размера компилятор должен выдавать предупреждение.
- Порядок применения и ассоциативность операторов определяются соглашениями языка C.

P4 поддерживает заголовки переменного размера для пакетов за счёт использования полей со специальным битом размера *. Размер таких полей выводится из общего размера заголовка (в байтах), указанного атрибутом length. Размер поля в битах составляет ((8 * length) - сумма размеров фиксированных полей). В заголовке разрешается наличие не более одного поля с размером *.

Ниже приведён пример объявления заголовка VLAN (802.1Q).

```
header_type vlan_t {
        fields {
                pcp             : 3;
                cfi             : 1;
                vid             : 12;
                ethertype       : 16;
        }
}
```

Типы метаданных заголовка объявляются с применением такого же синтаксиса.

```
header_type local_metadata_t {
        fields {
                cpu_code        : 16;   // Код для пакетов, отправляемых в CPU
                port_type       : 4;    // Тип порта: up, down, local...
                ingress_error   : 1;    // Ошибка при проверке входного порта
                was_mtagged     : 1;    // Отслеживание помеченного (mtag) на входе пакета
                copy_to_cpu     : 1;    // Специальный код, задающий копирование в CPU
                bad_packet      : 1;    // Прочие ошибки
        }
}
```

2.2 Экземпляры заголовков и метаданных

Хотя объявление типа header задаёт тип заголовка, пакет может включать множество экземпляров данного типа. P4 требует явно объявлять каждый экземпляр до ссылок на него. Имеется два типа экземпляров заголовков - пакеты и метаданные. Обычно заголовки пакетов идентифицируются по прибытии пакетов на вход устройства, а метаданные хранят информацию о пакете, которая обычно отсутствует в данных пакета (например, входной порт или временная метка). Большая часть метаданных представляет просто состояние на уровне пакета, используемое в процессе обработки подобно регистрам. Однако некоторые метаданные могут иметь особое значение для работы коммутатора. Например, система очередей может использовать значение определённого поля метаданных при выборе очереди для пакета. P4 признает такую зависимую от платформы семантику, но не пытается её представлять.

Заголовки пакетов (ключевое слово header) и метаданные (ключевое слово metadata) отличаются лишь их пригодностью (validity). В заголовках пакетов поддерживается отдельный индикатор пригодности, который проверяется явно, а метаданные считаются действительными всегда. Это различие дополнительно рассматривается в параграфе 2.2.1 Проверка пригодности экземпляров заголовков и метаданных. По умолчанию экземпляры метаданных инициализируются со значением 0. Форма BNF для заголовков и метаданных представлена ниже.

```
instance_declaration ::= header_instance | metadata_instance
header_instance ::= scalar_instance | array_instance
scalar_instance ::= header header_type_name instance_name ;
array_instance ::=
        header header_type_name
                instance_name "[" const_value "]" ;

metadata_instance ::=
        metadata header_type_name
                instance_name [ metadata_initializer ] | ;

metadata_initializer ::= { [ field_name : field_value ; ] + }
```

Ниже приведены некоторые замечания.

- Только заголовки (не экземпляры метаданных) могут образовывать массивы (стеки заголовков).
- В header_type_name должно быть имя объявленного типа заголовка.
- Экземпляры метаданных недопустимо создавать с типом заголовка переменного размера.
- Поля, названные в инициализаторе, должны присутствовать в списке полей типа заголовка.
- При наличии инициализации названные поля инициализируются указанными значениями, неназванные - 0.
- Для экземпляров заголовков компилятор должен выдавать ошибку, если общий размер всех полей в типе заголовка не является целым числом байтов. Компилятор может дополнять заголовок до целого числа байтов.

Например,

```
header vlan_t inner_vlan_tag;
```

показывает, что в Parsed Representation для пакета должно быть выделено пространство для заголовка vlan_t. Этот заголовок можно указать при синтаксическом разборе и в таблице СД по имени inner_vlan_tag.

Примером метаданных может служить

```
metadata local_metadata_t local_metadata;
```

Это указывает, что следует выделить объект типа local_metadata_t с именем local_metadata для применения в СД.

2.2.1 Проверка пригодности экземпляров заголовков и метаданных

Заголовки пакетов и их поля могут проверятся на предмет их действительности (пригодности) по соответствующему биту. Проверка пригодности и сборка (5 Сборка пакета) являются единственными точками, где заголовки пакетов и метаданные могут различаться.

Экземпляр заголовка, объявленный с ключевым словом header, является действительным, если он извлечён в процессе синтаксического анализа (4 Спецификация анализатора[1]) или действие делает его действительным. Поле внутри экземпляра заголовка является действительным, если действителен включающий поле экземпляр заголовка. Для экземпляра метаданных все поля являются действительными. Проверка действительности метаданных должна вызывать предупреждение компилятора и в любом случае должна возвращать значение true.

> *Исключение*. Причину исключения проще всего понять, рассматривая случай флагов. Предположим, например, что в метаданных используется однобитовый флаг, указывающий наличие у пакета некого атрибута (например, v4 или v6 для пакета IP). Здесь нет практической разницы между флагом со значением 0 и недействительным флагом. Точно также многим «индексным» полям метаданных может присваиваться зарезервированное значение для указания их непригодности. Однако в некоторых случаях было бы полезно иметь независимый бит действительности поля метаданных и над этим следует поработать.

При использовании недействительного пакета как части ключа поиска, значение заголовка будет неопределённым. Операция сопоставления может явно проверять пригодность экземпляра заголовка (или поля). При сборке учитываются лишь действительные заголовки пакетов (5 Сборка пакета).

2.2.2 Стек заголовков

P4 поддерживает понятие «стека заголовков», который представляет собой последовательность смежных экземпляров однотипных заголовков. Примерами могут служить стеки меток MPLS и тегов VLAN. Стеки заголовков объявляются как массивы (см. array_instance в параграфе 2.2 Экземпляры заголовков и метаданных).

[1] *В оригинале ошибочно указан раздел 5. Прим. перев.*

Экземпляры заголовков из стека указываются как элементы массива (в квадратных скобках) и эквивалентны не входящим в стек экземплярам. Анализатор поддерживает информацию для управления стеком заголовков.

2.3 Ссылки на заголовки и поля

Для спецификаций сопоставлений, действий и потоков управления нужны ссылки на экземпляры заголовков и их поля. Заголовки указываются по именам экземпляров, а для стеков дополнительно указывается индекс в квадратных скобках.

```
header_ref ::= instance_name | instance_name "[" index "]"
        index ::= const_value | last
```

Для указания конкретного поля заголовка служат имя экземпляра и поля, разделённые точкой.

```
field_ref ::= header_ref . field_name
```

Примером может служить inner_vlan_tag.vid, где inner_vlan_tag объявлен как экземпляр заголовка типа vlan_tag.

- Поля должны указываться в атрибуте fields при объявлении заголовка.
- Ссылки на поля должны всегда указываться относительно родительского заголовка. Это позволяет использовать одно имя поля в разных типах заголовков без возникновения путаницы.
- Каждый экземпляр заголовка в данный момент может быть действительным или недействительным. Это состояние может быть проверено в процессе обработки СД.
- Ссылки на недействительный заголовок (или какое-то из его полей) в процессе работы приводят к особому неопределённому (undefined) значению, влияние которого зависит от контекста.

Чтобы помочь программистам в создании синтаксических анализаторов для стеков заголовков, P4 поддерживает два ключевых слова next и last, которые могут применяться для ссылок на экземпляры заголовков в стеке. Такие ссылки преобразуются автоматически, когда анализатор вызывает функцию extract для заголовка. Если stk является стеком заголовков, то stk[next] изначально указывает в stk заголовок с индексом 0 и автоматически перемещается при каждом вызове extract(stk[next]). Если вызов extract(stk[next]) выполняется более N раз, где N - размер стека, вычисление stk[next] ведёт к исключению p4_pe_index_out_of_bounds. Ключевое слово last указывает экземпляр, на который будет указывать next перед последним возможным вызовом extract. Если такого экземпляра нет, возвращается p4_pe_index_out_of_bounds. Не допускается использование next и last за пределами анализатора.

Хотя допускается использование extract с постоянным индексом, попадающим в стек, вместе с применением next, смешивать эти режимы не рекомендуется во избежание путаницы для программистов.

2.4 Списки полей

Во многих случаях удобно указывать последовательность полей. Например, функция хэширования может принимать такую последовательность на входе или подсчёт контрольной суммы может выполняться для последовательности полей. P4 поддерживает такие объявления и каждая запись в списке может указывать конкретное поле, экземпляр заголовка (это эквивалентно указанию всех полей данного заголовка) или содержать фиксированное значение. В списках могут указываться заголовки пакетов и метаданные.

```
field_list_declaration ::=
        field_list field_list_name {
                [ field_list_entry ; ] +
        }
field_list_entry ::=
        field_ref | header_ref | field_value | field_list_name | payload
```

В списке полей можно указать другой список полей. В результате имена списков полей и имена экземпляров заголовков являются частью одного пространства имён. Рекурсивные ссылки в списках не поддерживаются.

Идентификатор payload показывает включение в список полей содержимого пакета, следующего после заголовка. Это сделано для поддержки особых случаев, таких как расчёт Ethernet CRC для всего пакета или контрольной суммы TCP.

3 Расчёт контрольных сумм и хэш-значений

Генераторы контрольных сумм и хэш-значений являются примерами функций, работающих с потоком байтов из пакета и дающих на выходе целое число. Такие функции часто применяются в сетях. Целочисленный результат может применяться, например, для контроля целостности пакета или в качестве псевдослучайного значения из определённого диапазона, полученного для пакетов или потоков.

P4 позволяет связать функцию с набором полей и ссылаться на результат операции (отображение пакетов на целые числа) в программах P4. Эти функции называют расчётом для списка полей (field list calculations) или объектом расчёта (calculation object). P4 не поддерживает задание выражений для алгоритма базовой функции, считая их примитивами действий. Для удобства задан набор известных алгоритмов.

Функция расчёта для списка полей сопоставляет с пакетом целое число и может настраиваться в процессе работы через интерфейс API. Целевые платформы могут различаться в части поддержки этих интерфейсов, но обычно поддерживается задание значения «затравки» (seed), а также могут настраиваться некоторые параметры алгоритма (например, коэффициенты используемого при расчёте полинома) и даже набор используемых полей.

Расчёт для списка полей может быть указан как свойство расчёта контрольных сумм (параграф 3.1 Контрольные суммы) или примитив действия.

```
field_list_calculation_declaration ::=
        field_list_calculation field_list_calculation_name {
                input {
                        [ field_list_name ; ] +
                }
                algorithm : stream_function_algorithm_name ;
```

```
            output_width : const_value ;
        }
```

Интерфейс API в процессе работы позволяет выбрать один из списков входных полей для использования. По умолчанию применяется указанное первым имя.

Значение output_width указывает выходной размер в битах.

Экземпляр поля исключается из расчёта (как будто его нет в списке), если заголовок поля недействителен.

Алгоритм указывается строковым значением (string). Ниже приведён список определённых алгоритмов, но платформы могут поддерживать другие алгоритмы.

- xor16 - операция XOR для взятых попарно байтов;
- csum16 - контрольная сумма заголовка IPv4 в соответствии с RFC 791 (https://tools.ietf.org/html/rfc791#page-14);
- crc16 - см. http://en.wikipedia.org/wiki/Crc16;
- crc32 - см. http://en.wikipedia.org/wiki/Crc32;
- programmable_crc - этот алгоритм позволяет задать произвольный полином CRC (см. http://en.wikipedia.org/wiki/Cyclic_redundancy_check).

3.1 Контрольные суммы

В некоторых полях (например, контрольная сумма IP) содержится результат потокового расчёта. P4 позволяет представлять такие зависимости с объявлением расчётных полей. Вычисляемые поля важны в том смысле, что они проверяются на входе и обновляются на выходе.

Синтаксис связывает последовательность директив проверки или обновления экземпляра конкретного поля и могут включать условия их применения. Первая запись, для которой условие выполняется пакетом (условие может отсутствовать) определяет привязку. Это позволяет менять расчёты на основе формата пакетов. Например, расчёт контрольной суммы TCP может несколько различаться для пакетов с заголовками IPv4 и IPv6. Отметим, что условия проверяются перед выполнением операции обновления или проверки. В настоящее время поддерживается ограниченный набор условий.

```
calculated_field_declaration ::=
        calculated_field field_ref { update_verify_spec + }
update_verify_spec ::=
        update_or_verify field_list_calculation_name [ if_cond ] ;
update_or_verify ::= update | verify
if_cond ::= if ( calc_bool_cond )
calc_bool_cond ::=
        valid ( header_ref | field_ref ) |
        field_ref == field_value
```

Ниже приведён пример объявления. Предполагается, что объявления field_list_calculation для tcpv4_calc и tcpv6_calc уже заданы, а ipv4 и ipv6 являются экземплярами заголовков.

```
calculated_field tcp.chksum {
        update tcpv4_calc if (valid(ipv4));
        update tcpv6_calc if (valid(ipv6));
        verify tcpv4_calc if (valid(ipv4));
        verify tcpv6_calc if (valid(ipv6));
}
```

Для контрольный сумм вычисление по списку полей предназначено для привязки списка полей и алгоритма к конкретным экземплярам полей. Это объявление показывает, что значение, сохранённое в field_ref, предполагается значением, рассчитанным для указанного набора полей в пакете. Хотя P4 поддерживает такие объявления в любом месте программы, размещать их следует сразу после объявлений соответствующего экземпляра заголовка. Поля переменного размера (*) не разрешается объявлять в списках.

Опция verify указывает, что анализатору следует рассчитать значение для списка полей и сравнить его со значением указанного поля. При расхождении возникает исключение p4_pe_checksum (см. параграф 4.6.1 Стандартные исключения анализатора[1]). Эта проверка выполняется в конце синтаксического анализа и лишь при действительном field_ref.

Опция update указывает системе на необходимость обновить значение поля, если были внесены изменения в какое-либо из полей, от которых данное поле зависит. Операция обновления выполняется при выходной сборке. Если ни одно поле не изменялось, сохраняется значение из конвейера СД.

4 Спецификация анализатора

Синтаксический анализатор P4 представляется как машина состояний (конечный автомат). Он может быть представлен графом анализа, с узлами в качестве состояний и рёбрами, задающими смену состояния. На рисунке 2 приведён очень простой пример. Отметим, что на этом рисунке указаны заголовки для каждого состояния. Хотя такой подход поддерживается в P4, он не является обязательным. Узел в графе анализа может быть «чистым узлом принятия решений», не связанным с конкретным экземпляром заголовка, а также возможна обработка одним узлом нескольких заголовков.

Ниже показаны несколько функций анализатора P4 для примера mTag. Функция start вызывает анализатор ethernet напрямую.

[1] *В оригинале ошибочно указан параграф 6.6.1. Прим. перев.*

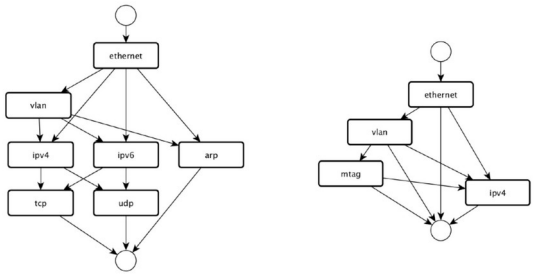

Рисунок 2. Простой граф анализа и граф mTag.

```
parser ethernet {
        extract(ethernet);     // Начало анализа с заголовка ethernet.
        return select(latest.ethertype) {
                0x8100:         vlan;
                0x800:          ipv4;
                default:        ingress;
        }
}
parser vlan {
        extract(vlan);
        return select(latest.ethertype) {
                0xaaaa:         mtag;
                0x800:          ipv4;
                default:        ingress;
        }
}
parser mtag {
        extract(mtag);
        return select(latest.ethertype) {
                0x800:          ipv4;
                default:        ingress;
        }
}
```

Ссылка на ingress прерывает синтаксический анализ и вызывает входную функцию потока управления.

4.1 Разобранное представление

Анализатор создаёт представление пакета, с которым работает этап СД. Оно называется разобранным представлением (Parsed Representation) пакета. Это набор экземпляров заголовков, которые действительны для пакета. Анализатор создаёт исходное разобранное представление как описано ниже. Операции СД могут менять это представление путём изменения значений полей и смены состояний пригодности для экземпляров заголовков. Последнее ведёт к исключению или добавлению соответствующего заголовка.

Разобранное представление содержит заголовки пакета с учётом их обновления операциями СД. Данные исходного пакета могут поддерживаться с помощью специальных операций (например, клонирование), описанных в разделе 14 Рециркуляция и клонирование.

Метаданные считаются частью Parsed Representation для пакета, поскольку они обычно трактуются подобно заголовкам.

4.2 Операция синтаксического анализа

Пакет передаётся анализатору, начиная с первого байта. Анализатор поддерживает значение текущего смещения в пакете, которое служит указателем на конкретный байт заголовка. Анализатор извлекает заголовки из пакета по текущему смещению в связанный с пакетом экземпляр заголовка (метаданные) и помечает этот экземпляр как действительный, обновляя Parsed Representation для пакета. Затем анализатор увеличивает смещение (для перехода к следующему действительному байту) и меняет своё состояние.

Программа P4 может проверять метаданные для принятия решений о смене состояния, хотя целевые платформы могут вносить ограничения для таких проверок. Например, может использоваться номер входного порта для определения начального состояния анализатора, допускающего разные форматы пакетов. Метаданные, предоставляемые клонированием или рециркуляцией точно так же можно использовать для управления поведением анализатора (см. раздел 14 Рециркуляция и клонирование).

В P4 каждое состояние представляется как функция синтаксического анализатора. Эти функции могут завершаться одним из четырёх способов.

- Оператор return, задающий имя выполняемой функции анализатора, которая служит следующим этапом обработки пакета.
- Оператор return, задающий имя функции управления (12 Обработка пакетов и поток управления). Это прерывает процесс анализа с переходом с операции СД путём вызова указанной функции управления.
- Явное выполнение оператора parse_error (см. параграф 4.6 Исключительные случаи при анализе).
- Неявная ошибка анализатора (4.6.1 Стандартные исключения анализатора).

Отметим, что два первых случая требуют нахождения имён функций анализатора и функций управления в одном пространстве имён. Компилятор должен выдавать ошибку при совпадении имён таких функций.

Определена операция select для ветвления на разных этапах обработки в зависимости от значений выражений, включающего поля или данные пакета.

Если заголовки нужно извлечь при входе в то или иное состояние, это указывается явно путём вызова функции extract (4.5 Функция extract[1]) в начале определения функции анализатора (4 Спецификация анализатора[2]).

4.3 Наборы значений

В некоторых случаях значение, определяющее переход анализатора из одного состояния в другое, нужно определять в процессе работы. Примером может служить MPLS, где значение метки определяет, какой заголовок следует после поля метки MPLS и это может динамически меняться в процессе работы. Для поддержки нужной функциональности в P4 служат наборы значений анализатора (Parser Value Set). Это именованные наборы значений с run-time API для добавления и удаления значений из набора. Имя набора можно указывать в условиях смены состояний анализатора.

Value Set содержат лишь значения, но не типы заголовков или информацию о смене состояния. Все значения в наборе должны соответствовать одному переходу состояния. Например, все метки MPLS, соответствующие переходу к IPv4, должны присутствовать в одном наборе, а метки для перехода к IPv6 - в другом. Наборы меток объявляются на верхнем уровне программы P4 за пределами функций анализатора и используют одно глобальное пространство имён. Наборы значений следует определять до их указания в функциях анализатора.

```
value_set_declaration ::= parser_value_set value_set_name;
```

Размер набора выводится из того места, где этот набор упомянут. Если набор значений применяется неоднократно и для него будет выводиться разный размер, компилятор должен сообщать об ошибке.

Интерфейс run-time API для обновления наборов значений анализатора должен разрешать пары «значение-маска», задаваемые совместно.

4.4 BNF для функций анализатора

Ниже представлена форма BNF для объявления функций анализатора.

```
parser_function_declaration ::=
       parser parser_state_name { parser_function_body }

parser_function_body ::=
       extract_or_set_statement*
       return_statement

extract_or_set_statement ::= extract_statement | set_statement
extract_statement ::= extract ( header_extract_ref );
header_extract_ref ::=
       instance_name |
       instance_name "[" header_extract_index "]"

header_extract_index ::= const_value | next

set_statement ::= set_metadata ( field_ref, metadata_expr ) ;
metadata_expr ::= field_value | field_or_data_ref

return_statement ::=
       return_value_type |
       return select ( select_exp ) { case_entry + }

return_value_type ::=
       return parser_state_name ; |
       return control_function_name ; |
       parse_error parser_exception_name ;

case_entry ::= value_list : case_return_value_type ;
value_list ::= value_or_masked [ , value_or_masked ]* | default

case_return_value_type ::=
       parser_state_name |
       control_function_name |
       parse_error parser_exception_name
value_or_masked ::=
       field_value | field_value mask field_value | value_set_name

select_exp ::= field_or_data_ref [, field_or_data_ref] *
field_or_data_ref ::=
       field_ref |
       latest.field_name |
       current( const_value , const_value )
```

Функция extract может извлекать лишь заголовки пакетов, но не метаданные. Функция select принимает разделённый запятыми список полей и выполняет для них конкатенацию, помещая левое значение в старшие биты объединённого значения. Операция select затем сравнивает значения в порядке из указания в программе для обнаружения совпадений. Оператор mask служит для указания троичного сопоставления (ternary) с использованием заданной маски. Сравнение выражения select со значением case используется лишь для указанных маской битов, т. е. для выражения select и значения поочерёдно выполняется операция AND с mask, а затем происходит сравнение.

Поддержка сравнений с маской и наборов значений может давать более одного совпадения. Предпочтение определяется порядком case (предпочтительным будет первое совпадение.

[1] *В оригинале ошибочно указан параграф 7.5. Прим. перев.*
[2] *В оригинале ошибочно указан раздел 7. Прим. перев.*

Ссылка на заголовок latest относится к извлечённому последним экземпляру заголовка в функции анализа. Использование такой формы без предшествующей операции extract в той же функции является ошибкой.

Ссылка на поле current(...) позволяет анализатору указать биты, которые ещё не разобраны по полям. Первый аргумент задаёт смещение в битах от текущей позиции, а второй аргумент - число битов. Результат трактуется как целое число без знака с указанным размером. Он преобразуется в поле метаданных в соответствии с правилами, описанными в приложении 15.7 Преобразование значений полей[1].

Указанное в операторе set_metadata поле должно относиться к экземпляру метаданных. Если размер значения отличается от размера целевого поля метаданных, выполняется преобразование в соответствии с приложением 15.7 Преобразование значений полей.

4.5 Функция extract

Функция extract принимает в качестве параметра экземпляр заголовка (это не могут быть метаданные). Извлекается копия данных из пакета по текущему смещению с переносом указателя в конец данного заголовка. Отметим использование специального идентификатора next (а не last) для стека заголовков, поскольку извлечение выполняется в следующее доступное свободное место.

4.6 Исключительные случаи при анализе

Существует два варианта трактовки пакетов при возникновении ошибки в процессе синтаксического анализа - отбрасывание и продолжение обработки пакета. В первом случае пакет может быть отброшен анализатором незамедлительно. Реализациям следует поддерживать один или несколько счётчиков для таких ошибок.

В другом случае процесс анализа останавливается с установкой специальных метаданных, указывающих ошибку при анализе, и пакет передаётся функции управления для обработки СД. Обработка выполняется в соответствии с заданными правилами СД, как для прочих пакетов, но набор правил может проверять метаданные об ошибке и применять те или иные действия, например, передавать пакет плоскости управления.

Существует множество условий, при которых распознаваемые P4 ошибки могут возникать неявно. Кроме того, программист может указывать такие ошибки, как исключительная ситуация parse_error, в функции синтаксического анализатора. Оба эти варианта обрабатываются однотипно. Обработчик исключительных ситуаций в синтаксическом анализаторе можно объявить в программе, как показано ниже. Может использоваться набор метаданных, за которым следует возврат в функцию управления или отбрасывание пакета. Отметим, что установка метаданных будет давать эффект лишь при выполнении оператора return.

```
parser_exception_declaration ::=
       parser_exception parser_exception_name {
              set_statement *
              return_or_drop ;
       }
return_or_drop ::= return_to_control | parser_drop
return_to_control ::= return control_function_name
```

4.6.1 Стандартные исключения анализатора

В таблице 2 приведён список имён исключительных (особых) ситуаций при синтаксическом анализе. Идентификаторы событий имеют префикс pe (parser exception - особый случай при анализе).

Таблица 2. Стандартные исключения синтаксического анализатора.

Идентификатор	***Событие***
p4_pe_index_out_of_bounds	Индекс стека заголовков выходит за объявленные границы.
p4_pe_out_of_packet	В пакете недостаточно байтов для выполнения операции извлечения (extract).
p4_pe_header_too_long	Рассчитанный размер заголовка превышает заявленный максимум.
p4_pe_header_too_short	Рассчитанный размер заголовка меньше минимального размера фиксированной части заголовка.
p4_pe_unhandled_select	Для оператора select не задан принятый по умолчанию вариант, а значение выражения отсутствует в списке вариантов.
p4_pe_checksum	Обнаружена ошибка в контрольной сумме.
p4_pe_default	Это не является особым случаем, но позволяет программисту задать используемый по умолчанию обработчик на случай отсутствия подходящего обработчика.

При передаче исключения конвейеру СД тип исключения указывается в метаданных (6 Стандартные внутренние метаданные).

4.6.2 Принятая по умолчанию обработка исключительных случаев

Если обработчик p4_pe_default задан и происходит исключительный случай, для которого в программе не указан обработчик parser_exception, вызывается p4_pe_default. При возникновении исключительного случая, для которого нет обработчика parser_exception в отсутствие обработчика p4_pe_default анализатор просто отбрасывает пакет.

5 Сборка пакета

На выходе элемент пересылки преобразует разобранное представление пакета (обновлённое в СД) в последовательный поток байтов для передачи. Этот процесс называется сборкой (deparsing), поскольку он в некотором смысле является обращением синтаксического анализа. В P4 применяется подход, в соответствии с которым, любой формат, который следует генерировать на выходе, должен быть представлен анализатором, используемым на входе. Таким образом, граф анализа, представленный в программе P4, служит для определения алгоритма, используемого при создании пакета из Parsed Representation. Отметим несколько важных моментов:

- при сборке используются лишь действительные (valid) заголовки;

[1] *В оригинале ошибочно указано приложение 17.5. Прим. перев.*

- если граф анализа является ациклическим, может быть реализовано топологическое упорядочение (т. е. линейный порядок в соответствии с порядком графа анализа), которое может служить для определения порядка заголовков при сборке;
- в общем случае циклы возникают в графе анализа при разборе стеков заголовков или набора необязательных заголовков, которые можно считать одним узлом графа анализа и обрабатывать как группу;
- поля метаданных не включаются в сборку напрямую (поскольку они не анализируются), но могут копироваться в поля заголовков пакета при обработке СД, что позволяет учесть их в выходном пакете.

6 Стандартные внутренние метаданные

Метаданные указывают состояния, связанные с каждым пакетом. Их можно считать набором переменных, связанным с пакетом, которые могут считываться и обновляться действиями таблиц. Однако некоторые метаданные имеют особое значение для работы системы. Примерами таких метаданных может служить номер входного или выходного порта. Первый является примером данных, доступных лишь для чтения и устанавливаемых при поступлении пакета в коммутатор. Значение второго указывается действием таблицы, а затем обрабатывается системой буферизации, в результате чего пакет передаётся в определённый выходной порт или порты.

Эта спецификация задаёт стандартные внутренние поля метаданных, поддержка которых обязательна для совместимых с P4 платформ. Хотя эти поля обязательны, формат их может зависеть от платформы и должен предоставляться платформой, автоматически включаться компилятором как заголовки или включаться в реализацию компилятора.

Стандартные внутренние метаданные (Standard Intrinisic Metadata) названы так потому, что они устанавливаются автоматически (например, ingress_port) или требуются для описания работы абстрактной машины (например, egress_port). В таблице 3 приведены поря, определённые для экземпляров standard_metadata.

Таблица 3. Поля Standard Intrinisic Metadata.

Поле	***Описание***
ingress_port	Порт, через который был принят пакет. Определён всегда и устанавливается до синтаксического анализа. Доступно лишь для чтения.
packet_length	Число байтов в пакете. Для кадров Ethernet поле CRC не учитывается. Устанавливается до анализа. Не может применяться в сопоставлении или указываться в действиях, если коммутатор работает в режиме cut-through. Доступно лишь для чтения.
egress_spec	Задаёт выходной порт. Не определено до установки операцией СД в процессе входной обработки. Может указывать физический порт, логический интерфейс (туннель, LAG, маршрут, VLAN) или multicast-группу.
egress_port	Физический порт для передачи экземпляра пакета. Значение определяется блоком буферизации и поэтому действительно лишь в выходных СД (см. раздел 13 Выбор выходного порта, репликация, очередь). Доступно лишь для чтения.
egress_instance	Неанализируемый идентификатор для обозначения экземпляров пакета. Как и для egress_port, значение определяется блоком буферизации и поэтому действительно лишь в выходных СД (см. раздел 13 Выбор выходного порта, репликация, очередь). Доступно лишь для чтения.
instance_type	Указывает тип экземпляра пакета (normal, ingress clone, egress clone, recirculated, resubmitted). Представление данных зависит от платформы.
parser_status	Результат синтаксического анализа пакета. Значение 0 говорит об отсутствии ошибок, иные значения указывают код ошибки, зависящий от платформы.
parser_error_location	При наличии ошибок в процессе анализа это поле указывает место в программе анализатора, где произошла ошибка. Представление данных зависит от платформы.

Платформы могут предоставлять свои определения внутренний метаданных, но зависящие от таких определений программы могут терять переносимость. В документе [3] указаны поля внутренний метаданных, поддерживаемые прототипом платформы BMv2 simple_switch.

7 Счётчики, измерители и регистры

Счётчики, измерители и регистры поддерживают состояния, связанные более чем с одним пакетом. Совместно эти элементы называют памятью состояний (stateful memories). Для таких элементов требуются ресурсы целевой платформы, управление которыми осуществляет компилятор.

В этом разделе отдельный счётчик, измеритель или регистр рассматриваются как «ячейка» памяти. В P4 память состояний реализована в форме именованного массива таких ячеек, имеющих один тип. Ячейки указываются именем и индексом. Чтение и обновление содержимого ячеек могут выполнять действия, применяемые таблицей. Целевые платформы могут вносить ограничения на объем вычислений при определении индекса для доступа к ячейке, а также ограничивать операции обновления содержимого ячеек. Например,

```
counter ip_pkts_by_dest {
        type : packets;
        direct : ip_host_table;
}
```

объявляет набор счётчиков, связанных с таблицей ip_host_table. Выделяется одна ячейка счётчика для каждой записи таблицы. В другом примере

```
meter customer_meters {
        type : bytes;
        instance_count : 1000;
}
```

объявляется массив из 1000 измерителей, названный customer_meters. Эти измерители можно указать из любого действия любой таблицы (хотя обычно это будут делать 1 - 2 таблицы).

P4 позволяет глобальный доступ к памяти состояний (из любой таблицы) или статические обращения с привязкой к одному экземпляру таблицы. Обычно записи одной или разных таблиц могут обращаться к одной ячейке. Это называется непрямым доступом (indirect access). P4 поддерживает также прямой доступ (direct access) к ресурсам памяти состояний, привязанным к одной таблице с выделением ячейки для каждой записи таблицы. Примером этого могут служить счётчики, связанные с отдельной записью таблицы.

Компилятор пытается выделить требуемые для программы ресурсы с учётом возможностей целевой платформы. Однако ограничения платформы могут воспрепятствовать этому, например, платформа может не поддерживать обращения к одному глобальному ресурсу из входного и выходного конвейера.

Счётчики, измерители и регистры указываются в специальных примитивах действий, как описано в параграфе 9.1 Примитивы действий.

7.1 Счётчики

Формат объявления счётчиков показан ниже.

```
counter_declaration ::=
        counter counter_name {
                type : counter_type ;
                [ direct_or_static ; ]
                [ instance_count : const_expr ; ]
                [ min_width : const_expr ; ]
                [ saturating ; ]
        }
counter_type ::= bytes | packets | packets_and_bytes
direct_or_static ::= direct_attribute | static_attribute
direct_attribute ::= direct : table_name
static_attribute ::= static : table_name
```

Атрибут min_width указывает минимальное число битов, требуемых для каждой ячейки. Компилятор или целевая платформа при этом могут создавать ячейки большего размера.

Атрибут saturating показывает, что счётчик будет прекращать подсчёт при достижении максимального значения (в соответствии с реальным размером ячейки). Если этот атрибут не установлен, счётчик будет работать в кольцевом режиме (wrap).

Если счётчик объявлен с атрибутом direct, с каждой записью указанной таблицы связывается ячейка счётчика. В этом случае не требуется операции по учёту для действия таблицы и счётчик обновляется автоматически при каждом применении записи. В результате имена счётчиков с атрибутом direct становятся недоступными для примитива count и компилятор должен выдавать ошибку при ссылках на такие счётчики.

В API, используемых в процессе работы, следует указывать действительный размер счётчика. Это требуется для определения максимального значения счётчика, необходимого для контроля насыщения счётчика или перехода от максимума в 0.

Если счётчик объявлен без атрибута direct, действия должны указывать его по имени и индексу.

Если счётчик объявлен с атрибутом static, ресурс счётчика выделяется для указанной таблицы. Компилятор должен возвращать ошибку при обнаружении ссылки на такой счётчик из действий другой таблицы.

Атрибут instance_count указывает число создаваемых экземпляров (ячеек) счётчика. Этот атрибут требуется для счётчиков без атрибута direct. Компилятор должен возвращать ошибку при объявлении счётчика с обоими атрибутами instance_count и direct или при отсутствии обоих атрибутов.

Счётчик типа bytes инкрементируется на размер пакета в байтах неявно (direct) или явно (static). Счётчик типа packets инкрементируется на 1 при каждом действии, выполняемом для этого счётчика. Счётчик типа packets_and_bytes фактически состоит из двух счётчиков, которые инкрементируются на размер пакета или 1.

7.2 Измерители

Объявление измерителя имеет показанную ниже форму, похожую на объявление счётчика.

```
meter_declaration ::=
        meter meter_name {
                type : meter_type ;
                [ result : field_ref ; ]
                [ direct_or_static ; ]
                [ instance_count : const_expr ; ]
        }
meter_type ::= bytes | packets
```

Измерители являются объектами с состоянием (stateful), измеряющими скорость данных числом пакетов или байтов в секунду и выводящими результат в форме одного из трёх «цветов» (красный, жёлтый, зелёный), представляемых 2-битовым полем. Представление «цвета» зависит от платформы, однако предполагается, что каждая платформа определяет константы METER_COLOR_RED, METER_COLOR_YELLOW и METER_COLOR_GREEN, которые понятны компилятору и могут применяться в переносимых программах P4.

Спецификация P4 не задаёт конкретных алгоритмов измерения для реализации измерителей, поэтому описание семантики цветов выходит за рамки P4. Хотя алгоритмы трёхцветной маркировки, заданные в RFC 2697 и RFC 2698, являются хорошими примерами, возможны и другие варианты. Конфигурация измерителей также зависит от платформы и не определяется в P4[1].

Если измеритель объявлен с атрибутом direct, ячейка измерителя связывается с каждой записью указанной таблицы. В таких случаях не требуется операции meter для действия таблицы и измеритель будет обновляться автоматически при

[1] *В общем случае все аспекты рабочей (run-time) конфигурации относятся к плоскости управления и выходят за рамки данной спецификации P4. В будущем сообщество P4 может задать эти аспекты в отдельной спецификации.*

каждом обращении к соответствующей записи таблицы, а результат измерения («цвет») будет храниться в поле, заданном атрибутом result. Атрибут result является обязательным для измерителей с атрибутом direct, а ссылки на такие измерители недопустимы в примитивах execute_meter и компилятор должен сообщать об ошибке при обнаружении таких ссылок.

Если измеритель объявлен с атрибутом static, его можно указывать лишь в действиях, вызываемых соответствующей таблицей через примитив execute_meter. Компилятор должен выдавать сообщение об ошибке при вызове такого измерителя из действия другой таблицы.

Атрибут instance_count указывает число выделяемых экземпляров (ячеек) измерителя и требуется для измерителей без атрибута direct.

7.3 Регистры

Регистры представляют собой память состояний и их значения доступны для считывания и записи из действий. Регистры похожи на счётчики, но могут использоваться для хранения состояний в более общем смысле. Простым примером может служить проверка обнаружения «первого пакета» определённого типа потока. Ячейка регистра выделяется для потока и инициализируется пустой (clear). Когда протокол сообщает о «первом пакете», таблица будет соответствовать этому значению (пустому) и обновит ячейку потока до состояния «активный» (marked). Следующие пакеты потока будут сопоставляться с той же ячейкой и текущее значение ячейки будет сохраняться в метаданных для пакета, а следующая таблица может убедиться, что поток был помечен как активный.

Объявление регистра похоже на объявления счётчиков и измерителей. Регистры объявляются с атрибутом width, указывающим число битов в каждой ячейке.

```
register_declaration ::=
        register register_name {
                width_declaration ;
                [ direct_or_static ; ]
                [ instance_count : const_value ; ]
                [ attribute_list ; ]
        }
width_declaration ::= width : const_value ;
attribute_list ::= attributes : attr_entry
attr_entry ::= signed | saturating | attr_entry , attr_entry
```

Атрибут instance_count указывает число экземпляров (ячеек), создаваемых для регистра. Этот атрибут требуется для регистров без атрибута direct.

Доступ к регистрам возможен с помощью примитивов register_read и register_write, описанных ниже.

8 Таблицы Match+Action (СД)

P4 позволяет задавать экземпляры таблиц с помощью объявления таблиц. Объявление задаёт точный набор полей, которые следует проверять при сопоставлении. С каждой записью таблицы связывается действие, выполняемое при совпадении (соответствии записи).

Если совпадения не найдено в таблице (miss), применяется заданное по умолчанию действие.

Каждая запись таблицы «сопоставление-действие (СД, match+action) имеет перечисленные ниже компоненты.

- Значения для сопоставления с разобранным представлением (Parsed Representation) пакета. Формат этих значений задаёт объявление таблицы.
- Ссылка на функцию действия (action), выполняемого при совпадении. Набор разрешённых действий задаётся в объявлении таблицы.
- Значения параметров, передаваемые действию при вызове его функции. Формат параметров определяется конкретной функцией, выбранной для записи.

9 Действия

В P4 действия объявляются как функции, имена которых используются при заполнении таблиц при работе для выбора связанных с записями таблицы действий. Эти действия называют составными действиями (compound action), чтобы отличить от примитивов, если различие не очевидно из контекста.

Функции действий принимают параметры. Передаваемые параметрам значения программируются в записи таблицы с помощью run-time API. Когда запись при сопоставлении соответствует пакету, заданные параметры передаются действию. Объявления таблиц P4 могут использоваться для генерации run-time API, которые будут иметь параметры, соответствующие параметрам действия в записи. Обычно компилятор принимает на себя ответственность за корректное сопоставление этих значений в run-time API и программе P4.

В дополнение к значениям из соответствующей записи таблицы действие имеет доступ к заголовкам и метаданным из Parsed Representation.

Функции действий создаются из примитивов действий. Стандартный набор примитивов описан в следующем параграфе, но платформы могут добавлять свои примитивы. Однако использование специфических для платформы примитивов ограничивает переносимость программы.

Ниже приведены две функции из примера mTag. Первая функция задаёт отправку копии пакета в CPU. Параметры cpu_code и bad_packet раскрываются API и устанавливаются в соответствии со значениями, представленными при добавлении записи в таблицу.

```
// Отправка копии пакета в CPU.
action common_copy_pkt_to_cpu(cpu_code, bad_packet) {
        modify_field(local_metadata.copy_to_cpu, 1);
        modify_field(local_metadata.cpu_code, cpu_code);
```

```
        modify_field(local_metadata.bad_packet, bad_packet);
}
```

Эта функция устанавливает тег mTag и вызывается только на краевых коммутаторах.

```
// Добавление в пакет тега mTag и выбор выходной спецификации по up1.
action add_mTag(up1, up2, down1, down2) {
        add_header(mtag);
        // Копирование VLAN ethertype в mTag
        modify_field(mtag.ethertype, vlan.ethertype);
        // Установка VLAN ethertype для сигнализации mTag
        modify_field(vlan.ethertype, 0xaaaa);
        // Добавление в тег информации о заданной отправителем маршрутизации
        modify_field(mtag.up1, up1);
        modify_field(mtag.up2, up2);
        modify_field(mtag.down1, down1);
        modify_field(mtag.down2, down2);
        // Установка выходного порта для получателя по тегу
        modify_field(standard_metadata.egress_spec, up1);
}
```

9.1 Примитивы действий

P4 включает набор стандартных примитивов действий (операций). Конкретная платформа может поддерживать дополнительные примитивы действий. Способ раскрытия этих дополнительных примитивов FE-компилятору для семантической проверки выходит за рамки спецификации.

Некоторые целевые платформы поддерживают не все стандартные примитивы действий. Коммутатор может ограничивать привязки переменных и разрешённые комбинации параметров.

В таблице приведена краткая сводка стандартных примитивов действий, а более подробные описания даны ниже.

Таблица 4. Примитивы действий.

Имя в API	***Описание***
add_header	Добавляет заголовок в представление разобранного пакета (Parsed Representation).
copy_header	Копирует один экземпляр заголовка в другой.
remove_header	Помечает экземпляр заголовка как недействительный (invalid).
modify_field	Задаёт значение поля в представлении разобранного пакета (Parsed Representation).
add_to_field	Прибавляет значение к полю.
add	Складывает два значения и помещает результат в поле.
subtract_from_field	Вычитает значение из поля.
subtract	Вычитает одно значение из другого и помещает результат в поле.
modify_field_with_hash_based_offset	Выполняет расчёт для списка полей и использует результат для генерации величины смещения.
modify_field_rng_uniform	Генерирует случайное значение из заданного диапазона и помещает результат в поле.
bit_and	Выполняет побитовую операцию И (AND) для двух значений и помещает результат в поле.
bit_or	Выполняет побитовую операцию ИЛИ (OR) для двух значений и помещает результат в поле.
bit_xor	Выполняет побитовую операцию Исключительное-ИЛИ (XOR) для двух значений и помещает результат в поле.
shift_left	Выполняет операцию сдвига dst = value1 << value2.
shift_right	Выполняет операцию сдвига dst = value1 >> value2.
truncate	Выполняет отсечку пакета на выходе.
drop	Отбрасывает пакет (во входном конвейере).
no_op	Пустая операция.
push	Проталкивает все экземпляры заголовков «вниз» по массиву (стеку) и добавляет новый заголовок на вершину.
pop	Выталкивает заголовок с вершины массива (стека), перемещая оставшиеся заголовки на одну позицию вверх.
count	Обновляет значение счётчика.
execute_meter	Выполняет операцию измерения.
register_read	Считывание указанного индексом экземпляра регистра и запись полученного значения в поле.
register_write	Запись значения в указанный индексом регистр.
generate_digest	Генерация дайджеста (подписи) пакета и отправка принимающему.
resubmit	Повторное представление исходного пакета с метаданными синтаксическому анализатору.
recirculate	Повторное представление обработанного пакета.
clone_ingress_pkt_to_ingress	Передача копии исходного пакета синтаксическому анализатору. Псевдоним clone_i2i.
clone_egress_pkt_to_ingress	Передача копии обработанного пакета синтаксическому анализатору. Псевдоним clone_e2i.
clone_ingress_pkt_to_egress	Передача копии исходного пакета механизму буферизации (Buffer Mechanism). Псевдоним clone_i2e.
clone_egress_pkt_to_egress	Передача копии обработанного пакета механизму буферизации (Buffer Mechanism). Псевдоним clone_e2e.

Типы параметров действий указаны в таблице.

Таблица 5. Примитивы действий.

Обозначение	***Описание***
HDR	Литеральное имя экземпляра заголовка.
ARR	Имя массива экземпляров заголовков без индекса.
FLD	Ссылка на поле в представлении разобранного пакета (Parsed Representation) в форме header_instance.field_name
FLDLIST	Экземпляр списка полей, объявленного с field_list.
VAL	Промежуточное значение или значение из параметров действия в записи таблицы. Во втором случае представляется как параметр включающей функции (см. примеры ниже).
C-REF	Имя массива счётчиков (определяется при компиляции).
M-REF	Имя массива измерителей (определяется при компиляции).
R-REF	Имя массива регистров (определяется при компиляции).
FLC-REF	Ссылка на расчёт для списка полей (определяется при компиляции).

Ниже приведены спецификации API для стандартных примитивов действий.

add_header(header_instance)

Добавляет заголовок в представление проанализированного пакета (Parsed Representation).

header_instance

(HDR) Имя добавляемого экземпляра заголовка.

Для указанного экземпляра заголовка устанавливается бит valid. Если заголовок был недействительным, все его поля инициализируются значением 0, а для действительного ранее заголовка поля сохраняются.

copy_header(destination, source)

Копирует один экземпляр заголовка в другой.

destination

(HDR) Имя целевого экземпляра заголовка.

source

(HDR) Имя исходного экземпляра заголовка.

Все поля копируются из заголовка source в заголовок destination. Если исходный заголовок недействителен, destination также станет недействительным. Экземпляры source и destination должны быть однотипными.

remove_header(header_instance)

Помечает заголовок как недействительный (invalid).

header_instance

(HDR) Имя удаляемого экземпляра заголовка.

Указанный заголовок помечается как недействительный и становится недоступным для последующих этапов СД (match+action), а также не будет преобразовываться в последовательность (serialize) на выходе. Все поля экземпляра заголовка становятся неинициализированными.

modify_field(dest, value [, mask])

Устанавливает значение заданного поля в представлении проанализированного пакета (Parsed Representation).

dest

(FLD) Имя изменяемого поля заголовка (получатель).

value

(VAL или FLD) Устанавливаемое значение (источник).

mask

(VAL) Необязательная маска для указания изменяемых битов.

Обновляет значение указанного поля. В качестве параметра value может служить:

- непосредственное значение (число);
- значение из параметров действия в совпадающей записи (в этом случае используется имя параметра из включающей его функции);
- ссылка на поле Parsed Representation.

Этот примитив позволяет программисту копировать одно поле в другое. Если размер, значение или маска источника больше поля dest, старшие биты источника отсекаются. Если размер источника меньше размера получателя, значение будет расширено с учётом знака. Если родительский экземпляр заголовка недействителен, операция приведёт к неопределённому значению dest. При наличии маски выполняется операция (current_value & ~ mask) | (value & mask). При отсутствии маски устанавливаются все биты получателя.

add_to_field(dest, value)

Добавляет значение к полю.

dest

(FLD) Имя изменяемого поля заголовка (получатель).

value

(VAL или FLD) Устанавливаемое значение (источник).

Значение поля dest обновляется путём сложения со значением параметра value, которым может служить параметр таблицы, непосредственное значение или ссылка на поле (см. modify_field). Если value является ссылкой на поле недействительного заголовка, значение dest становится неопределённым. Логическое поведение описано в параграфе 9.1.1 Назначение полей и атрибуты насыщения. Если value содержит непосредственное значение, оно может быть отрицательным.

add(dest, value1, value2)

Складывает value1 и value2, сохраняя сумму в dest.

dest

(FLD) Имя изменяемого поля (получатель).

value1

(VAL или FLD) Первое слагаемое.

value2

(VAL или FLD) Второе слагаемое.

В поле dest записывается сумма двух параметров value, каждый из которых может быть параметром таблицы, непосредственным значением или ссылкой на поле (см. modify_field). Если любой из параметров value является ссылкой на поле недействительного заголовка, значение dest становится неопределённым. Логическое поведение

описано в параграфе 9.1.1 Назначение полей и атрибуты насыщения. Если value содержит непосредственное значение, оно может быть отрицательным.

subtract_from_field(dest, value)

Вычитает значение из поля.

dest

(FLD) Имя изменяемого поля заголовка (получатель).

value

(VAL или FLD) Вычитаемое значение.

Значение поля dest обновляется путём вычитания значения параметра value, которым может служить параметр таблицы, непосредственное значение или ссылка на поле (см. modify_field). Если value является ссылкой на поле недействительного заголовка, значение dest становится неопределённым. Логическое поведение описано в параграфе 9.1.1 Назначение полей и атрибуты насыщения. Если value содержит непосредственное значение, оно может быть отрицательным.

subtract(dest, value1, value2)

Вычитает value2 из value1 и записывает результат в поле dest.

dest

(FLD) Имя изменяемого поля (получатель).

value1

(VAL или FLD) Уменьшаемое значение.

value2

(VAL или FLD) Вычитаемое значение.

В поле dest записывается разность (value1 - value2) двух параметров value, каждый из которых может быть параметром таблицы, непосредственным значением или ссылкой на поле (см. modify_field). Если любой из параметров value является ссылкой на поле недействительного заголовка, значение dest становится неопределённым. Логическое поведение описано в параграфе 9.1.1 Назначение полей и атрибуты насыщения. Если value содержит непосредственное значение, оно может быть отрицательным.

modify_field_with_hash_based_offset(dest, base, field_list_calc, size)

Рассчитывает хэш-значение для списка полей.

dest

(FLD) Имя изменяемого поля (получатель).

base

(VAL) База для сложения с хэш-значением.

field_list_calc

(FLC-REF) Используемый для расчёта. метод.

size

(VAL) Размер диапазона хэш-значений (должен быть положительным числом).

Выполняется расчёт хэш-значения для списка полей. Результат лежит в диапазоне base - (base + size - 1) и рассчитывается как (base + (hash_value % size)). Результат расчёта. можно применять в качестве дайджеста (подписи) длинного ключа (например, кортежа с 5 элементами) или смещения (индекса) для ссылки на массив регистров. При записи результата в dest выполняется обычное преобразование значений.

Если любое из полей для field_list_calc относится к недействительному заголовку, это поле исключается из списка при расчёте хэш-значения.

modify_field_rng_uniform(dest, lower_bound, upper_bound)

Генерирует случайное значение из заданного диапазона.

dest

(FLD) Имя изменяемого поля (получатель).

lower_bound

(VAL или FLD) Нижняя (включительная) граница диапазона.

upper_bound

(VAL или FLD) Верхняя (включительная) граница диапазона.

Случайное значение выбирается из диапазона [lower_bound - upper_bound] и записывается в dest. Платформы могут вносить ограничения для lower_bound и upper_bound, например, требовать lower_bound = 0 и upper_bound = 2^n - 1.

Если любой из параметров lower_bound или upper_bound является ссылкой на поле недействительного заголовка, значение dest будет неопределённым.

bit_and(dest, value1, value2)

Выполняет побитовую операцию AND (И) для двух параметров.

dest

(FLD) Имя изменяемого поля (получатель).

value1

(VAL или FLD) Первый параметр.

value2

(VAL или FLD) Второй параметр.

Значение поля dest обновляется результатом побитовой операции AND для значений value1 и value2, каждое из которых может быть параметром таблицы, непосредственным значением или ссылкой на поле (см. modify_field). Если любой из параметров value является ссылкой на поле недействительного заголовка, значение dest становится неопределённым. Целевые платформы могут требовать одинакового размера dest, value1 и value2.

bit_or(dest, value1, value2)

Выполняет побитовую операцию OR (ИЛИ) для двух параметров.

dest

(FLD) Имя изменяемого поля (получатель).

value1

(VAL или FLD) Первый параметр.

value2

(VAL или FLD) Второй параметр.

Значение поля dest обновляется результатом побитовой операции OR для значений value1 и value2, каждое из которых может быть параметром таблицы, непосредственным значением или ссылкой на поле (см. modify_field).

Если любой из параметров value является ссылкой на поле недействительного заголовка, значение dest становится неопределённым. Целевые платформы могут требовать одинакового размера dest, value1 и value2.

bit_xor(dest, value1, value2)

Выполняет побитовую операцию XOR (Исключительное-ИЛИ) для двух параметров.

dest

(FLD) Имя изменяемого поля (получатель).

value1

(VAL или FLD) Первый параметр.

value2

(VAL или FLD) Второй параметр.

Значение поля dest обновляется результатом побитовой операции XOR для значений value1 и value2, каждое из которых может быть параметром таблицы, непосредственным значением или ссылкой на поле (см. modify_field). Если любой из параметров value является ссылкой на поле недействительного заголовка, значение dest становится неопределённым. Целевые платформы могут требовать одинакового размера dest, value1 и value2.

shift_left(dest, value1, value2)

Выполняет побитовый сдвиг влево параметра value1 на value2 позиций.

dest

(FLD) Имя изменяемого поля (получатель).

value1

(VAL или FLD) Значение для сдвига.

value2

(VAL или FLD) Размер сдвига (положительное значение).

Поле dest получает результат сдвига value1 << value2. Каждый из параметров может быть параметром таблицы, непосредственным значением или ссылкой на поле (см. modify_field). Если любой из параметров value является ссылкой на поле недействительного заголовка, значение dest становится неопределённым. Параметр value2 должен иметь положительное значение.

shift_right(dest, value1, value2)

Выполняет побитовый сдвиг вправо параметра value1 на value2 позиций.

dest

(FLD) Имя изменяемого поля (получатель).

value1

(VAL или FLD) Значение для сдвига.

value2

(VAL или FLD) Размер сдвига (положительное значение).

Поле dest получает результат сдвига value1 >> value2. Каждый из параметров может быть параметром таблицы, непосредственным значением или ссылкой на поле (см. modify_field). Если любой из параметров value является ссылкой на поле недействительного заголовка, значение dest становится неопределённым. Параметр value2 должен иметь положительное значение.

truncate(length)

Отсекает пакет на выходе.

length

(VAL) Задаёт число передаваемых байтов.

Задаёт отсечку размера выходного пакета до указанного параметром числа байтов. Если пакет меньше заданного размера, он не изменяется.

Обычно (но не обязательно) это действие применяется в выходном конвейере.

drop()

Задаёт отбрасывание пакета на выходе.

Указывает, что пакет не следует передавать. Примитив предназначен для выходного конвейера, где он является единственным способом указать, что передавать пакет не следует. Во входном конвейере этот примитив эквивалентен установке в метаданных egress_spec значения drop (зависит от платформы).

При вызове примитива из входного конвейера обработка пакета продолжится до завершения конвейера и последующие таблицы могут изменить значение egress_spec, отменив отбрасывание. Действие невозможно переопределить в выходном конвейере.

no_op()

Пустая операция.

Никаких действий по отношению к пакету не выполняется и поток управления продолжается в соответствии с текущей функцией.

push(array, count)

Проталкивает имеющиеся в массиве (стеке) заголовки «вниз» и добавляет в начало (на вершину) новые действительные заголовки.

array

(ARR) Имя изменяемого экземпляра массива.

count

(VAL) Число вталкиваемых в массив элементов.

Этот примитив служит для инициализации элементов на вершине стека заголовков. Имеющиеся элементы смещаются на count позиций в стеке, т. е. элемент с индексом N получит индекс N+count и в стек будет добавлено count действительных элементов с индексами от 0 до count-1. Примитив не меняет размер массива и элементы, индекс которых выходит за предел размера массива, будут потеряны.

pop(array, count)

Выталкивает экземпляры заголовков из начала массива (вершины стека), перемещая оставшиеся в массиве заголовки вверх.

array

(ARR) Имя изменяемого экземпляра массива.

count

(VAL) Число выталкиваемых из массива элементов.

Этот примитив служит для удаления (выталкивания) элементов из стека заголовков. Элемент с индексом N перемещается в позицию N-count, а элементы с индексами от 0 до count-1 удаляются из массива (теряются).

Пустые элементы в конце массива становятся недействительными. Выталкивание элементов из пустого массива или выталкивание большего числа элементов, нежели имеется в стеке, приводит к возврату пустого массива.

count(counter_ref, index)

Обновляет значение счётчика.

counter_ref

(C-REF) Имя массива счётчиков.

index

(VAL) Смещение изменяемого счётчика в массиве.

Значение счётчика пакетов увеличивается на 1, а счётчика байтов - на размер пакета. Массив счётчиков определяется в момент компиляции. Индекс массива может быть параметром записи таблицы или определяться в момент компиляции. Ссылка на массив счётчиков с прямым отображением в этом примитиве является ошибкой.

execute_meter(meter_ref, index, field)

Выполняет операцию измерения и записывает результат в указанное параметром поле.

meter_ref

(M-REF) Имя массива измерителей.

index

(VAL) Смещение экземпляра измерителя в массиве (применимо только для измерителей типа indirect).

field

(FLD) Поле для записи результата измерения.

Выполняется операция измерения, указанная параметрами meter_ref и index. Для прямых измерителей (тип direct) параметр index игнорируется, поскольку нужную ячейку определяет запись таблицы. Размер пакета передаётся измерителю. Примитив обновляет состояние измерителя и полученные данные («цвет» пакета) возвращаются в параметре field. Если field относится к недействительному заголовку, состояние измерителя обновляется, но после этого отбрасывается.

register_read(dest, register_ref, index)

Читает данные из регистра.

dest

(FLD) Имя экземпляра поля для записи значения из регистра (получатель).

register_ref

(R-REF) Имя массива регистров.

index

(VAL) Смещение регистра в массиве (применимо лишь к регистрам типа indirect).

Значение регистра, указанного параметрами register_ref и index, считывается и помещается в поле dest. Для регистров с прямым доступом (тип direct) параметр index не принимается во внимание, поскольку ячейка определяется записью таблицы.

register_write(register_ref, index, value)

Записывает значение в регистр.

register_ref

(R-REF) Имя массива регистров.

index

(VAL) Смещение регистра в массиве (применимо лишь к регистрам типа indirect).

value

(VAL или FLD) Записываемое в регистр значение.

Значение параметра value записывается в регистр, указанный параметрами register_ref и index. Параметр value может быть параметром таблицы, непосредственным значением или ссылкой на поле (см. modify_field). Для регистров с прямым доступом параметр index не принимается во внимание, поскольку ячейка определяется записью таблицы.

generate_digest(receiver, field_list)

Создаёт дайджест пакета и передаёт её получателю.

receiver

(VAL) Неанализируемый (opaque) идентификатор получателя.

field_list

(FLDLIST) Список полей, включаемых в дайджест.

Указанный список полей заполняется данными из пакета и отправляется зависимым от платформы механизмом агенту, способному обработать такой объект. Описание возможных получателей выходит за рамки спецификации P4. Примером такого получателя может служить CPU (передача по каналу, используемому для передачи пакетов) или сопроцессор, подключенный к специальной шине.

Эту функцию можно применять также для операций самообновления, таких как изучение адресов (address learning).

resubmit(field_list)

Применяется во входном конвейере, помечая пакет для повторного представления синтаксическому анализатору.

field_list

(FLDLIST) Список ссылок на поля метаданных.

Примитив применяется только во входном конвейере и помечает пакет для повторного представления анализатору. Входная обработка помеченного пакета выполняется до завершения конвейера с целью заполнения требуемых полей метаданных. В конце конвейера данные исходного пакета представляются заново синтаксическому анализатору вместе со значениями полей из field_list, установленными в процессе обработки. Эти значения переопределяют исходные метаданные, созданные инициализатором при объявлении экземпляра. В параметре field_list могут указываться лишь поля метаданных (но не заголовка). При неоднократном вызове примитива применяется лишь список полей из последнего действия resubmit и повторно представляется лишь один пакет.

В представленном повторно пакете устанавливается instance_type для индикации повтора (см. раздел 14 Рециркуляция и клонирование).

recirculate(field_list)

На выходе помечает пакет для повторного представления синтаксическому анализатору.

field_list

(FLDLIST) Список ссылок на поля метаданных.

Примитив применяется только в выходном конвейере и помечает пакет для повторной обработки (рециркуляции). Выходная обработка пакета выполняется до завершения конвейера и сборки пакета, после чего собранный заново пакет передаётся анализатору вместе с метаданными из field_list (на момент завершения выходной обработки). Эти значения метаданных переопределяют исходные метаданные, созданные инициализатором при объявлении экземпляра. В параметре field_list могут указываться лишь поля метаданных (но не заголовка). При неоднократном вызове примитива применяется лишь список полей из последнего действия recirculate и повторно представляется лишь один пакет.

В представленном повторно пакете устанавливается instance_type для индикации повтора (см. раздел 14 Рециркуляция и клонирование).

clone_ingress_pkt_to_ingress(clone_spec, field_list)

Создаёт копию исходного пакета и представляет её входному анализатору.

clone_spec

(VAL) Неанализируемый (opaque) идентификатор, задающий дополнительные параметры клонирования.

field_list

(FLDLIST) Список ссылок на поля метаданных.

Это действие указывает коммутатору создание копии исходного пакета (до изменений, внесённых таблицами СД) и её представление анализатору в виде независимого экземпляра пакета. Действие может выполняться сразу при вызове примитива или откладываться до буферизации исходного пакета. Обработка исходного пакета продолжается независимо от клона.

Параметр clone_spec позволяет задать зависящие от платформы характеристики операции клонирования. Это может быть просто идентификатор сессии. Например, при клонировании может выполняться отсечка пакета по размеру, зависящему от сессии. Концепция сессии является необязательной и некоторые платформы могут игнорировать параметр.

В клонированном экземпляре устанавливается поле instance_type для указания клона.

Поля метаданных, указанные в field_list (со значениями на момент завершения входной обработки) копируются в Parsed Representation экземпляра клона. Эти значения метаданных переопределяют исходные метаданные, созданные инициализатором при объявлении экземпляра (до синтаксического анализа). В параметре field_list могут указываться лишь поля метаданных (но не заголовка).

Этот примитив также имеет псевдоним clone_i2i (см. раздел 14 Рециркуляция и клонирование).

clone_egress_pkt_to_ingress(clone_spec, field_list)

Создаёт дубликат выходного пакета и представляет его анализатору.

clone_spec

(VAL) Неанализируемый (opaque) идентификатор, задающий дополнительные параметры клонирования.

field_list

(FLDLIST) Список ссылок на поля метаданных.

Пакет помечается для клонирования на выходе. После завершения обработки исходного пакета в выходном конвейере копия собранного заново пакета (с учётом всех изменений, внесённых таблицами СД) передаётся синтаксическому анализатору как независимый экземпляр пакета. Обработка исходного пакета продолжается обычным путём.

Параметр clone_spec позволяет задать зависящие от платформы характеристики операции клонирования, как указано для clone_ingress_pkt_to_ingress.

Поля метаданных, указанные в field_list (со значениями на момент завершения выходной обработки) копируются в экземпляр клона. Эти значения метаданных переопределяют исходные метаданные, созданные инициализатором при объявлении экземпляра (до синтаксического анализа). В параметре field_list могут указываться лишь поля метаданных (но не заголовка). В клонированном экземпляре устанавливается поле instance_type для указания клона.

Этот примитив также имеет псевдоним clone_e2i (см. раздел 14 Рециркуляция и клонирование).

clone_ingress_pkt_to_egress(clone_spec, field_list)

Создаёт копию исходного пакета и передаёт её механизму буферизации (Buffering Mechanism).

clone_spec

(VAL) Неанализируемый (opaque) идентификатор, задающий дополнительные параметры клонирования.

field_list

(FLDLIST) Список ссылок на поля метаданных.

Это действие указывает коммутатору создать копию исходного пакета. Когда функция выходного потока управления начнёт обработку клона, поля из field_list будут инициализированы значениями, которые имели метаданные в момент завершения функции входного потока управления для исходного пакета. В параметре field_list могут указываться лишь поля метаданных (но не заголовка).

Клон представляется напрямую механизму буферизации как независимый экземпляр пакета без обработки таблицами входного конвейера. Обработка исходного пакета продолжается обычным путём.

Параметр clone_spec позволяет задать зависящие от платформы характеристики клонирования, как указано для clone_ingress_pkt_to_ingress. В дополнение к другим атрибутам сессии clone_spec определяет стандартные метаданные egress_spec, представляемые механизму буферизации.

В клонированном экземпляре устанавливается поле instance_type для указания клона.

Этот примитив также имеет псевдоним clone_i2e (см. раздел 14 Рециркуляция и клонирование).

clone_egress_pkt_to_egress(clone_spec, field_list)

Дублирует выходную версию пакета и представляет её механизму буферизации.

clone_spec

(VAL) Неанализируемый (opaque) идентификатор, задающий дополнительные параметры клонирования.

field_list

(FLDLIST) Список ссылок на поля метаданных.

Пакет помечается для клонирования на выходе. По завершении обработки исходного пакета в выходном конвейере пакет и его Parsed Representation для заголовков (включая все изменения в таблицах СД) вместе с полями метаданных field_list (с учётом выходной обработки) представляются механизму буферизации как новый пакет. Обработка исходного пакета продолжается обычным путём. В параметре field_list могут указываться лишь поля метаданных (но не заголовка).

Параметр clone_spec позволяет задать зависящие от платформы характеристики операции клонирования, как указано для clone_ingress_pkt_to_ingress. В дополнение к другим атрибутам сессии clone_spec определяет стандартные метаданные egress_spec, представляемые механизму буферизации.
В клонированном экземпляре устанавливается поле instance_type для указания клона.
Этот примитив также имеет псевдоним clone_e2e (см. раздел 14 Рециркуляция и клонирование).

9.1.1 Назначение полей и атрибуты насыщения

Напомним, что при объявлении поля могут быть указаны атрибуты знака и насыщения, от которых зависит логика поведения add_to_field, как показано ниже для поля без знака.

```
tmp = field + value
if (field.saturating && tmp < field.min)
        field = field.min
else if (field.saturating && tmp > field.max)
        field = field.max
else
        field = tmp % 2^field.width
```

где

- field.saturating - логическое значение, указывающее насыщение поля;
- field.min - минимальное разрешённое значение, определяемое размером поля и наличием знака;
- field.max - максимальное разрешённое значение, определяемое размером поля и наличием знака;
- field.width - число битов в поле.

9.1.2 Привязка параметров

В некоторых из описанных выше примитивов параметры могут иметь разные формы:

- непосредственное значение;
- значение из данных параметра действия в записи таблицы;
- ссылка на экземпляр поля, значение которого будет использовано.

Язык P4 не задаёт ограничений на выбор между этими формами. Однако следует отметить качественные различия (в части разных функциональных требований к базовой платформе) между указанием конкретного экземпляра поля в программе P4 и возможностью run-time API задавать экземпляр поля для ссылки при добавлении записи в таблицу. Это вопрос привязки и в первом случае привязка выполняется в момент компиляции, а второй вариант позволяет привязывать значения в процессе работы. Платформы могут ограничивать гибкость привязки параметров. Различие должно отражаться в генерируемых интерфейсах run-time API.

9.2 Определение действий

Формат объявления действий показан ниже.

```
action_function_declaration ::=
        action action_header { action_statement * }
action_header ::= action_name "(" [ param_list ] ")"
param_list ::= param_name [, param_name]*
action_statement ::= action_name "(" [ arg [, arg]* ] ")" ;
arg ::= param_name | field_value | field_ref | header_ref
```

Ссылки на счётчики, измерители и регистры указываются с помощью индексов и передаются в аргументе param_name или field_value.

Объявление функции действия должно следовать приведённым ниже соглашениям.

- Все параметры, указанные в action_header, являются обязательными. Необязательных параметров не поддерживается.
- Тело функции содержит лишь:
  - вызовы примитивов действий;
  - вызовы функций других действий;
  - рекурсия не поддерживается.

В приведённом ниже примере параметры dst_mac, src_mac и vid раскрываются через runtime API для добавления записей в таблицу, использующую действие. Значения, передаваемые в API, будут установлены в таблице и могут быть переданы действию для пакетов, соответствующих записи.

```
action route_ipv4(dst_mac, src_mac, vid) {
        modify_field(ethernet.dst_addr, dst_mac);
        modify_field(ethernet.src_addr, src_mac);
        modify_field(vlan_tag.vid, vid);
        add_to_field(ipv4.ttl, -1);
}
```

9.2.1 Семантика последовательных и параллельных операций

В любой модели выполнения инструкций нужно указать последовательное или параллельное исполнение набора команд, чтобы задать поведение системы. Рассмотрим в качестве примера приведённые ниже операторы.

```
modify_field(hdr.fieldA, 1);
modify_field(hdr.fieldB, hdr.fieldA);
```

Предположим, что hdr.fieldA начинается со значения 0. Возникает вопрос - какое значение будет иметь hdr.fieldB после выполнения этого набора инструкций? При последовательной обработке по завершении первого оператора fieldA будет иметь значение 1 и второй оператор передаст это значение в поле fieldB. При параллельном исполнении обработка обоих операторов начнётся одновременно, поэтому hdr.fieldA во второй инструкции будет иметь значение 0 (поскольку оно ещё не изменено) и hdr.fieldB также получит значение 0.

P4 предполагает последовательное выполнение для всех примитивов действий, выполняемых в результате совпадения в данной таблице. Выполнение действий из разных таблиц также происходит последовательно и порядок операций определяется потоком управления, как описано в разделе 12 Обработка пакетов и поток управления.

10 Объявление профиля действий

В некоторых экземплярах значения параметров действий (action) не специфичны для совпадающей записи и могут совместно использоваться несколькими записями. Это можно выразить в P4 с помощью профилей действий, которые представляют собой объявляемую структуру, задающую список возможных действий, а также могущую включать иные атрибуты.

Записи помещаются в таблицу в процессе работы для указания одного действия (из числа указанных в профиле действий), которое будет выполняться при выборе данной записи, а также используемых значений параметров действия.

Вместо статической привязки одной записи из профиля действий к каждой записи таблицы сопоставления, можно связать несколько записей из профиля действий с одной записью таблицы сопоставления и позволить системе (т. е. логике плоскости данных) динамически связывать одну из записей профиля действий с каждым классом пакетов. Такое поведение включается атрибутом dynamic_action_selection. При задании этого атрибута записи профиля действий могут собираться в группы в процессе работы, а запись таблицы сопоставления можно связать с группой записей профиля действий. Для задания конкретного механизма плоскости данных, выбирающего определённую запись профиля действий в группе, нужно предоставить селектор действий. Этот селектор выбирает конкретную запись профиля действий для каждого пакета (псевдо)случайно или предсказуемо на основе полей заголовка и/или метаданных.

Ниже представлено определение профиля действий в формате BNF.

```
action_profile_declaration ::=
        action_profile action_profile_name {
                action_specification
                [ size : const_expr ; ]
                [ dynamic_action_selection : selector_name ; ]
        }
action_specification ::=
        actions { [ action_name ; ] + }
action_selector_declaration ::=
        action_selector selector_name {
                selection_key : field_list_calculation_name ;
}
```

Профили действий задаются и применяются в соответствии с приведённым ниже соглашением.

- Атрибут size указывает число записей, требуемых для профиля действий. Если заданный размер не поддерживается, при обработке объявления будет выдан сигнал об ошибке. При опущенном атрибуте размера нет гарантии создания нужного числа записей в профиле действий в процессе работы.

11 Объявления таблиц

Таблицы являются объявляемыми структурами, задающими операции сопоставления и действия, а также могущими включать иные атрибуты. Спецификация действия или профиля действий в таблице указывает, какие функции действия (операции) применимы для данной записи таблицы.

Объявление таблицы задаёт список полей, используемых для сопоставления с пакетами. Это могут быть ссылки на заголовки, бит действительности (valid) заголовка, или маскируемое поле заголовка. Маски в объявлении таблиц не следует путать с масками троичного сопоставления (ternary), они применяются к полю статически до операций поиска в таблице. Отметим, что поля сопоставления (ключи поиска) могут указываться в объявлении таблицы с масками. Это позволяет выполнять поиск в таблице по субполям, а не только по целому полю. Эти маски предназначены для таблиц с сопоставлением exact, позволяя в синтаксисе троичное сопоставления, хотя это является функционально избыточным.

Сопоставление всегда выполняется для конъюнкции (операция AND) всех полей ключа поиска, заданных в определении таблицы.

Ниже приведено объявление таблицы в формате BNF.

```
table_declaration ::=
        table table_name {
                [ reads { field_match + } ]
                table_actions
                [ min_size : const_value ; ]
                [ max_size : const_value ; ]
                [ size : const_value ; ]
                [ support_timeout : true | false ; ]
        }
field_match ::= field_or_masked_ref : field_match_type ;
field_or_masked_ref ::=
        header_ref | header_ref "." valid | field_ref | field_ref mask const_value
field_match_type ::= exact | ternary | lpm | range | valid
table_actions ::=
```

```
        action_specification | action_profile_specification
action_profile_specification ::=
        action_profile : action_profile_name ;
```

Приведённый ниже пример взят из программы краевого коммутатора mTag, где адрес получателя L2 из пакета отображается на mTag. Если отображения не удаётся найти, пакет может копироваться в CPU.

```
// Если в пакет нужно добавить mtag выполняется операция добавления тега.
table mTag_table {
        reads {
                ethernet.dst_addr      : exact;
                vlan.vid               : exact;
        }
        actions {
                add_mTag;      // Действие вызывается для добавления mtag.
                common_copy_pkt_to_cpu; // При отсутствии mtag пакет передаётся CPU
                no_op;
        }
        max_size        : 20000;
```

Реализация ECMP с использованием профиля действий рассмотрена в параграфе 15.8.3 Пример выбора ECMP.

Типы сопоставления описаны ниже.

exact
Значение поля должно точно совпадать со значением в таблице.

ternary
Для каждой записи в таблице предоставляется маска, которая применяется к сравниваемому полю до сравнения (операция AND). В результате совпадением считается соответствие битов, заданных маской. Поскольку такое сопоставление может давать несколько записей, для записей с сопоставлением ternary нужно задавать приоритет.

lpm
Максимально длинное совпадение префикса, которое является частным случаем сопоставления ternary, где маска содержит непрерывную последовательность 1 в старших битах и 0 в младших. Размер совпадающего префикса определяется числом единиц в маске и это число служит приоритетом для записи.

range
Каждая запись содержит минимальное и максимальное значение диапазона для соответствия записи таблицы (границы включаются в диапазон). Для определения порядка служит подпись поля.

valid
Этот тип сопоставления применим лишь для полей заголовков и экземпляров заголовков, но не для метаданных. В таблице должно быть указано значение true (соответствует действительному полю или заголовку) или false (соответствует недействительному полю или заголовку).

При объявлении и применении таблиц используются приведённые ниже соглашения.

- Ссылки на заголовки для сопоставления могут применяться только в сопоставлениях типа valid.
- При обработке пакета будет применяться в точности одна операция, указанная в action_specification или action_profile_specification.
    - Записи включаются в таблицу в процессе работы и каждое правило задаёт единственное действие в случае соответствия.
    - Действия в списке могут быть примитивами или составными (композитными).
- В процессе работы операция вставки записи в таблицу (не часть кода P4) должна задавать:
    - значения всех полей, указываемых в операции чтения (reads);
    - имя действия из action_specification или the action_profile_specification, а также параметры, передаваемые используемой действием функции.
- Используемое по умолчанию действие при отсутствии совпадающей записи. Это действие задаётся в процессе работы. Если такое действие не указано и соответствующей ключу записи не найдено в таблице, пакет остаётся без изменений и обработка продолжается в соответствии с потоком управления.
- Если операция чтения (reads) не применяется, таблица всегда будет использовать принятое по умолчанию действие. Если такое действие не задано, таблица не будет воздействовать на пакет.
- Может использоваться ключевое слово mask для сопоставления лишь некоторых битов поля. Маска применяется однократно к полю проанализированного представления (Parsed Representation) до операций сопоставления (в отличие от масок в записях, которые могут различаться у разных записей).
- Сопоставление valid показывает, что родительский заголовок поля (или само поле в случае метаданных) проверяется на действительность. Значение 1 будет соответствовать действительному полю, 0 — недействительному. Отметим, что поля метаданных всегда действительны.
- Атрибут min_size указывает минимальное число записей, которые нужны в таблице. Если это не поддерживается, разбор объявления таблицы будет давать ошибку.
- Атрибут max_size указывает максимальное число записей, которые предполагаются в таблице. Если в процессе работы достигнут максимум, новая операция вставки записи может быть отвергнута.
- Атрибут size эквивалентен заданию одинаковых значений min_size и max_size.
- Хотя атрибуты size и min_size не обязательны, отсутствие любого из них может приводить к пропуску (исключению) таблицы компилятором в соответствии с другими требованиями.
- Атрибут support_timeout служит для контроля устаревания таблицы. Он не обязателен и по умолчанию таблица не стареет (false).

Примитив действия no-op определён в параграфе 9.1 Примитивы действий и может применяться для записей, которым не нужно менять пакет.

12 Обработка пакетов и поток управления

Пакет обрабатывается цепочкой таблиц «сопоставление-действие (СД, match+action). В процессе настройки поток управления (порядок применения таблиц) может быть выражен императивной программой, которая может применять таблицы, вызывать другие функции потока управления и проверять условия.

Выполнение таблиц задаётся инструкциями apply, которые сами по себе могут воздействовать на поток управления, к которому относится пакет, путём задания набора блоков управления, один из которых выбирается для выполнения. Выбор такого блока может определяться действием для пакета или найденным для пакета совпадением.

Инструкции apply могут применяться в трёх режимах:

- последовательный, где поток управления безусловно переходит к следующему оператору;
- выбор действия, где выполненное действие определяет следующий блок применяемых инструкций;
- проверка совпадения или отсутствия, результат которой определяет следующий выполняемый блок.

Ниже приведены примеры каждого из режимов в формате BNF. В комбинациями с блоками if-else это обеспечивает механизм задания потока управления.

```
control_function_declaration ::=
        control control_fn_name control_block
control_block ::= { control_statement * }
control_statement ::=
        apply_table_call |
        apply_and_select_block |
        if_else_statement |
        control_fn_name ( ) ;

apply_table_call ::= apply ( table_name ) ;

apply_and_select_block ::= apply ( table_name ) { [ case_list ] }
case_list ::= action_case + | hit_miss_case +
action_case ::= action_or_default control_block
action_or_default ::= action_name | default
hit_miss_case ::= hit_or_miss control_block
hit_or_miss ::= hit | miss

if_else_statement ::=
        if ( bool_expr ) control_block
        [ else_block ]
else_block ::= else control_block | else if_else_statement
bool_expr ::=
        valid ( header_ref ) | bool_expr bool_op bool_expr |
        not bool_expr | ( bool_expr ) | exp rel_op exp | true | false

exp ::=
        exp bin_op exp | un_op exp | field_ref |
        value | ( exp )

bin_op ::= "+" | "*" | - | "<<" | ">>" | \& | "|" | ^
un_op ::= ~ | -
bool_op ::= or | and
rel_op ::= > | >= | == | <= | < | !=
```

Во многих случаях удобно задавать последовательность полей. Например, порядок выполнения и ассоциативность операторов соответствуют соглашениям языка C. Как указано в параграфе 4.2 Операция синтаксического анализа, анализатор возвращает имя функции управления для начала обработки СД. По завершении обработки пакет передаётся механизму управления очередями (если пакет не отбрасывается). При извлечении пакета из очереди выполняется выходная функция (egress), если она задана. Затем пакет передаётся в выходной порт, указанный полем метаданных egress_port field.

Таблицы вызываются для пакета с помощью оператора apply, как описано в начале этого раздела. Если одна таблица вызывается неоднократно в потоке управления, эти вызовы относятся к одному экземпляру таблицы, т. е. одному набору статистики, счётчиков, измерителей и операций СД для таблицы. Платформы могут вносить ограничения для вызовов таблиц, такие как запрет рекурсии, которые позволяют указывать таблицы и вызывать функции потока управления лишь однократно.

Простейшим примером потока управления является вызов цепочки таблиц с помощью оператора apply.

```
// Функция управления входным конвейером
control ingress {
        // Проверка согласованности mTag и порта
        apply(check_mtag);
        apply(identify_port);
        apply(select_output_port);
}
```

Оператор apply можно использовать для управления потоком инструкций на основе результатов поиска совпадений в таблице. Это осуществляется путём выполнения заключённого в скобки блока, следующего за оператором apply в для совпадающей (или miss) метки в блоке выбора. Программа mTag включает приведённый ниже пример.

```
apply(egress_meter) {
        hit { // При совпадении во входной таблице измерителей применяется правило
```

```
                apply(meter_policy);
        }
}
```

Кроме того, оператор apply позволяет управлять потоком инструкций на основе действия, применяемого таблицей к пакету. Например,

```
apply(routing_table) {
        ipv4_route_action { // Используется действие IPv4
                apply(v4_rpf);
                apply(v4_acl);
        }
        ipv6_route_action { // Используется действие IPv6
                apply(v6_option_check);
                apply(v6_acl);
        }
        default { // используется иное действие
                if (standard_metadata.ingress_port == 1) {
                        apply(cpu_ingress_check);
                }
        }
}
```

Отметим, что эти два режима (выбор совпадения или действия) нельзя смешивать. Они различаются потому, что зарезервированные слова hit и miss нельзя использовать в качестве имён функций для действий.

13 Выбор выходного порта, репликация, очередь

В P4 поле метаданных egress_spec служит для указания получателя или получателей пакета. Кроме того, для устройств, поддерживающих очереди, egress_spec может указывать связанную с каждым адресатом очередь. Значение egress_spec может представлять физический или логический (туннель, LAG, VLAN) порт или multicast-группу.

P4 предполагает, что механизм буферизации реализует функцию, сопоставляющую egress_spec с набором экземпляров пакета, представляемых триплетом (packet, egress_port, egress_instance). Механизм буферизации отвечает за создание экземпляров пакета вместе с полями метаданных и подобающую отправку в выходной порт через выходные таблицы СД.

Сопоставление egress_spec с набором экземпляров пакета в настоящее время выходит за рамки P4. Элемент пересылки может статически отображать значения на адресатов или разрешать настройку отображения через управляющий интерфейс. Интерфейсы программирования таблиц в процессе работы должны иметь доступ к этой информации для подобающего программирования таблиц, объявленных в программе P4.

Прохождение пакетов через элемент пересылки показано на рисунке 1. Напомним, что обработка делится на входную и выходную, между которыми может выполняться буферизация пакетов. Работа анализатора обычно завершается указанием функции управления, используемой для начала обработки. По завершении этой функции пакет представляется в систему буферизации. Предполагается, что буферы организованы в одну или множество очередей по выходным портам. Структура очередей и дисциплины управления ими считаются зависящими от целевой платформы, хотя эти платформы могут использовать P4 для раскрытия своей конфигурации (и даже для задания действий по событиям плоскости данных) в части поведения очередей.

Одна копия каждого пакета проходит через входной конвейер (Ingress Pipeline). По завершении входной обработки коммутатор определяет очередь (очереди) для размещения пакета на основе значения egress_spec. Пакет, имеющий множество адресатов, может быть помещён в несколько очередей.

При извлечении пакета из очереди он передаётся выходному конвейеру (Egress Pipeline) функцией управления выходом. Для каждого адресата в Egress Pipeline передаётся отдельная копия пакета, что требует репликации пакета механизмом буферизации. Физический выходной порт уже известен на момент извлечения пакета из очереди и его идентификатор передаётся через выходной конвейер как неизменяемое поле метаданных egress_port. Для поддержки передачи множества копий пакета в один физический порт (например, при наличии множества VLAN на порту), неизменяемое поле метаданных egress_instance содержит уникальный идентификатор для каждой копии. Семантика egress_instance зависит от платформы.

14 Рециркуляция и клонирование

Многие стандартные сетевые функции, такие как «зеркалирование» и рекурсивная обработка пакета, требуют более сложных примитивов для установки и проверки значений полей. Для поддержки таких операций в P4 имеются примитивы действий, которые позволяют организовать рециркуляцию (возврат в начало конвейера обработки) и клонирование (создание дополнительного экземпляра) пакетов. Отметим, что клонирование не является механизмом, который обычно используется для групповых пакетов (multicast). Предполагается, что обработка групповых пакетов обычно выполняется механизмом буферизации (Buffering Mechanism) вместе с выходным конвейером (13 Выбор выходного порта, репликация, очередь).

Приведённая ниже таблица 6 содержит набор операций. Первые 4 операции (префикс clone) создают полностью новый экземпляр пакета. Две последних операции работают с исходным пакетом, не создавая новых пакетов.

Таблица 6. Примитивы клонирования и рециркуляции.

Имя примитива	***Источник***	***Место вставки***
clone_ingress_pkt_to_ingress	Исходный входной пакет	Входной анализатор
clone_egress_pkt_to_ingress	Пакет после сборки	Входной анализатор
clone_ingress_pkt_to_egress	Исходный входной пакет	Механизм буферизации
clone_egress_pkt_to_egress	Пакет после сборки	Механизм буферизации
resubmit	Исходный входной пакет	Входной анализатор
recirculate	Пакет после сборки	Входной анализатор

14.1 Клонирование

Операции clone создают новый экземпляр пакета, а обработка исходного пакета продолжается как обычно. Термин «клонирование» (вместо «отражения» - mirror) для подчёркивания того, что это действие (action) отвечает лишь за создание копии пакета. «Отражение» требует дополнительной настройки, а механизм клонирования может иметь дополнительные применения.

Источником для клонирования может быть исходный экземпляр пакета (клонирование во входном конвейере) или выходящий из коммутатора пакет (клонирование на выходе). Обработка нового экземпляра может ограничиваться выходным конвейером (клон на выход) или выполняться с самого начала (клон на вход). В результате возникает 4 разных варианта, которые различаются значением поля метаданных instance_type.

При выполнении для пакета множества операций клонирования создаётся соответствующее число экземпляров. Однако некоторые платформы могут ограничивать число создаваемых клонов.

14.1.1 Клонирование на вход

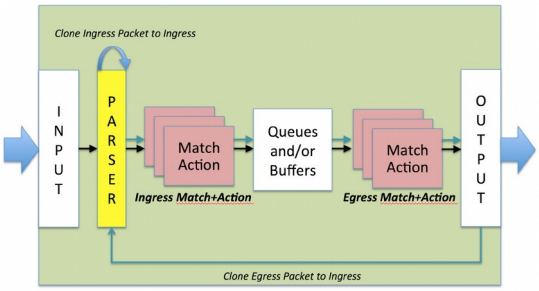

Рисунок 3. Клонирование на вход.

На рисунке 3 показаны пути пакетов, клонированных во входной конвейер. Источником пакета может быть входной или выходной конвейер.

14.1.2 Клонирование на выход

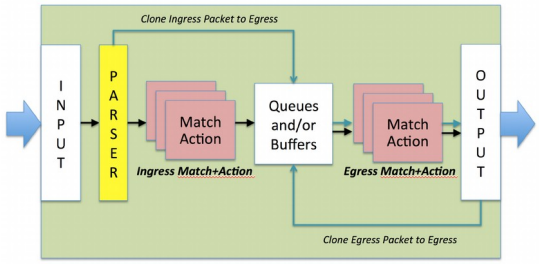

Рисунок 4. Клонирование на выход.

На рисунке 4 показаны пути пакетов, клонированных в выходной конвейер. Источником пакета может быть входной (копия пакета передаётся блоку буферизации) или выходной (копия пакета и его разобранное представление передаются блоку буферизации до выходной сборки).

Поскольку блоку буферизации нужна спецификация выхода (metadata.egress_spec) для управления обработкой пакета, эта спецификация должна быть связана с clone_spec для экземпляра примитивом операции. На некоторых платформах в качестве clone_spec может просто использоваться egress_spec.

14.1.3 Отражение

Отражение или мониторинга порта является стандартной сетевой функцией (см. например, http://en.wikipedia.org/wiki/Port_mirroring). В этом параграфе описан подход к реализации этой функции в P4. Мониторинг включает несколько этапов:

- идентификация пакетов для «отражения»;
- создание копий обнаруженных пакетов;
- задание действий, выполняемых с созданными копиями.

Обычно эти функции логически группируются в сеанс «отражения» (mirror session). В предположении минимальной поддержки со стороны платформы (например, предоставление внутренних метаданных) программа P4 может включать для поддержки входного мониторинга пакетов такие параметры как входной порт, VLAN ID, адрес L3 и протокол IP.

В приведённом примере предполагается, что блок буферизации обеспечивает программируемое сопоставление (отображение) параметра clone_spec, передаваемого примитиву clone_i2e, на номер выходного порта (egress_port). Сначала объявляется таблица для сопоставления по указанным параметрам, которая будет указывать действия вида

```
action mirror_select(session) { // Выбор пакетов и сопоставление с сессией
        modify_field(local_metadata.mirror_session, session);
        clone_i2e(session, mirror_fld_list);
}
```

где

```
field_list mirror_field_list {
        local_metadata.mirror_session;
}
```

указывает, что в клонированных пакетах должна указываться сессия отражения. Это действие создаёт новую копию входного пакета для представления в выходной конвейер. Интерфейс run-time API позволяет точно указать отображаемые (клонируемые) пакеты. Он также позволяет гибко выбирать идентификаторы сессий отражения для каждого такого пакета. Таблица mirror_select будет включена в поток управления для входного конвейера (возможно в начале обработки).

В выходной конвейер будет включена таблица local_metadata.mirror_session. Предположим, что значение 0 указывает отсутствие «отражения», поэтому таблица будет применяться для всех пакетов, но действия по отражению будут применены лишь к тем пакетам, которые помечены для мониторинга. Например,

```
action mirror_execute(trunc_length) {
        truncate(trunc_length);
}
```

В этом случае единственным действием является отсечка отображаемого пакета. Однако функция может включать данные, используемые для инкапсуляции заголовка, позволяющей передать каждый сеанс мониторинга в свой удалённый сеанс. Значения заголовков инкапсуляции будут создаваться в процессе работы.

Для отображения на выходе применяется похожий подход, основанный на примитиве clone_e2e.

14.2 Рециркуляция и повторное представление

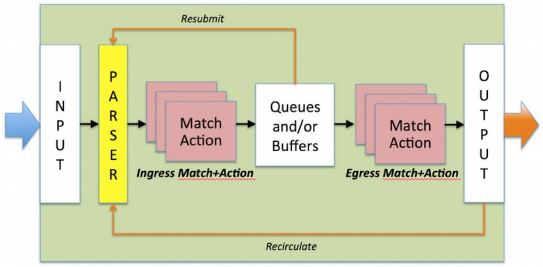

Рисунок 5. Рециркуляция и повторное представление.

На рисунке 5 показаны пути пакетов, повторно представляемых для обработки анализатору. Сверху показан путь пакетов повторного представления (resubmit), отправляемых действием входного конвейера. По завершении входного конвейера повторно представленный исходный пакет снова попадает в синтаксический анализатор вместе с метаданными, заданными действием. Анализатор может использовать эти метаданные для принятия при анализе решения, отличающегося от принятого в первом проходе.

Нижний путь показывает рециркуляцию пакета. После завершения входной и выходной обработки пакет собирается (возможно с использованием метаданных исходного пакета) и передаётся анализатору.

Операции resubmit и recirculate различаются по полю instance_type.

15 Приложения

15.1 Ошибки

- Синтаксис действий в определении таблицы не согласован. В спецификации предложено разделение элементов пробелами, но для согласованности следует использовать двоеточие.
- Синтаксис ссылок не счётчики не согласован со ссылками на измерители. В измерителях используются скобки, а в счётчиках - отдельный параметр.
- Механизм ссылок на вывод измерителей задан чересчур жёстко. Выход измерителя (поле метаданных с «цветом» от измерителя) разрешено указывать как при объявлении, так и при вызове измерителя.

15.2 Соглашения о программировании (не завершено)

Ниже приведены соглашения, предложенные для программ P4.

- Синтаксический анализ начинается с состояния анализатора start.
- Поток управления начинается со входной функции управления.

15.3 История выпусков

Таблица 7. История выпусков.

Выпуск	***Дата***	***Изменения***
1.0.0-rc1	08.09.2014	Первая открытая версия.
1.0.0-rc2	09.09.2014	Исправлены незначительные опечатки.
1.0.0-rc3	30.12.2014	Включены некоторые пропущенные символы обращения (отрицания, ~). Отбрасывание в анализаторе обозначено parser_drop. Добавлен примитив действия add. Добавлен раздел, указывающий ошибки.
1.0.1	28.01.2015	Добавлены селекторы профилей действий и действий, а также атрибут таблиц support_timeout.
1.0.2	03.03.2015	Добавлены примитивы действий push и pop.
1.0.3	03.11.2016	15.3.3 Изменения в версии 1.0.3
1.0.4	24.05.2017	15.3.2 Изменения в версии 1.0.4
1.0.5	31.05.2018	15.3.1 Изменения в версии 1.0.5

15.3.1 Изменения в версии 1.0.5

- 2.2.2 Стек заголовков
  - Изменена семантика стека заголовков для совместимости со спецификацией $P4_{16}$ и известными реализациями.

15.3.2 Изменения в версии 1.0.4

- 9.1 Примитивы действий
  - Изменена семантика примитивов действий, чтобы обращение к недействительному заголовку давало неопределённый результат и непригодные заголовки пропускались в расчётах по списку полей.
- 9.2.1 Семантика последовательных и параллельных операций
  - Семантика параллельных операций заменена на последовательную для соответствия имеющимся реализациям (например, bmv2).
- Проверка пригодности (2.2.1 Проверка пригодности экземпляров заголовков и метаданных)
  - Семантика изменена так, что поля непригодных заголовков становятся недействительными в операциях сопоставления с таблицами.
- 11 Объявления таблиц
  - Добавлено псевдополе hdr.valid, которое можно сопоставлять с любой таблицей.

15.3.3 Изменения в версии 1.0.3

- Счётчики и измерители (7.1 Счётчики и 7.2 Измерители)
  - Добавлен тип счётчиков packets_and_bytes.
  - Приведён пример цветового кодирования для измерителя.
  - Устранена неоднозначность при анализе путём замены meter() на execute_meter() (9.1 Примитивы действий).
- Схема регистров и методология доступа (7.3 Регистры и 9.1 Примитивы действий)
  - Отменена схема регистров.
  - Добавлены примитивы register_read и register_write вместо метода доступа на основе скобок.
- Объявление зависимых от платформы примитивов действий (9.1 Примитивы действий)
  - Дополнительные примитивы действий нельзя объявлять в коде P4. Отсутствие строгой типизации и указателей направления (in, out) делало такую возможность практически бесполезной.
- Новые примитивы действий (9.1 Примитивы действий)
  - Арифметические операторы subtract() и subtract_from_field().
  - Побитовые логические операторы bit_and(), bit_or() и bit_xor().
  - Операторы побитового сдвига shift_left() и shift_right().
  - Генератор случайных чисел modify_field_rng_uniform().
- Прочие изменения
  - Примитив modify_field_with_hash_based_offset() взамен set_field_to_hash_index (9.1 Примитивы действий).
  - Все параметры примитивов действий push, pop, resubmit, recirculate, clone_* стали обязательными.
  - Параметр field_list примитивов clone_* может задавать лишь поля метаданных, а не заголовков.
  - Разрешено пустое тело функции при объявлении действия (9.2 Определение действий).
  - Указан список внутренних метаданных, поддерживаемых прототипом BMv2 [3], в разделе 6 Стандартные внутренние метаданные.

15.4 Терминология (не завершено)

Таблица 8. Определения терминов.

Термин	*Перевод*	*Определение*
Control Flow	Поток управления	Логика выбора применяемых таблиц в конвейере, служащая для определения порядка.
Egress Queuing	Выходные очереди	Абстрактный функциональный блок P4, разделяющий входную и выходную обработку. Реализации могут раскрывать в этом блоке интерфейс управления очередями и ресурсами буферов, но спецификация P4 этого не задаёт.
Egress Specification	Спецификация выхода	Метаданные, устанавливаемые входным конвейером для указания набора выходных портов (и числа экземпляров для каждого порта), куда передаётся пакет.
Order Dependency	Зависимость порядка	Последовательность операций сопоставления и выполнения действий, которая задаёт порядок выполнения. Например, одна таблица может устанавливать поле, применяемое другой таблицей при сопоставлении. Поток управления служит для задания ожидаемых воздействий.

Parsed Representation	Разобранное представление	Представление заголовков пакета в форме набора экземпляров, каждый из которых состоит из полей.
Parser	Синтаксический анализатор	Функциональный блок, отображающий пакет на Parsed Representation.
Pipeline	Конвейер	Цепочка таблиц СД для обработки пакетов в процессе работы. Конвейера отличается от набора таблиц в момент настройки конфигурации, хотя в некоторых реализациях они могут совпадать.

15.5 P4 BNF

```
p4_program ::= p4_declaration +

p4_declaration ::=
        header_type_declaration |
        instance_declaration |
        field_list_declaration |
        field_list_calculation_declarationcalculated_field_declaration |
        value_set_declaration |
        parser_function_declaration |
        parser_exception_declaration |
        counter_declaration |
        meter_declaration |
        register_declaration |
        action_function_declaration |
        action_profile_declaration |
        action_selector_declaration |
        table_declaration |
        control_function_declaration

const_value ::= [ "+" | - ] [ width_spec ] unsigned_value
unsigned_value ::= binary_value | decimal_value | hexadecimal_value
binary_value ::= binary_base binary_digit+
decimal_value ::= decimal_digit+
hexadecimal_value ::= hexadecimal_base hexadecimal_digit+

binary_base ::= 0b | 0B
hexadecimal_base ::= 0x | 0X

binary_digit ::= _ | 0 | 1
decimal_digit ::= binary_digit | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
hexadecimal_digit ::=
        decimal_digit | a | A | b | B | c | C | d | D | e | E | f | F
width_spec ::= decimal_digit+ '
field_value ::= const_value
header_type_declaration ::=
        header_type header_type_name { header_dec_body }

header_dec_body ::=
        fields { field_dec + }
        [ length : length_exp ; ]
        [ max_length : const_value ; ]

field_dec ::= field_name : bit_width [ ( field_mod ) ];
field_mod ::= signed | saturating | field_mod , field_mod
length_bin_op ::= "+" | - | "*" | "<<" | ">>"
length_exp ::=
        const_value |
        field_name |
        length_exp length_bin_op length_exp |
        ( length_exp )
bit_width ::= const_value | "*"
instance_declaration ::= header_instance | metadata_instance
header_instance ::= scalar_instance | array_instance
scalar_instance ::= header header_type_name instance_name ;
array_instance ::=
        header header_type_name
                instance_name "[" const_value "]" ;

metadata_instance ::=
        metadata header_type_name
                instance_name [ metadata_initializer ] | ;

metadata_initializer ::= { [ field_name : field_value ; ] + }
header_ref ::= instance_name | instance_name "[" index "]"
index ::= const_value | last
field_ref ::= header_ref . field_name
field_list_declaration ::=
        field_list field_list_name {
                [ field_list_entry ; ] +
        }

field_list_entry ::=
        field_ref | header_ref | field_value | field_list_name | payload
field_list_calculation_declaration ::=
        field_list_calculation field_list_calculation_name {
                input {
```

```
                        [ field_list_name ; ] +
                }
                algorithm : stream_function_algorithm_name ;
                output_width : const_value ;
        }
calculated_field_declaration ::=
        calculated_field field_ref { update_verify_spec + }

update_verify_spec ::=
        update_or_verify field_list_calculation_name [ if_cond ] ;

update_or_verify ::= update | verify
if_cond ::= if ( calc_bool_cond )
calc_bool_cond ::=
        valid ( header_ref | field_ref ) |
        field_ref == field_value
value_set_declaration ::= parser_value_set value_set_name;
parser_function_declaration ::=
        parser parser_state_name { parser_function_body }

parser_function_body ::=
        extract_or_set_statement*
        return_statement

extract_or_set_statement ::= extract_statement | set_statement
extract_statement ::= extract ( header_extract_ref );
header_extract_ref ::=
        instance_name |
        instance_name "[" header_extract_index "]"

header_extract_index ::= const_value | next

set_statement ::= set_metadata ( field_ref, metadata_expr ) ;
metadata_expr ::= field_value | field_or_data_ref

return_statement ::=
        return_value_type |
        return select ( select_exp ) { case_entry + }

return_value_type ::=
        return parser_state_name ; |
        return control_function_name ; |
        parse_error parser_exception_name ;

case_entry ::= value_list : case_return_value_type ;
value_list ::= value_or_masked [ , value_or_masked ]* | default

case_return_value_type ::=
        parser_state_name |
        control_function_name |
        parse_error parser_exception_name

value_or_masked ::=
        field_value | field_value mask field_value | value_set_name

select_exp ::= field_or_data_ref [, field_or_data_ref] *
field_or_data_ref ::=
        field_ref |
        latest.field_name |
        current( const_value , const_value )
parser_exception_declaration ::=
        parser_exception parser_exception_name {
                set_statement *
                return_or_drop ;
        }

return_or_drop ::= return_to_control | parser_drop
return_to_control ::= return control_function_name
counter_declaration ::=
        counter counter_name {
                type : counter_type ;
                [ direct_or_static ; ]
                [ instance_count : const_expr ; ]
                [ min_width : const_expr ; ]
                [ saturating ; ]
        }

counter_type ::= bytes | packets | packets_and_bytes
direct_or_static ::= direct_attribute | static_attribute
direct_attribute ::= direct : table_name
static_attribute ::= static : table_name
meter_declaration ::=
        meter meter_name {
                type : meter_type ;
                [ result : field_ref ; ]
                [ direct_or_static ; ]
```

```
                [ instance_count : const_expr ; ]
        }

meter_type ::= bytes | packets
register_declaration ::=
        register register_name {
                width_declaration ;
                [ direct_or_static ; ]
                [ instance_count : const_value ; ]
                [ attribute_list ; ]
        }

width_declaration ::= width : const_value ;

attribute_list ::= attributes : attr_entry
attr_entry ::= signed | saturating | attr_entry , attr_entry
action_function_declaration ::=
        action action_header { action_statement * }

action_header ::= action_name "(" [ param_list ] ")"
param_list ::= param_name [, param_name]*
action_statement ::= action_name "(" [ arg [, arg]* ] ")" ;
arg ::= param_name | field_value | field_ref | header_ref

action_profile_declaration ::=
        action_profile action_profile_name {
                action_specification
                [ size : const_value ; ]
                [ dynamic_action_selection : selector_name ; ]
        }

action_specification ::=
        actions { [ action_name ] + }

action_selector_declaration ::=
        action_selector selector_name {
                selection_key : field_list_calculation_name ;
        }

table_declaration ::=
        table table_name {
                [ reads { field_match + } ]
                table_actions
                [ min_size : const_value ; ]
                [ max_size : const_value ; ]
                [ size : const_value ; ]
                [ support_timeout : true | false ; ]
        }
field_match ::= field_or_masked_ref : field_match_type ;
field_or_masked_ref ::=
        header_ref | header_ref "." valid | field_ref | field_ref mask const_value

field_match_type ::= exact | ternary | lpm | range | valid

table_actions ::=
        action_specification | action_profile_specification

action_specification ::=
        actions { [ action_name ] + }

action_profile_specification ::=
        action_profile : action_profile_name

control_function_declaration ::=
        control control_fn_name control_block
control_block ::= { control_statement * }
control_statement ::=
        apply_table_call |
        apply_and_select_block |
        if_else_statement |
        control_fn_name ( ) ;

apply_table_call ::= apply ( table_name ) ;

apply_and_select_block ::= apply ( table_name ) { [ case_list ] }
case_list ::= action_case + | hit_miss_case +
action_case ::= action_or_default control_block
action_or_default ::= action_name | default
hit_miss_case ::= hit_or_miss control_block
hit_or_miss ::= hit | miss

if_else_statement ::=
        if ( bool_expr ) control_block
        [ else_block ]

else_block ::= else control_block | else if_else_statement
```

```
bool_expr ::=
        valid ( header_ref ) | bool_expr bool_op bool_expr |
        not bool_expr | ( bool_expr ) | exp rel_op exp | true | false

exp ::=
        exp bin_op exp | un_op exp | field_ref |
        value | ( exp )

bin_op ::= "+" | "*" | - | "<<" | ">>" | \& | "|" | ^
un_op ::= ~ | -
bool_op ::= or | and
rel_op ::= > | >= | == | <= | < | !=
```

15.6 Зарезервированные слова P4

Ниже приведён список зарезервированных слов P4, которые не следует применять в идентификаторах[1].

```
apply
current
default
else
hit
if
last
latest
parse_error
payload
select
switch
```

15.7 Преобразование значений полей

Как отмечено в параграфе 1.5.1 Спецификации значений, может потребоваться преобразование значений, используемых в выражениях или назначенных экземпляру поля. Преобразование зависит от размера и наличия знака в исходном и целевом значении, а также насыщаемости целевого значения. Значение считается имеющим знак, если (1) оно имеет в начале явный символ - (минус) или является экземпляром поля, объявленным с атрибутом signed.

Правила преобразования приведены ниже.

- Если размеры источника и цели совпадают, используется двоичное представление источника, но интерпретация может измениться в зависимости от наличия знака:
  - 7-битовое значение без знака 127 при преобразовании в 7-битовое значение со знаком даст -1.
- Если размер источника меньше, значение расширяется с учётом наличия знака:
  - источник 7'b1111111 со знаком преобразуется 8-битовое значение 8'b11111111;
  - источник 4'b1100 без знака преобразуется в 8-битовое значение 8'b00001100.
- Если размер источника больше, результат зависит от насыщаемости получателя. Эффект должен быть таким же, как при сложении источника с нулём:
  - источник с отрицательным знаком при преобразовании в насыщаемый получатель получит значение 0;
  - источник без знака со значением 17 при преобразовании в 4-битовое значение без знака с насыщением даст в результате 15, поскольку это значение получателя при насыщении;
  - источник без знака со значением 17 при преобразовании в 4-битовое значение без знака и насыщения даст в результате 1, поскольку произойдёт переход в 0 при значении 15 (эквивалентно отсечке источника).

Для выражений определяется значение с большим размером и остальные значения приводятся к этому размеру с учётом наличия у них знака. Затем выражение вычисляется и результат преобразуется в соответствии с назначением.

15.8 Примеры

15.8.1 Аннотированный пример mTag

В этом параграфе представлен пример mTag, включающий две разных программы P4 - mtag-edge и mtag-aggregation, как указано в параграфе 1.2 Пример mTag. Код программ написан на языке P4, что позволяет использовать препроцессор C для файлов P4. В результате директивы #define и #include в программе дают такой же эффект, как в коде на языке C. Это удобно в примера, но язык P4 не требует применения такого синтаксиса.

Код примера разделен на несколько файлов, перечисленных ниже.

- headers.p4 содержит объявления всех типов заголовков, используемых обеими программами;
- parser.p4 содержит код анализатора, используемого обеими программами;
- actions.p4 описывает действия, используемые обеими программами;
- mtag-edge.p4 содержит программу для краевого коммутатора;
- mtag-aggregation.p4 содержит программу для коммутатора агрегирования;

Полностью программный код представлен на сайте P4 [2].

Рассмотрим сначала файл header.p4.

[1] *Вопрос фактического резервирования слов в P4 остаётся открытым.*

```
//*************************************************************
// Определения типов заголовков
//*************************************************************
// Стандартный заголовок L2 Ethernet
header_type ethernet_t {
        fields {
                dst_addr        : 48; // размер поля в битах
                src_addr        : 48;
                ethertype       : 16;
        }
}

// Стандартный тег VLAN
header_type vlan_t {
        fields {
                pcp             :3;
                cfi             :1;
                vid             :12;
                ethertype       :16;
        }
}

// Специальный тег mTag управления пересылкой на уровне агрегирования в ЦОД
header_type mTag_t {
        fields {
                up1             : 8; // От краевого коммутатора в агрегирующий
                up2             : 8; // От нижестоящего к вышестоящему коммутатору
                down1           : 8; // От вышестоящего к нижестоящему коммутатору
                down2           : 8; // От агрегирующего коммутатора к краевому
                ethertype       : 16;// Ethertype в инкапсулированном пакете
        }
}
// Стандартный заголовок IPv4
header_type ipv4_t {
        fields {
                version         : 4;
                ihl             : 4;
                diffserv        : 8;
                totalLen        : 16;
                identification  : 16;
                flags           : 3;
                fragOffset      : 13;
                ttl             : 8;
                protocol        : 8;
                hdrChecksum     : 16;
                srcAddr         : 32;
                dstAddr         : 32;
                options         : *;    // Опции переменного размера
        }
        length : ihl * 4;
        max_length : 60;
}

// Предполагаются стандартные метаданные от компилятора.


// Далее определяются локальные метаданные.
//
// copy_to_cpu является примером зависящих от платформы
// внутренних метаданных. Это особое значение, приводящее
// к копированию пересылаемого пакета в управляющий CPU.

header_type local_metadata_t {
        fields {
                cpu_code        : 16;   // Код для отправки пакета в CPU
                port_type       : 4;    // Тип порта: up, down, local...
                ingress_error   : 1;    // Ошибка при проверке входного порта
                was_mtagged     : 1;    // Отслеживание mTag на входе
                copy_to_cpu     : 1;    // особый код для копирования пакета в CPU
                bad_packet      : 1;    // Прочие ошибки
                color           : 8;    // Для измерителей
        }
}
```

Ниже приведён код анализатора, используемого обеими программами.

```
//*************************************************************
// Функция анализатора и связанные с ней определения
//*************************************************************
//*************************************************************

// Определения экземпляров заголовков
//
// Экземпляры заголовков обычно определяются в анализаторе,
// поскольку здесь они инициализируются.
//
//*************************************************************
```

```
header ethernet_t ethernet;
header vlan_t vlan;
header mTag_t mtag;
header ipv4_t ipv4;

// Объявление локального экземпляра метаданных.
metadata local_metadata_t local_metadata;

////////////////////////////////////////////////////////////////
// Описание состояний анализатора
////////////////////////////////////////////////////////////////

// анализ начинается с заголовка Ethernet.
parser start {
        return ethernet;
}

parser ethernet {
        extract(ethernet);     // Извлечение заголовка  Ethernet.
        return select(latest.ethertype) {
                0x8100:         vlan;
                0x800:          ipv4;
                default:        ingress;
        }
}

// Извлечение тега VLAN и проверка mTag
parser vlan {
        extract(vlan);
                return select(latest.ethertype) {
                0xaaaa:         mtag;
                0x800:          ipv4;
                default:        ingress;
        }
}

// mTag может размещаться только после тега VLAN
parser mtag {
        extract(mtag);
        return select(latest.ethertype) {
                0x800:          ipv4;
                default:        ingress;
        }
}

parser ipv4 {
        extract(ipv4);
        return ingress;        // Анализ завершён, начинается сопоставление
}
```

В следующем файле определены применяемые программами действия.

```
////////////////////////////////////////////////////////////////
//
// actions.p4
//
// Этот файл определяет общие действия, которые могут применяться
// краевым или агрегирующим коммутатором.
//
////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////
// Действия, используемые в таблицах
////////////////////////////////////////////////////////////////

// Копирование пакета в CPU;
action common_copy_pkt_to_cpu(cpu_code, bad_packet) {
        modify_field(local_metadata.copy_to_cpu, 1);
        modify_field(local_metadata.cpu_code, cpu_code);
        modify_field(local_metadata.bad_packet, bad_packet);
}

// Отбрасывание пакета; возможна отправка в CPU и маркировка (плохой)
action common_drop_pkt(do_copy, cpu_code, bad_packet) {
        modify_field(local_metadata.copy_to_cpu, do_copy);
        modify_field(local_metadata.cpu_code, cpu_code);
        modify_field(local_metadata.bad_packet, bad_packet);
        drop();
}
// Указание типа порта. См. mtag_port_type.
// Разрешает индикацию ошибок.
action common_set_port_type(port_type, ingress_error) {
```

```
        modify_field(local_metadata.port_type, port_type);
        modify_field(local_metadata.ingress_error, ingress_error);
}
```

Далее приведён фрагмент программы mtag-edge.

```
//////////////////////////////////////////////////////////////
//
// mtag-edge.p4
//
// Файл задаёт поведение краевого коммутатора в примере mTag.
//
//
//////////////////////////////////////////////////////////////

// Включение определений заголовков и анализатора (с экземплярами)
#include "headers.p4"
#include "parser.p4"
#include "actions.p4" // Для действий с префиксом common_

#define PORT_COUNT 64 // Число портов коммутатора

//////////////////////////////////////////////////////////////
// Определения таблиц
//////////////////////////////////////////////////////////////

// Удаление mtag для локальной обработки и коммутации.
action _strip_mtag() {
        // Вырезание тега из пакета.
        remove_header(mtag);
        // Сохранение информации о наличии тега.
        modify_field(local_metadata.was_mtagged, 1);
}
// Теги mtag всегда вырезаются на краевом коммутаторе
table strip_mtag {
        reads {
                mtag    : valid; // Тег mtag был найден?
        }
        actions {
                _strip_mtag;   // Вырезать mtag и записать метаданные.
                no_op;         // Пропустить без обработки.
        }
}

//////////////////////////////////////////////////////////////
// Идентификация входного порта: local, up1, up2, down1, down2
table identify_port {
        reads {
                standard_metadata.ingress_port : exact;
        }
        actions { // Каждая запись таблицы задаёт одно действие.
                common_set_port_type;
                common_drop_pkt;       // Неизвестный порт.
                no_op;                 // Продолжение обработки пакета.
        }
        max_size : 64; // Одно правило на порт.
}
. . .  // Здесь задан код локальной коммутации

// Добавление mTag в пакет и выбор выходной спецификации по up1
action add_mTag(up1, up2, down1, down2) {
        add_header(mtag);      // Копирование VLAN ethertype в mTag
        modify_field(mtag.ethertype, vlan.ethertype);
        // Установка VLAN ethertype для сигнализации mTag
        modify_field(vlan.ethertype, 0xaaaa);
        // Добавление маршрутной информации источника тега
        modify_field(mtag.up1, up1);
        modify_field(mtag.up2, up2);
        modify_field(mtag.down1, down1);
        modify_field(mtag.down2, down2);
        // Установка выходного порта получателя по данным тега
        modify_field(standard_metadata.egress_spec, up1);
}
// Чет пакетов и байтов
counter pkts_by_dest {
        type : packets;
        direct : mTag_table;
}
counter bytes_by_dest {
        type : bytes;
        direct : mTag_table;
}
// Проверка и при необходимости добавление mtag.
table mTag_table {
        reads {
                ethernet.dst_addr      : exact;
                vlan.vid               : exact;
```

```
        }
        actions {
                add_mTag;       // Действие для установки mtag.
                // Возможна отправка в CPU, если mtag не устанавливается
                common_copy_pkt_to_cpu;
                no_op;
        }
max_size        : 20000;
}
// Пакеты от уровня агрегирования должны оставаться локальными
table egress_check {
        reads {
                standard_metadata.ingress_port : exact;
                local_metadata.was_mtagged : exact;
        }
        actions {
                common_drop_pkt;
                no_op;
        }
        max_size : PORT_COUNT; // Не более 1 правила на порт
}

// Выходные измерения. Их можно выполнить напрямую, но здесь программе
// разрешено использовать отображение для привязки ячейки измерителя
// к паре отправитель-получатель
meter per_dest_by_source {
        type : bytes;
        result : local_metadata.color;
        instance_count : PORT_COUNT * PORT_COUNT;    // Для пары отпр.-получ.
}
action meter_pkt(meter_idx) {
        execute_meter(per_dest_by_source, meter_idx, local_metadata.color);
}

// Цветовая маркировка для восходящих (uplink) портов.
table egress_meter {
        reads {
                standard_metadata.ingress_port : exact;
                mtag.up1 : exact;
        }
        actions {
                meter_pkt;
                no_op;
        }
        size : PORT_COUNT * PORT_COUNT;      // Может быть меньше
}
// Применение правил измерителя
counter per_color_drops {
        type : packets;
        direct : meter_policy;
}

table meter_policy {
        reads {
                metadata.ingress_port : exact;
                local_metadata.color : exact;
        }
        actions {
                drop; //Автоматически учитывается приведённым выше прямым счётчиком
                no_op;
        }
        size : 4 * PORT_COUNT;
}

////////////////////////////////////////////////////////////////
// Определения функций управления
////////////////////////////////////////////////////////////////

// Входная функция управления
control ingress {
        // Всегда вырезать тег mtag, сохраняя состояние
        apply(strip_mtag);
        // Определение типа порта-источника
        apply(identify_port);
        // Продолжение, если нет ошибки source_check
        if (local_metadata.ingress_error == 0) {
                // Попытка коммутации конечному хосту
                apply(local_switching); // Сопоставление по адресу получателя (не показано)
                // Не локальная коммутация с попыткой установить mtag
                if (standard_metadata.egress_spec == 0) {
                        apply(mTag_table);
                }
        }
}

// Выходная функция управления
```

```
control egress {
        // Проверка неизвестного состояния на выходе и непригодного mTag.
        apply(egress_check);
        // Применение таблицы egress_meter и измерение в случае совпадения
        apply(egress_meter) {
                hit {
                        apply(meter_policy);
                }
        }
}
```

Ниже показана фрагмент программы mtag-aggregation.

```
//////////////////////////////////////////////////////////////
//
// mtag-aggregation.p4
//
//////////////////////////////////////////////////////////////

// Включение определений заголовков и анализатора (с экземплярами)
#include "headers.p4"
#include "parser.p4"
#include "actions.p4" // Для действий с префиксом common_

//////////////////////////////////////////////////////////////
// Таблица check_mtag
// Проверяется тег mtag и при отсутствии выполняется правило drop или to-cpu
//////////////////////////////////////////////////////////////
table check_mtag { // Статически запрограммировано с 1 записью
        . . . // Проверка пригодности тега mtag; отбрасывание или копия в CPU
}

//////////////////////////////////////////////////////////////
// Таблица identify_port
// Проверка направления порта (up, down) заданного при работе.
//////////////////////////////////////////////////////////////

table identify_port {
        . . . // Считывание входного порта, вызов common_set_port_type.
}

//////////////////////////////////////////////////////////////

// Действия по копированию нужного поля из mtag в выходную спецификацию
action use_mtag_up1() { // Не применяется агрегирующим коммутатором
        modify_field(standard_metadata.egress_spec, mtag.up1);
}
action use_mtag_up2() {
        modify_field(standard_metadata.egress_spec, mtag.up2);
}
action use_mtag_down1() {
        modify_field(standard_metadata.egress_spec, mtag.down1);
}
action use_mtag_down2() {
        modify_field(standard_metadata.egress_spec, mtag.down2);
}

// Таблица для выбора выходной спецификации по mtag
table select_output_port {
        reads {
                local_metadata.port_type      : exact; // Up, down (1 или 2).
        }
        actions {
                use_mtag_up1;
                use_mtag_up2;
                use_mtag_down1;
                use_mtag_down2;
                // Если тип порта не опознан, применяется предыдущее правила
                no_op;
        }
        max_size : 4; // Нужна лишь 1 запись на тип порта.
}

//////////////////////////////////////////////////////////////
// Определения управляющих функций
//////////////////////////////////////////////////////////////

// Входная функция управления
control ingress {
        // Проверка соответствия mTag и порта
        apply(check_mtag);
        apply(identify_port);
        apply(select_output_port);
}
// В примере mtag-agg не используется выходная функция управления.
```

Ниже представлен пример заголовочного файла, который можно применить в рассмотренном выше примере mtag. Этот файл включает ряд перечисленных ниже элементов.

- Определения типов для портов (mtag_port_type_t), уровня измерителей (mtag_meter_levels_t) и обработчиков в таблицах (entry_handle_t).
- Пример функции для добавления записи в таблицу identify_port (table_identify_port_add_with_set_port_type). Действие для использования в записи указано в конце имени функции set_port_type.
- Функции для установки принятого по умолчанию действия в таблице identify_port (table_indentify_port_default_common_drop_pkt и table_indentify_port_default_common_set_port_type).
- Функция для добавления записи в таблицу mTag (table_mTag_table_add_with_add_mTag).
- Функция для чтения счётчика, связанного с таблицей измерителя (counter_per_color_drops_get).

```
/**
* Пример заголовочного файла (run-time) для примера CCR mTag
*/

#ifndef MTAG_RUN_TIME_H
#define MTAG_RUN_TIME_H

/**
* @brief типы портов, требуемые для примера mtag
*
* Указывает типы портов для краевого и агрегирующего коммутатора.
*/
typedef enum mtag_port_type_e {
        MTAG_PORT_UNKNOWN,     /* Тип неинициализированного порта */
        MTAG_PORT_LOCAL,       /* Локальный порт краевого коммутатора*/
        MTAG_PORT_EDGE_TO_AG1,/* Up1 - от краевого коммутатора на уровень агрегирования 11 */
        MTAG_PORT_AG1_TO_AG2, /* Up2 - от уровня агрегирования 1 на уровень 2 */
        MTAG_PORT_AG2_TO_AG1, /* Down2 - от уровня агрегирования 2 на уровень 1 */
        MTAG_PORT_AG1_TO_EDGE,/* Down1 - от уровня агрегирования 1 в краевой коммутатор*/
        MTAG_PORT_ILLEGAL,     /* Недопустимое значение */
        MTAG_PORT_COUNT
} mtag_port_type_t;

/**
* @brief Цвета для измерителей
*
* Краевой коммутатор поддерживает измерение от локальных портов
* на уровень агрегирования.
*/
typedef enum mtag_meter_levels_e {
        MTAG_METER_COLOR_GREEN,       /* Нет перегрузки */
        MTAG_METER_COLOR_YELLOW,      /* Выше нижнего уровня */
        MTAG_METER_COLOR_RED,         /* Выше верхнего уровня */
        MTAG_METER_COUNT
} mtag_meter_levels_t;

typedef uint32_t entry_handle_t;

/* таблица mTag */
/**
* @brief Добавление записи в таблицу идентификации портов на краю
* @param ingress_port - определённый номер порта
* @param port_type - тип, связанный с портом
* @param ingress_error - значение для индикации ошибки
*/
entry_handle_t table_identify_port_add_with_set_port_type(
        uint32_t ingress_port,
        mtag_port_type_t port_type,
        uint8_t ingress_error);

/**
* @brief Установка принятого по умолчанию действия таблицы
* идентификации портов для передачи пакета в CPU.
* @param do_copy - 1, если нужно отправить копию пакета в CPU
* @param cpu_code - при установленном указывает код причины
* @param bad_packet - 1 для пометки пакета как «плохого»
*
* Это позволяет программисту сказать: «Если порт не указан,
* это ошибка и нужно посмотреть пакет».
*
* Можно также просто отбросить пакет.
*/

int table_indentify_port_default_common_drop_pkt(
        uint8_t do_copy,
        uint16_t cpu_code,
        uint8_t bad_packet);

/**
* @brief Установка принятого по умолчанию действия для
* таблицы идентификации портов.
* @param port_type - тип порта
* @param ingress_error - значение для индикации ошибки
*
```

```
* Это позволяет программисту сказать, что по умолчанию
* порт считается локальным
*/
int table_indentify_port_default_common_set_port_type(
        mtag_port_type_t port_type,
        uint8_t ingress_error);

/**
* @brief Добавляет запись в таблицу mtag.
* @param dst_addr - адрес L2 MAC для сопоставления
* @param vid - тег VLAN для сопоставления
* @param up1 - значение up1 для использования в mTag
* @param up2 - значение up2 для использования в mTag
* @param down1 - значение down1 для использования в mTag
* @param down2 - значение down2 для использования в mTag
*/
entry_handle_t table_mTag_table_add_with_add_mTag(
        mac_addr_t dst_addr, uint16_t vid,
        uint8_t up1, uint8_t up2, uint8_t down1, uint8_t down2);

/**
* @brief Возвращает число фактов отбрасывания по входным порта и цвету
* @param ingress_port - входной порт для запроса значения
* @param color - цвет для запроса значения
* @param count (output) - текущее значение параметра.
* @returns - 0 при успешном считывании.
*/
int counter_per_color_drops_get(
        uint32_t ingress_port,
        mtag_meter_levels_t color,
        uint64_t *count);
#endif /* MTAG_RUN_TIME_H */
```

15.8.2 Гистерезис при измерениях mTag с регистрами

В предыдущем параграфе коммутатор mtag-edge использовал измеритель между локальным портом и уровнем агрегирования. Предположим, что моделирование сети показывает преимущество использования гистерезиса для измерителей. Т. е. когда измеритель находится в состоянии red, пакеты отбрасываются, пока тот не перейдёт в состояние green (не yellow). Это можно реализовать путём добавления массива регистров в параллель с измерителями. Каждая ячейка массива будет хранить «предыдущий» цвет для измерителя. Для этого в программу нужно внести изменения. Индекс измерителя для удобства сохраняется в локальных метаданных.

```
//////////////////////////////////////////////////////////////////
//
// headers.p4 - добавляет индекс измерителя в локальные метаданные.
//
//////////////////////////////////////////////////////////////////
header_type local_metadata_t {
        fields {
                cpu_code        : 16;   // Код для пакета, передаваемого в CPU
                port_type       : 4;    // Тип порта - up, down, local...
                ingress_error   : 1;    // Ошибка при проверке входного порта
                was_mtagged     : 1;    // Отслеживать пакеты с mtag на входе
                copy_to_cpu     : 1;    // Специальный код для отправки копии в CPU
                bad_packet      : 1;    // Прочие ошибки
                color           : 8;    // Для измерителя
                prev_color      : 8;    // Для гистерезиса измерителя
                meter_idx       : 16;   // Индекс измерителя
        }
}

//////////////////////////////////////////////////////////////////
// mtag-edge.p4 - объявление регистров и добавление таблицы для их обновления
//////////////////////////////////////////////////////////////////

// Регистр сохраняет «предыдущее» состояние (цвет) измерителя.
// Индекс совпадает с индексом измерителя.
register prev_color {
        width : 8;      // Пара для указанного выше измерителя.
        instance_count : PORT_COUNT * PORT_COUNT;
}

// Обновление цвета, сохранённого в регистре
action update_prev_color(new_color) {
        register_write(prev_color, local_metadata.meter_idx, new_color);
}

// Изменение цвета в соответствии с параметром
action mark_pkt(color) {
        modify_field(local_metadata.color, color);
}

// Обновление измерителя пакетов для сохранения данных
action meter_pkt(meter_idx) {
        // Сохранение индекса и предшествующего цвета в метаданных пакета
        modify_field(local_metadata.meter_idx, meter_idx);
```

```
        register_read(local_metadata.prev_color, prev_color, meter_idx);
        execute_meter(per_dest_by_source, meter_idx, local_metadata.color);
}

//
// Эта таблица статически заполняется по приведённым ниже правилам
// color: green,       prev_color: red       ==> update_prev_color(green)
// color: red,         prev_color: green     ==> update_prev_color(red)
// color: yellow,      prev_color: red       ==> mark_pkt(red)
// Иначе no-op.
//
table hysteresis_check {
        reads {
                local_metadata.color : exact;
                local_metadata.prev_color : exact;
        }
        actions {
                update_prev_color;
                mark_pkt;
                no_op;
        }
        size : 4;
}
////////////////////////////////////////////////////////////////
// Проверка гистерезиса в выходной функции управления.
////////////////////////////////////////////////////////////////
control egress {
        // Проверка неизвестного состояния выхода или непригодного mTag.
        apply(egress_check);
        apply(egress_meter) {
                hit {
                        apply(hysteresis_check);
                        apply(meter_policy);
                }
        }
}
```

15.8.3 Пример выбора ECMP

Этот пример показывает возможную реализацию выбора среди равноценных путей (ECMP) с помощью профиля действий с селектором действия.

```
table ipv4_routing {
        reads {
                ipv4.dstAddr: lpm;
        }
        action_profile : ecmp_action_profile;
        size : 16384;  // 16K возможных префиксов IPv4
}
action_profile ecmp_action_profile {
        actions {
                nhop_set;
                no_op;
        }
        size : 4096;   // 4K возможных next hop
        dynamic_action_selection : ecmp_selector;
}
// Список полей, применяемых для определения ECMP next hop
field_list l3_hash_fields {
        ipv4.srcAddr;
        ipv4.dstAddr;
        ipv4.protocol;
        tcp.sport;
        tcp.dport;
}
field_list_calculation ecmp_hash {
        input {
                l3_hash_fields;
        }
        algorithm : crc16;
        output_width : 16;
}
action_selector ecmp_selector {
        selection_key : ecmp_hash;
}
```

15.9 Свойства, предлагаемые для будущих версий

Предполагается дальнейшее развитие языка P4 с реализацией новых возможностей и устранением имеющихся проблем. Постепенное обновление будет реализоваться в выпусках с измененным младшим номером версии. Ниже приведены свойства, рассматриваемые для реализации в будущих версиях P4.

Таблица 9. Предложения на будущее.

Название	***Суть предложения***
Поддержка операторов присваивания	Возможность манипулировать полями и заголовками с помощью операторов присваивания, таких как = или +=.

Поддержка типизации	Поддержка типов для данных и объектов.
Улучшенная поддержка инкапсуляции	Поддержка более эффективных примитивов действий и функций анализатора для инкапсуляции.
Изменение конфигурации при работе	Возможности языка и соглашения, позволяющие более эффективно и согласованно менять конфигурацию в процессе работы.
Псевдонимы полей и заголовков	Поддержка механизма, позволяющего ссылаться на разные поля или экземпляры заголовков опосредованно (через псевдонимы) для того, чтобы приложения могли задавать правила одновременно для разных форматов пакетов.
Гибкое включение функций	Добавить возможность выбора при компиляции или в процессе работы тех или иных свойств в соответствии с их доступностью.
Средства отладки	Поддержка более эффективной отладки с добавлением таких возможностей как самоанализ объектов, разные уровни записи в системный журнал и отладка по событиям.
Непрямое сопоставление с таблицами	Поддержка похожих на базы данных таблиц с возможностью нескольких запросов для одного СД.
Циклы в анализаторе	Поддержка в анализаторе циклов, позволяющих работать с заголовками переменного размера, списками опций, TLV и т. п.
Select+Extract в анализаторе	Расширение оператора select для совместного использования с оператором extract, позволяющего выбрать экземпляр заголовка для извлечения.

15.10 Литература

[1] Bosshart, et al. P4: Programming Protocol-Independent Packet Processors[1]. Computer Communication Review, July 2014. http://www.sigcomm.org/ccr/papers/2014/July/0000000.0000004.

[2] The P4 Language Consortium web site. http://www.p4.org.

[3] The BMv2 Simple Switch target. https://github.com/p4lang/behavioral-model/blob/master/docs/simple_switch.md.

Перевод на русский язык

Николай Малых

nmalykh@protokols.ru

[1] *Перевод статьи на русский язык доступен по ссылке. Прим. перев.*