RecordFlux Language Reference

Release

AdaCore

Apr 27, 2026

Contents

1 General

The specification language describes protocol message formats based on types. For each type of the specification language a description of its syntax and semantics and an example is given.

A simple variant of Backus-Naur Form is used to describe the syntax.

- Optional elements are enclosed in [brackets].
- Elements which occur zero or more times are enclosed in {curly brackets}.
- Terms are grouped using (parentheses).

- Alternatives are delimited by the | character.
- Reserved keywords and literals are marked in 'single quotes' or "double quotes".
- Literal ranges denote a character range from a start character to an end character separated by .. (e.g. *"a" .. "f"*).
- A production can use a non-formal verbal description starting with a # sign (e.g. *# All non-whitespace Unicode characters*).
- Non-terminals are referenced using hyperlinks (e.g. *prefix_non_terminal*).

To convey some semantic information regarding the usage of the non-terminal in a given context its name can be prefixed by an identifier when referencing it. Non-terminals with prefixes are equivalent to the respective non-terminals without the prefix.

See the section *Variable Declaration* for an example.

2 Basic Elements

The following basic elements are used to describe the syntax of the language. The type system is inspired by Ada, but differs in some details. In contrast to Ada, integer variables are considered type-compatible. Explicit type conversions of integer variables are neither required nor supported.

2.1 Comments

Comments are started with two dashes (--) and extend to the end of the line. Block comments do not exist in RecordFlux.

2.2 Names

Syntax

```
name           ::= letter { [ "_" ] alphanum { alphanum } }
alphanum       ::= letter | dec_digit
letter         ::= "A" .. "Z" | "a" .. "z"
qualified_name ::= name { "::" name }
```

By convention a name starts with a capital and after each underscore follows a capital as well. A qualified name starts with the package hierachy the respective entity is located in, separated by *::*.

Example

```
Mixed_Case_With_Underscores
```

2.3 Numbers

Syntax

```
number          ::= baseless_number
                  | bin_number
                  | oct_number
                  | dec_number
                  | hex_number
baseless_number ::= dec_digit
                    { [ "_" ] dec_digit { dec_digit } }
bin_digit       ::= "0" | "1"
oct_digit       ::= "0" .. "7"
dec_digit       ::= "0" .. "9"
hex_digit       ::= dec_digit | "A" .. "F"
```

```
bin_number      ::= "2" "#" bin_digit
                    { [ "_" ] bin_digit { bin_digit } } "#"
oct_number      ::= "8" "#" oct_digit
                    { [ "_" ] oct_digit { oct_digit } } "#"
dec_number      ::= "10" "#" dec_digit
                    { [ "_" ] dec_digit { dec_digit } } "#"
hex_number      ::= "16" "#" hex_digit
                    { [ "_" ] hex_digit { hex_digit } } "#"
```

Static Semantics

A baseless number (one not containing a "#" symbol) is considered base 10. Numbers with an explicit base (2, 8, 10, or 16) are considered to have that respective base.

Example

```
16#DEAD_C0DE#
```

2.4 Strings

Syntax

```
string    ::= '"' { character } '"'
character ::= # Any Unicode character except QUOTATION MARK
```

Static Semantics

Internally, strings are UTF-8 encoded. There is currently no way to represent the QUOTATION MARK character (U+0022).

Example

```
"Hello World"
```

2.5 Mathematical Expressions

Expressions of this category yield an integer result.

Syntax

```
math_expression       ::= ( math_expression
                            ( "+" | "-" ) math_unop_term )
                            | math_unop_term
math_unop_term        ::= ( "-" math_term ) | math_term
math_term             ::= ( math_term
                            ( "*" | "/" | "mod" )
                            math_factor )
                            | math_factor
math_factor           ::= ( math_primary "**" math_primary )
                            | math_suffix
math_suffix           ::= ( math_suffix "'" math_attribute )
                            | math_primary
math_attribute        ::= "First" | "Size" | "Last"
math_primary          ::= number
                            | qualified_name
                            | selected
                            | paren_math_expression
```

```
paren_math_expression ::= "(" math_expression ")"
```

Example

```
(2 ** (V'First + 23) - 12) + 1
```

2.6 Sequence Expressions

Syntax

```
seq_expression       ::= seq_paren_expression
                       | comprehension
                       | aggregate
                       | selected
                       | string
                       | qualified_name
seq_paren_expression ::= "(" seq_expression ")"
```

2.7 Boolean Expressions

Expressions of this category yield a boolean result. Standard boolean expressions are used in then-clauses and refinements of messages.

Syntax

```
bool_expression       ::= ( bool_expression
                            ( "and" | "or" ) bool_unop_term )
                          | bool_unop_term
bool_unop_term        ::= ( "not" bool_term ) | bool_term
bool_term             ::= "True"
                          | "False"
                          | bool_suffix
                          | qualified_name
                          | bool_relation
                          | math_relation
                          | seq_relation
                          | bool_paren_expression
bool_suffix           ::= name "'" bool_attribute
bool_attribute        ::= "Valid_Checksum"
bool_relation         ::= bool_expression
                            ( "=" | "/=" ) bool_expression
math_relation         ::= math_expression
                            ( "=" | "/=" | "<=" | "<" | ">=" | ">" )
                            math_expression
seq_relation          ::= seq_expression
                            ( "=" | "/=" ) seq_expression
seq_membership        ::= expression
                            ( "in" | "not in" ) seq_expression
bool_paren_expression ::= "(" bool_expression ")"
```

Static Semantics

The semantics of the `Valid_Checksum` attribute is explained in the section *Message Types*.

Example

```
V < 20 and U = True
```

2.8 Extended Boolean Expressions

Expressions of this category yield a boolean result. Extended boolean expressions are used in state machines.

Syntax

```
ext_bool_expression       ::= ( ext_bool_expression
                              ( "and" | "or" )
                              ext_bool_unop_term )
                              | ext_bool_unop_term
ext_bool_unop_term        ::= ( "not" ext_bool_term )
                              | ext_bool_term
ext_bool_term             ::= "True"
                              | "False"
                              | ext_bool_suffix
                              | qualified_name
                              | ext_bool_relation
                              | math_relation
                              | seq_relation
                              | seq_membership
                              | quantified_expression
                              | selected
                              | ext_bool_paren_expression
ext_bool_suffix           ::= expression "'" ext_bool_attribute
ext_bool_attribute        ::= "Valid" | "Has_Data" | "Present"
ext_bool_relation         ::= ext_bool_expression
                              ( "=" | "/=" )
                              ext_bool_expression
ext_bool_paren_expression ::= "(" ext_bool_expression ")"
```

Static Semantics

The Valid attribute allows to determine the validity of a message or sequence.

Whether a channel contains data can be checked with the Has_Data attribute.

2.9 Sizes

A size aspect defines the size of a type or message field in bits.

Syntax

```
size_aspect     ::= "Size" "=>" size_expression
size_expression ::= math_expression
```

3 Scalar Types

3.1 Integer Types

An integer type is used to represent whole numbers. In RecordFlux integer types can be specified in several ways as explained next.

Syntax

```
integer_type ::= range_type
               | unsigned_type
```

Range Integer Types

A range integer type is the most general form of integer type specifications. It allows one to explicitly specify the lower and upper bound for the type, as well as the size of the type in bits. At the moment only non-negative integers are supported.

Syntax

```
range_type ::= "type" name "is"
               "range" first ".." last
               "with" size_aspect
first      ::= math_expression
last       ::= math_expression
```

Static Semantics

The set of values of a range integer type consists of all numbers from the lower bound to the upper bound. The lower bound must be >= 0 and <= the upper bound. The bit size has to be specified explicitly and must be a value between 1 and 63 (inclusive). It does not have to be a multiple of 8 bits. However, the size of the type must be able to accommodate the upper bound.

Example

```
type Type_Length is range 46 .. 2 ** 16 - 1 with Size => 16
```

Unsigned Integer Types

For unsigned integers which cover the whole range of their specified size the following shorthand syntax is available.

Syntax

```
unsigned_type ::= "type" name
                  "is" "unsigned" size_expression
```

Static Semantics

The above syntax is equivalent to the definition of a range integer type where the lower limit is `0` and upper limit `2 ** size - 1`.

Example

```
-- Value range: 0 .. 63
type Address is unsigned 6
```

3.2 Enumeration Types

An enumeration type represents a value out of a list of possible values.

Syntax

```
enumeration_type    ::= "type" name "is" "(" literals ")"
                        "with" enumeration_aspects
literals            ::= enumeration_literal
                        { "," enumeration_literal }
enumeration_literal ::= name [ "=>" number ]
enumeration_aspects ::= enumeration_aspect
```

```
                          { "," enumeration_aspect }
  enumeration_aspect  ::= size_aspect | always_valid_aspect
  always_valid_aspect ::= "Always_Valid"
                          [ "=>" ( "True" | "False" ) ]
```

Static Semantics

The set of values of an enumeration type consists of the list of declared enumeration literals. Each enumeration literal has a distinct value. If no explicit value is given, the first literal is zero, and the value of each subsequent literal is incremented by one. Literals with and without explicit value must not be intermixed in one definition. The bit size of the enumeration type must be specified explicitly. Optionally, an enumeration type can be flagged as always valid. A message field with such type is always considered valid, whether or not its value corresponds to one of the specified literals.

Example

```
type Tag is (Msg_Error, Msg_Data) with Size => 1
```

```
type Ether_Type is
   (ET_IPv4            => 16#0800#,
    ET_ARP             => 16#0806#,
    ET_VLAN_Tag        => 16#8100#,
    ET_IPv6            => 16#86DD#,
    ET_VLAN_Tag_Double => 16#9100#)
with Size => 16, Always_Valid
```

3.3 Boolean

`Boolean` is a built-in enumeration type with the literals `False => 0` and `True => 1` with a size of 1 bit.

4 Parameters and Arguments

Parameters define the objects directly visible within functions or parameterized messages and their associated types.

```
  parameter      ::= name ":" qualified_name
  parameter_list ::= "(" parameter { ";" parameter } ")"
```

Named arguments associate a parameter with an expression.

```
  named_argument      ::= parameter_name "=>" expression
  named_argument_list ::= named_argument
                          { "," named_argument }
```

5 Message Types

A message type is a collection of fields. Additional *then clauses* enable the definition of conditions and dependencies between fields.

Syntax

```
  message_type         ::= "type" name [ parameter_list ] "is"
                           ( "message"
                           [ null_field ]
                           field
                           { field }
```

```
                        "end" "message" [ "with"
                        message_aspects ]
                        | "null" "message" )
field                ::= name ":" qualified_name
                        [ "(" named_argument_list ")" ]
                        [ "with" aspects ]
                        { then_clause } ";"
null_field           ::= "null" then_clause { then_clause } ";"
target_field         ::= field_name | "null"
then_clause          ::= "then" target_field
                        [ "with" aspects ]
                        [ "if" bool_expression ]
aspects              ::= aspect { "," aspect }
aspect               ::= first_aspect | size_aspect
first_aspect         ::= "First" "=>" math_expression
message_aspects      ::= message_aspect { "," message_aspect }
message_aspect       ::= checksum_aspect | byteorder_aspect
checksum_aspect      ::= "Checksum" "=>"
                        "(" checksum_definition
                        { "," checksum_definition } ")"
checksum_definition  ::= name "=>"
                        "(" checksum_element
                        { "," checksum_element } ")"
checksum_element     ::= name
                        | name "'" "Size"
                        | field_range
field_range          ::= field_range_first ".." field_range_last
field_range_first    ::= name "'" "First"
                        | name "'" "Last" "+" "1"
field_range_last     ::= name "'" "Last"
                        | name "'" "First" "-" "1"
byteorder_aspect     ::= "Byte_Order" "=>" byteorder_definition
byteorder_definition ::= "High_Order_First"
                        | "Low_Order_First"
```

Static Semantics

A message type specifies the message format of a protocol. A message is represented by a graph-based model. Each node in the graph corresponds to one field in a message. The links in the graph define the order of the fields. A link is represented by a then clause in the specification. If no then clause is given, it is assumed that always the next field of the message follows. If no further field follows, it is assumed that the message ends with this field. The end of a message can also be denoted explicitly by adding a then clause to *null*. Optionally, a then clause can contain a condition under which the corresponding field follows and aspects which enable the definition of the size of the next field and the location of its first bit. These aspects can also be specified for the field directly. Each aspect can be specified either for the field or in all incoming then clauses, but not in both. The condition can refer to previous fields (including the field containing the then clause). If required, a null field can be used to specify the size of the first field in the message. An empty message can be represented by a null message.

A message can be parameterized. Message parameters can be used in conditions and aspects and enable the definition of message formats that depend on prior negotiation. Only scalar types are allowed for parameters.

The field type `Opaque` represents an unconstrained sequence of bytes. The size of opaque fields and sequence fields must be defined by a size aspect, if another field can follow. If no size aspect is given, the field size is implicitly defined by the available space (defined by the outer message when parsing or by the written data when serializing). Opaque fields and sequence fields must be byte aligned. The size of a message must be a multiple of 8 bit.

A checksum aspect specifies which parts of a message are covered by a checksum. The definition of the checksum calculation is not part of the specification. Code based on the message specification must provide a function which is able to verify a checksum using the specified checksum elements. A checksum element can be a field value, a field size or a range of fields. The point where a checksum should be checked during parsing or generated during serialization must be defined for each checksum. For this purpose the `Valid_Checksum` attribute is added to a condition. All message parts on which the checksum depends have to be known at this point.

The `Byte_Order` aspect allows the user to specify the endianness of the message, with the two possible choices `High_Order_First` (big endian, or network byte order) and `Low_Order_First` (little endian). If the `Byte_Order` aspect is not specified, the byte order of the message is set to `High_Order_First`.

`Message'First`, `Message'Last` and `Message'Size` can be used in expressions to refer to the position of the first or last bit of the message or the size of the message. All bytes which were received when parsing or were written when serializing are considered as part of the message.

Example

```
type Frame is
   message
      Destination : Address;
      Source : Address;
      Type_Length_TPID : Type_Length
         then TPID
            with First => Type_Length_TPID'First
            if Type_Length_TPID = 16#8100#
         then Payload
            with Size => Type_Length_TPID * 8
            if Type_Length_TPID <= 1500
         then Ether_Type
            with First => Type_Length_TPID'First
            if Type_Length_TPID >= 1536 and Type_Length_TPID /= 16#8100#;
      TPID : TPID;
      TCI : TCI;
      Ether_Type : Ether_Type;
      Payload : Opaque
         then null
            if Payload'Size / 8 >= 46 and Payload'Size / 8 <= 1500;
   end message
```

```
type Empty_Message is null message
```

6 Type Refinements

A type refinement describes the relation of an opaque field in a message type to another message type.

Syntax

```
type_refinement ::= "for" refined_qualified_name "use"
                    "("
                    refined_field_name
                    "=>" message_qualified_name
                    ")"
                    [ "if" bool_expression ]
```

Static Semantics

A type refinement describes under which condition a specific message can be expected inside of a payload field. Only fields of type `Opaque` can be refined. Types defined in other packages are referenced by a qualified name in the form `Package_Name::Message_Type_Name`. The condition can refer to fields of the refined type. To indicate that a refined field is empty (i.e. does not exist) under a certain condition, a null message can be used as message type.

Example

```
for Ethernet::Frame use (Payload => IPv4::Packet)
   if Ether_Type = Ethernet::IPV4
```

7 Type Derivations

A type derivation enables the creation of a new message type based on an existing message type.

Syntax

```
type_derivation ::= "type" name "is"
                    "new" base_type_qualified_name
```

Static Semantics

A derived message type derives its specification from a base type. Type refinements of a base message type are not inherited by the derived message type.

Example

```
type Specific_Extension is new Extension
```

8 Sequence Types

A sequence type represents a list of similar elements.

Syntax

```
sequence_type ::= "type" name "is" "sequence"
                  "of" element_qualified_name
```

Static Semantics

A sequence consists of a number of elements with the same type. Scalar types as well as message types can be used as element type.

Example

```
type Options is sequence of Option
```

9 State Machines

A state machine defines the dynamic behavior of a protocol using a finite state machine. The first defined state is considered the initial state. The external interface of a state machine is defined by parameters. The declaration part enables the declaration of state machine global variables. The main part of a state machine definition is the state definitions.

Syntax

```
state_machine ::= "generic"
```

```
                { state_machine_parameter }
                "machine" name "is"
                { state_machine_declaration }
                "begin"
                state
                { state }
                "end" name
```

Example

```
generic
   X : Channel with Readable, Writable;
   with function F return T;
   with function G (P : T) return Boolean;
machine S is
   Y : Boolean := False;
begin
   state A
      with Desc => "rfc1149.txt+51:4-52:9"
   is
      Z : Boolean := Y;
      M : TLV::Message;
   begin
      X'Read (M);
   transition
      goto null
         with Desc => "rfc1149.txt+45:4-47:8"
         if Z = True
            and G (F) = True
      goto A
   end A;
end S
```

9.1 State Machine Parameters

Functions and channels can be defined as state machine parameters.

Syntax

```
state_machine_parameter ::= ( function_declaration
                            | channel_declaration
                            ) ";"
```

Functions

Functions enable the execution of externally defined code.

Syntax

```
function_declaration ::= "with" "function" name
                         [ parameter_list ]
                         "return" type_qualified_name
```

Static Semantics

Allowed parameter types:

- Scalars
- Definite messages
- Opaque fields of messages

Allowed return types:

- Scalars
- Definite messages

Definite messages are messages with no optional fields and an explicit size (i.e. all size aspects contain no reference to `Message`).

SPARK

For each function declaration in the state machine specification a procedure declaration is added to the corresponding state machine package. The return type and parameters of a function are represented by the first and subsequent parameters of the generated procedure declaration.

Example

```
with function Decrypt
   (Key_Update_Message : Key_Update_Message;
    Sequence_Number    : Sequence_Number;
    Encrypted_Record   : Opaque)
return
   TLS_Inner_Plaintext
```

Channels

Channels provide a way for communicating with other systems using messages.

Syntax

```
channel_declaration ::= name ":" "Channel" "with"
                        channel_aspect
                        { "," channel_aspect }
channel_aspect      ::= "Readable" | "Writable"
```

Static Semantics

Channels can be readable or writable (non-exclusive).

Example

```
Data_Channel : Channel with Readable, Writable
```

9.2 Declarations

Variables and renamings can be globally declared (i.e. for the scope of the complete state machine).

Syntax

```
state_machine_declaration ::= ( variable_declaration
                              | renaming_declaration
                              ) ";"
```

Variable Declaration

A declared variable must have a type and can be optionally initialized using an expression.

Syntax

```
variable_declaration ::= variable_name
                         ":" type_qualified_name
                         [ ":=" initialization_expression ]
```

Example

```
Error_Sent : Boolean := False
```

Renaming Declaration

Syntax

```
renaming_declaration ::= name ":" message_qualified_name
                         "renames"
                         message_variable_name
                         "." field_name
```

Example

```
Client_Hello_Message : TLS_Handshake::Client_Hello renames Client_Hello_Handshake_
↪Message.Payload
```

9.3 States

A state defines the to be executed actions and the transitions to subsequent states.

Variable declarations and renaming declarations in a state have a state-local scope, i.e., local declarations cannot be accessed from other states.

Syntax

```
state              ::= "state" name
                       [ "with" description_aspect ]
                       "is"
                       { local_state_machine_declaration }
                       "begin"
                       { state_action }
                       "transition"
                       { conditional_transition }
                       transition
                       [ "exception"
                       transition ]
                       "end" name ";"
description_aspect ::= "Desc" "=>" string
```

Static Semantics

An exception transition must be defined just in case any action might lead to a critical (potentially non-recoverable) error:

- Insufficient memory for setting a field of a message
- Insufficient memory for appending an element to a sequence or extending a sequence by another sequence

Exception transitions are currently also used for other cases.

A local declaration must not hide a global declaration.

The states where the actions include either `Read` or `Write` operations are referred to as **IO states**. See the sections *Read Attribute Statements* and *Write Attribute Statements* for more information about those operations. IO states cannot contain any declarations and they must not contain any other operations than channel IO. Each channel and each message can be read or written at most once in a given IO state.

Dynamic Semantics

After entering a state the declarations and actions of the state are executed. If a non-recoverable error occurs, the execution is aborted and the state is changed based on the exception transition. When all action were executed successfully, the conditions of the transitions are checked in the given order. If a condition is fulfilled, the corresponding transition is taken to change the state. If no condition could be fulfilled or no conditional transitions were defined, the default transition is used.

Example

```
state A
   with Desc => "rfc1149.txt+51:4-52:9"
is
   Z : Boolean := Y;
   M : TLV::Message;
begin
   X'Read (M);
transition
   goto B
      with Desc => "rfc1149.txt+45:4-47:8"
      if Z = True and G (F) = True
   goto A
end A
```

State Transitions

State transitions define the conditions for the change to subsequent states. An arbitrary number of conditional transitions can be defined. The last transition in a state definition is the default transition, which does not contain any condition. The transition target must be either a state name or *null*, which represents the final state.

Syntax

```
conditional_transition ::= transition
                           "if"
                           conditional_ext_bool_expression
transition             ::= "goto" state_name
                           [ "with" description_aspect ]
```

Example

```
goto B
   with Desc => "rfc1149.txt+45:4-47:8"
   if Z = True and G (F) = True
```

State Actions

The state actions are executed after entering a state.

Syntax

```
state_action ::= ( message_field_assignment
                 | assignment
                 | append
                 | extend
                 | reset
                 | read
                 | write
                 ) ";"
```

Assignment Statements

An assignment sets the value of variable.

Syntax

```
assignment ::= variable_name ":=" expression
```

Dynamic Semantics

An assignment always creates a copy of the original object.

Example

```
Error_Sent := True
```

Message Field Assignment Statements

A message field assignment sets the value of a message field.

Syntax

```
message_field_assignment ::= variable_name "." field_name
                             ":=" expression
```

Dynamic Semantics

Message fields must be set in order. Trying to set a message field which is not a valid next field leads to an exception transition. All subsequent fields of the set message field are invalidated.

Example

```
Packet.Length := 42
```

Append Attribute Statements

An element is added to the end of a sequence using the Append attribute.

Syntax

```
append ::= sequence_name "'" "Append" "(" expression ")"
```

Dynamic Semantics

Appending an element to a sequence might lead to an exception transition.

Example

```
Parameter_Request_List'Append (DHCP::Domain_Name_Option)
```

Extend Attribute Statements

The Extend attributes adds a sequence of elements to the end of a sequence.

Syntax

```
extend ::= sequence_name "'" "Extend" "(" expression ")"
```

Dynamic Semantics

Extending a sequence might lead to an exception transition.

Example

```
Parameter_Request_List'Extend (Parameters)
```

Reset Attribute Statements

The state of a message or sequence can be cleared using the Reset attribute.

Syntax

```
reset ::= name "'" "Reset" [ "(" named_argument_list ")" ]
```

Static Semantics

When resetting a parameterized message, the intended values for the parameters of the message must be defined.

Dynamic Semantics

The existing state of a message or sequence is removed (and the corresponding buffer is cleared).

Example

```
Message'Reset
```

Read Attribute Statements

The read attribute statement is used to retrieve a message from a channel.

Syntax

```
read ::= channel_name "'" "Read" "(" expression ")"
```

Example

```
Data_Channel'Read (Message)
```

Write Attribute Statements

A message can be sent through a channel using a write attribute statement.

Syntax

```
write ::= channel_name "'" "Write" "(" expression ")"
```

Dynamic Semantics

Writing an invalid message leads to an exception transition.

Example

```
Data_Channel'Write (Message)
```

9.4 Expressions

Syntax

```
expression ::= math_expression
             | ext_bool_expression
             | message_aggregate
             | attribute_reference
             | selected
             | seq_expression
             | comprehension
             | conversion
             | call
             | case_expression
             | name
```

Message Aggregates

Syntax

```
message_aggregate                  ::= message_qualified_name "'"
                                       "("
                                       message_aggregate_association_list
                                       ")"
message_aggregate_association_list ::= named_argument_list
                                       | "null" "message"
```

Dynamic Semantics

An invalid condition during message creation leads to an exception transition. Insufficient memory during the message creation leads to an exception transition.

Example

```
TLS_Record::TLS_Record'(Tag                   => TLS_Record::Alert,
                        Legacy_Record_Version => TLS_Record::TLS_1_2,
                        Length                => Alert_Message'Size / 8,
                        Fragment              => Alert_Message'Opaque)
```

```
Null_Message'(null message)
```

Aggregates

An aggregate is a collection of elements.

Syntax

```
aggregate ::= "[" [ number { "," number } ] "]"
```

Example

```
[0, 1, 2]
```

```
[]
```

Attribute Expressions

Syntax

```
attribute_reference  ::= expression
                         "'" attribute_designator
attribute_designator ::= "Opaque" | "Head"
```

Static Semantics

The byte representation of a message can be retrieved using the Opaque attribute.

The Head attribute returns the first element of a sequence.

Dynamic Semantics

The use of the Opaque attribute on an invalid message or the use of the Head attribute on an empty sequence leads to an exception transition.

Example

```
Message'Opaque
```

Selected Expressions

The Selected expression is used to get a value of a message field.

Syntax

```
selected ::= message_expression "." field_name
```

Dynamic Semantics

Accesses to message fields that were detected as invalid during parsing lead to an exception transition.

Example

```
Ethernet_Frame.Payload
```

List Comprehensions

A list comprehension provides a way to create a new sequence based on an existing sequence.

Syntax

```
comprehension ::= "["
                  "for" name "in" iterable_expression
                  [ "if" condition_ext_bool_expression ]
                  "=>" selector_expression
                  "]"
```

Dynamic Semantics

An access to an invalid element in iterable *expression* leads to an exception transition.

Example

```
[for O in Offer.Options if O.Code = DHCP::DHCP_Message_Type_Option => O.DHCP_Message_
↪Type]
```

Quantified Expressions

Quantified expressions enable reasoning about properties of sequences.

Syntax

```
quantified_expression ::= "for" quantifier name
                          "in" iterable_expression
                          "=>"
                          predicate_ext_bool_expression
quantifier            ::= "all" | "some"
```

Example

```
for all E in Server_Hello_Message.Extensions => E.Tag /= TLS_Handshake::ET_Supported_
↪Versions
```

Calls

All functions which are declared in the state machine parameters can be called.

Syntax

```
call ::= qualified_name
         [ "(" argument_expression
         { "," argument_expression } ")" ]
```

Example

```
Decrypt (Key_Update_Message, Sequence_Number, TLS_Record_Message.Encrypted_Record)
```

Conversions

An opaque field of a message can be converted to a message.

Syntax

```
conversion ::= message_qualified_name
               "(" message_expression "." field_name ")"
```

Static Semantics

A conversion is only allowed if a refinement for the message field and the intended target type exists.

Dynamic Semantics

An invalid condition of a refinement leads to an exception transition.

Example

```
Key_Update_Message (Handshake_Control_Message.Data)
```

Case Expressions

A *case expression* selects one of several alternative dependent *expressions* for evaluation based on the value of a selecting *expression*.

Syntax

```
case_expression             ::= "(" "case" selecting_expression
                                "is" case_expression_alternative { ","
                                case_expression_alternative } ")"
case_expression_alternative ::= "when" discrete_choice_list
                                "=>" dependent_expression
discrete_choice_list        ::= discrete_choice
                                { "|" discrete_choice }
discrete_choice             ::= number | qualified_name
```

Static Semantics

The type of all the dependent *expression*s shall be compatible to the type of the *case expression*. Each value of the type of the selecting *expression* shall be covered by a *discrete choice*. Two distinct *discrete choices* of a *case expression* shall not cover the same value.

Example

```
(case Value is
    when T::V1 | T::V2 => 2,
    when T::V3         => 4)
```

10 Packages

A package is used to structure a specification.

Syntax

```
package           ::= "package" name "is"
                      { basic_declaration }
                      "end" name ";"
basic_declaration ::= ( integer_type
                      | enumeration_type
                      | message_type
                      | type_refinement
                      | type_derivation
                      | sequence_type
                      | state_machine ) ";"
```

Static Semantics

A package is a collection of types and state machines. By convention one protocol is specified in one package.

Example

```
package Ethernet is

   type Address is unsigned 48;
   type Type_Length is range 46 .. 2 ** 16 - 1 with Size => 16;
   type TPID is range 16#8100# .. 16#8100# with Size => 16;
   type TCI is unsigned 16;
```

(continues on next page)

(continued from previous page)

```
type Ether_Type is
   (ET_IPv4            => 16#0800#,
    ET_ARP             => 16#0806#,
    ET_VLAN_Tag        => 16#8100#,
    ET_IPv6            => 16#86DD#,
    ET_VLAN_Tag_Double => 16#9100#)
with Size => 16, Always_Valid;

type Frame is
   message
      Destination : Address;
      Source : Address;
      Type_Length_TPID : Type_Length
         then TPID
            with First => Type_Length_TPID'First
            if Type_Length_TPID = 16#8100#
         then Payload
            with Size => Type_Length_TPID * 8
            if Type_Length_TPID <= 1500
         then Ether_Type
            with First => Type_Length_TPID'First
            if Type_Length_TPID >= 1536 and Type_Length_TPID /= 16#8100#;
      TPID : TPID;
      TCI : TCI;
      Ether_Type : Ether_Type;
      Payload : Opaque
         then null
            if Payload'Size / 8 >= 46 and Payload'Size / 8 <= 1500;
   end message;

generic
   Input : Channel with Readable;
   Output : Channel with Writable;
machine Validator is
   Frame : Ethernet::Frame;
begin
   state Validate
   is
   begin
      Input'Read (Frame);
   transition
      goto Forward
         if Frame'Valid
      goto Validate
   exception
      goto null
   end Validate;

   state Forward
   is
   begin
      Output'Write (Frame);
```

(continues on next page)

(continued from previous page)

```
      transition
         goto Validate
      end Forward;
   end Validator;

end Ethernet;
```

11 Context Clauses

The context clause is used to specify the relation to other packages and consists of a list of with clauses.

Syntax

```
context ::= { "with" package_name ";" }
```

Static Semantics

For each package referenced in a file, a corresponding with clause has to be added to the beginning of the file.

Example

```
with Ethernet;
with IPv4;
```

12 Files

A RecordFlux specification file is recognized by the file extension `.rflx`. Each specification file contains exactly one package. The file name must match the package name in lower case characters.

Syntax

```
file ::= context package
```

Example

File: `in_ethernet.rflx`.

```
with Ethernet;
with IPv4;

package In_Ethernet is

   for Ethernet::Frame use (Payload => IPv4::Packet)
      if Ether_Type = Ethernet::ET_IPv4;

end In_Ethernet;
```