

30. Reflection API

This clause explains how to access structural information about the type system through the application programming interface (API), also known as the Reflection API, and defines the Reflection API itself. See also: [Annex E](#) (for examples).

30.1 Introduction

Reflection (sometimes called introspection) is a programmatic interface into the meta-data of a program. Most object-oriented (OO) languages and systems supply some way of referring to meta-level entities—mainly type-related, such as classes, methods, and fields. The richness of modeling concepts supported by the *e* language calls for a much more comprehensive reflection facility than that of other languages. Reflection interfaces are typically used for constructing external developer aid tools such as class browsers, data browsers, debugging aids, source browsers, etc. They can also be used to implement generic utilities, where the modeling powers of the language such as inheritance and polymorphism are not strong enough, e.g., object serialization utilities like packing or register filling. The reflection interface in *e* is designed to facilitate such third-party applications and tools, where these problems typically arise.

30.1.1 Representation

The *e* reflection facility is a class API. Each structural element in the program is represented by an object (an *e* struct instance). One object represents, for example, the type **int**, another represents the struct type **any_struct**, another represents the method **to_string()**, and so on. These representations are called *meta-objects*. Meta-objects are classified into different groups which are called *meta-types*. These *e* struct types form a hierarchy of abstractions and show the different relations between such entities. All of these meta-types have a common prefix to their name, `rf_`.

30.1.2 Structure

The interface is divided into three main parts: *type information*, *aspect information*, and *value query and manipulation*, which correspond to subclauses [30.2](#), [30.3](#), and [30.4](#) respectively. This grouping of the API's functionality cuts across meta-types so that the interface of one struct may consist of methods that are defined in different parts (e.g., some methods of the struct **rf_event** are described in [30.2](#), others in [30.3](#), and the rest in [30.4](#)).

Each meta-type is introduced separately. Its location in the type hierarchy is denoted by showing its *like* inheritance (if any), which has the usual inheritance implications (e.g., the method **is_private()**, which is defined for the struct **rf_struct_member**, also exists for **rf_field**, which is *like* **rf_struct_member**). The concept behind each meta-type is explained, its methods are detailed, and each method's return type is set off by a colon (:). For example, in **rf_named_entity.get_name()**: string, the type returned is a string.

30.1.3 Terminology and conventions

- There is a possibility of confusion when dealing with meta-data and meta-types, as objects are used to represent types, methods, and so on; while they themselves, like any other object, are instances of types and have methods. However, for the sake of readability, and where there is no ambiguity, the phrase “the type/method/field” is shorthand for “an object representing the type/method/field.” For example, the method **rf_struct.get_methods()** returns a list of methods of this struct, which means it returns a list of objects representing this struct's methods.

Clarifying the concept of **like** and **when** inheritance (see [6.1](#)) is a major concern. Therefore, two trivial examples are used in many places to illustrate these definitions. One is the hierarchy of *dog*, with *bulldog* and *poodle* as its **like** heirs. The other is the struct `packet`, having an enumerated field `size`, and a

1

Issue
732Issue
5734Issue
746

0

15

Issue
733

20

25

30

Issue
746Issue
734

40

45

50

55

1 Boolean field `corrupt`, which allow for **when** subtypes such as `small packet` or `big corrupt packet`.

5 30.2 Type information

The core of the reflection API is the representation of types in *e*. This part of the interface enables all type-related queries concerning scalar types, list types, struct types, methods, fields, and events. From the viewpoint of the reflection API, units are simply structs (see [30.2.2](#) for how to query if a struct is a unit).

10 30.2.1 Named entities

This defines the types of named entities.

15 30.2.1.1 `rf_named_entity`

Named entities are types and struct members. Both kinds are entities that, once declared, become part of the lexicon of the language. All named entities have a name (string) and are either visible or hidden. The importance of this abstraction is related to [30.3](#).

- 20 a) **`rf_named_entity.get_name()`**: string
Returns the name of this entity.
- 25 b) **`rf_named_entity.is_visible()`**: bool
Returns TRUE if this entity is visible. Otherwise, returns FALSE. *Invisible (hidden) named entities* include fields and methods of any user-defined structs, which are not shown in printing and visualization tools.

30 30.2.1.2 `rf_type`

This method has *like* inheritance from `rf_named_entity` (see [30.2.1.1](#)).

- 35 a) **`rf_type.is_public()`**: bool
Returns TRUE if this type has unrestricted access. Otherwise, returns FALSE (when this type was declared with a *package* modifier).
- 40 b) **`rf_type.get_qualified_name()`**: string
Returns the fully qualified name of the type (i.e., with the declaring package name followed by the `::` operator).

40 30.2.2 Struct types: `rf_struct`

This method has *like* inheritance from `rf_type` (see [30.2.1.2](#)). See also: [Clause 6](#).

- 45 a) **`rf_struct.get_fields()`**: list of `rf_field`
Returns a list containing all fields of this struct (declared by it or inherited from its parent type).
- b) **`rf_struct.get_declared_fields()`**: list of `rf_field`
Returns a list containing all fields declared in the context of this struct (a subset of `rf_struct.get_fields()`).
- 50 c) **`rf_struct.get_field(name: string)`**: `rf_field`
Returns the field of this struct with the *name* or NULL if no such field exists. Field names are unique in the context of a struct.
- 55 d) **`rf_struct.get_methods()`**: list of `rf_method`
Returns a list containing all methods of this struct (declared by it or inherited from its parent type).

- e) **rf_struct.get_declared_methods()**: list of rf_method 1
Returns a list containing all methods declared in the context of this struct (a subset of **rf_struct.get_methods()**). Methods that are declared by a parent type and extended or overridden in the context of this struct are not returned (see also: [6.3](#)). 5
- f) **rf_struct.get_method(name: string)**: rf_method 5
Returns the method of this struct with the *name* or NULL if no such method exists. Method names are unique in the context of a struct.
- g) **rf_struct.get_events()**: list of rf_event 10
Returns a list of all events of this struct (declared by it or inherited from its parent type).
- h) **rf_struct.get_declared_events()**: list of rf_event 15
Returns a list of all events declared in the context of this struct (a subset of **rf_struct.get_events()**). Events that are declared by a parent type and overridden in the context of this struct are not returned (see also: [6.3](#)).
- i) **rf_struct.get_event(name: string)**: rf_event 20
Returns the event of this struct with the *name* or NULL if no such event exists. Event names are unique in the context of a struct.
- j) **rf_struct.is_unit()**: bool 20
Returns TRUE if this struct is a unit. Otherwise, returns FALSE. Returning TRUE is the only indication this meta-object represents a unit rather than a regular struct type.

30.2.3 Struct members 25

Struct members are represented by instances of the meta-types **rf_field** (see [30.2.3.2](#)), **rf_method** (see [30.2.3.3](#)), and **rf_event** (see [30.2.3.5](#)). Each member is introduced for the first time in the context of some struct—its *declaring struct*. Its access rights: one of **package**, **protected**, **private**, or **public** (the default) are assigned to it upon its declaration. 30

30.2.3.1 rf_struct_member

- This method has *like* inheritance from **rf_named_entity** (see [30.2.1.1](#)). 35
- a) **rf_struct_member.get_declaring_struct()**: rf_struct 35
Returns the struct where this member was introduced. This applies also to the empty definition of methods or declarations of undefined methods.
- b) **rf_struct_member.applies_to(rf_struct)**: bool 40
Returns TRUE if this struct member applies to instances of *rf_struct*; i.e., this was declared by *rf_struct* or by a different struct that contains *rf_struct* (see also: [6.2](#)). Otherwise, returns FALSE.
- c) **rf_struct_member.is_private()**: bool 45
Returns TRUE if this struct member was declared with the **private** access modifier; i.e., it is accessible only within the context of both its package and its declaring struct or its subtypes. Otherwise, returns FALSE.
- d) **rf_struct_member.is_protected()**: bool 50
Returns TRUE if this struct member was declared with **protected** access modifier; i.e., it is accessible only within the context of the declaring struct or its subtypes. Otherwise, returns FALSE.
- e) **rf_struct_member.is_package_private()**: bool 55
Returns TRUE if this struct member was declared with **package** access modifier; i.e., it is accessible only within the context of the package where it was declared. Otherwise, returns FALSE.

- 1 f) **rf_struct_member.is_public()**: bool

Returns TRUE if this struct member was declared without an access modifier; i.e., its access is not restricted. Otherwise, returns FALSE.

5 **30.2.3.2 rf_field**

This method has *like* inheritance from **rf_struct_member** (see [30.2.3.1](#)).

- 10 a) **rf_field.get_type()**: rf_type

Returns the declared type of the field.

- b) **rf_field.is_physical()**: bool

Returns TRUE if the field is declared physical (i.e., with the % modifier (see [6.8](#))). Otherwise, returns FALSE. Physical fields are those that are packed when the struct is packed.

- 15 c) **rf_field.is_ungenerated()**: bool

Returns TRUE if the field is declared as ungenerated (i.e., with the ! modifier (see [6.8](#))). Otherwise, returns FALSE. Ungenerated fields are not generated automatically when the struct is generated.

- 20 d) **rf_field.is_unit_instance()**: bool

Returns TRUE if the field is an instance of a unit (i.e., declared as **is instance** (see [7.2.2](#))). Otherwise, returns FALSE.

25 **30.2.3.3 rf_method**

This method has *like* inheritance from **rf_struct_member** (see [30.2.3.1](#)).

- 30 a) **rf_method.get_result_type()**: rf_type

Returns the object that represents the result type of this method or NULL if the method does not return any value.

- b) **rf_method.get_parameters()**: list of rf_parameter

Returns a list of formal parameters of this method. If the method has no parameters, the list is empty.

- 35 c) **rf_method.is_tcm()**: bool

Returns TRUE if this method may consume time; i.e., it is declared as a TCM. Otherwise, returns FALSE.

- d) **rf_method.is_inline()**: bool

Returns TRUE if the method is declared as *inline* (see [17.1.1](#)). Otherwise, returns FALSE.

- 40 e) **rf_method.get_sampling_event()**: rf_event

Returns the object that represents the default sampling event in case this method is a TCM or NULL otherwise.

45 **30.2.3.4 rf_parameter**

- a) **rf_parameter.get_name()**: string

Returns the name given to this parameter in the declaration method.

- 50 b) **rf_parameter.get_type()**: rf_type

Returns the type of this parameter.

- c) **rf_parameter.is_by_reference()**: bool

Returns TRUE if this parameter is passed by reference; i.e., it was declared using * (see [17.3.1](#)). Otherwise, returns FALSE.

30.2.3.5 rf_event 1

This method has *like* inheritance from **rf_struct_member** (see [30.2.3.1](#)).

30.2.4 Inheritance and when subtypes 5

****During the Draft 3 Review, compare all summary material on this page against Clause 6, Structs**** 10

There are two mechanisms for subtyping in *e*. One is OO single inheritance, where a struct is declared as deriving from another—*like inheritance*. The other is closer to predicate classes, where a behavioral or structural feature of an object is determined by some state or attribute of it—*when subtyping*. In both cases, a new struct type is defined in terms of an existing one. But, the relations between the two kinds of struct types are different and they are represented by different kinds of meta-objects. Thus, there are two kinds of struct types: **like** structs and **when** subtypes. 15

Issue 735

The two mechanisms, **like** and **when**, do not mix. *Like* inheritance lays the basic type hierarchy. Only the leaves of the hierarchy tree, the structs that have no *like* subtypes, can serve as a base for *when* proliferations. Each **when** variant is a different type, but unlike **like** structs, these types are not derived from each other and do not form a hierarchy. To mark this difference, the set of **when** subtypes is called a *when family* and the **like** struct that serves as the base for these proliferations is called a *when base*. Also, a **when** subtype can be determined by a field that is declared in the context of another **when** subtype; however, such subtypes are part of the same *when family* with the same *when base*. 20

Issue 746

See also: [Clause 6](#).

30.2.4.1 Canonical names 30

Each value of an enumerated or a Boolean field of an object can serve as a determinant of its structure and behavior. A set of one or more field-values pairs (determinations) corresponds to a potential **when** subtype. One such type can have a number of names by which it is identified—using fully qualified determinants or without them (e.g., `big' size packet` versus `big packet`)—and in a different determinant order (e.g., `big corrupt packet` versus `corrupt big packet`). However, the method `get_name()` returns a *canonical name*, the fully qualified determinants in the reverse order of the declaration of the fields. For example, `TRUE' corrupt big' size packet` is a canonical name of one of `packet`'s subtypes, given that field `corrupt` was defined after field `size`. 35

30.2.4.2 Explicit and significant subtypes 40

A **when** variant of a struct is called an *explicit* subtype if it is explicitly given some distinctive structural or behavioral content by some **when** or **extend** constructs (see [6.6](#) and [6.3](#)). Subtypes can be true variants of a struct, i.e., have distinct content, even when they are not explicitly defined: they consist of the conjunction of two or more explicit subtypes. These are called *significant* subtypes. Significant subtypes are important because each object in the program has exactly one type that describes it exhaustively (see [30.4.2](#)). 45

For example, the struct `packet` with enumerated field `size` (`big` or `small`) and Boolean field `corrupt` has four possible **when** subtypes. If only `big packet` and `corrupt packet` are defined as explicit variants of `packet` (using the constructs `when big packet {...}` and `when corrupt packet {...}`), then only they are explicit subtypes. In this case, `corrupt big packet` is also a significant subtype, since it has some distinctive features. On the other hand, `small packet` is neither explicit nor significant, since instances of it are equivalent to instances of `packet`. 50

55

30.2.4.3 Generalized relationships

There are a number of generalized relations which apply to both **like** structs and **when** subtypes, such as containment and mutual exclusion. However, regular inheritance relations, e.g., whether a struct is a direct parent type or a direct subtype of another, are applicable only to **like** structs.

- a) **rf_struct.is_contained_in**(*rf_struct*): bool

Returns TRUE if every instance of this struct is an instance of *rf_struct*. Otherwise, returns FALSE. For example, *bulldog* is contained in *dog*; whereas, *corrupt small packet* is contained in *small packet*, in *corrupt packet*, in *packet*, and in itself, but *small packet* is not contained in *corrupt packet*.

- b) **rf_struct.is_disjoint**(*rf_struct*): bool

Returns TRUE if every instance of this struct is not an instance of *rf_struct* and vice versa; i.e., the two types are mutually exclusive. Otherwise, returns FALSE. **like** structs are disjoint if they are not identical and neither one is contained in the other, e.g., *bulldog* and *poodle* are disjoint, *big packet* and *small packet* are disjoint, but *big packet* and *corrupt packet* are not.

- c) **rf_struct.is_independent**(*rf_struct*): bool

Returns TRUE if an instance of this struct is possibly, but not necessarily, an instance of *rf_struct*. Otherwise, returns FALSE. This relation holds only between two **when** subtypes that are neither contained nor mutually exclusive. For example, *big packet* is independent of *corrupt packet*, but not of *corrupt small packet*.

- d) **rf_struct.get_when_base**(): rf_like_struct

Returns the struct which is the base of the **when** struct family. This struct itself is returned if it is not a **when** subtype (regardless of whether it actually contains **when** subtypes).

30.2.4.4 rf_like_struct

This method has *like* inheritance from **rf_struct** (see [30.2.2](#)).

- a) **rf_like_struct.get_supertype**(): rf_like_struct

Returns the immediate **like** parent type of this struct.

- b) **rf_like_struct.get_direct_like_subtypes**(): list of rf_like_struct

Returns the set of immediate subtypes of this struct in the **like** struct hierarchy.

- c) **rf_like_struct.get_all_like_subtypes**(): list of rf_like_struct

Returns the set of all subtypes of this struct in the **like** struct hierarchy.

- d) **rf_like_struct.get_when_subtypes**(): list of rf_when_subtype

Returns the set of all defined subtypes in the when struct family for this struct. If this struct is not a leaf in the *like* hierarchy (i.e., it has **like** subtypes), the method returns an empty list. Any subtypes that are significant, but not defined, are not returned (see [30.2.4.5](#)).

30.2.4.5 rf_when_subtype

This method has *like* inheritance from **rf_struct** (see [30.2.2](#)).

- a) **rf_when_subtype.get_short_name**(): string

Returns a short version of the canonical name, i.e., without determinant qualification unless ambiguity requires it. For example, a type whose canonical name is *corrupt' TRUE big' size packet* would (normally) have the short name *corrupt big packet*. *Qualified determinants* appear in the short name when the same value name is a possible value of more than one field.

- b) **rf_when_subtype.is_explicit()**: bool 1
Returns TRUE if this **when** subtype is explicitly defined in the program (by a **when** or an **extend** construct). Otherwise, returns FALSE (for both *significant* and *insignificant* subtypes).
- c) **rf_when_subtype.get_contributors()**: list of rf_when_subtype 5
Returns the set of subtypes that contribute to the definition of this subtype, i.e., all the explicit subtypes where this subtype is contained, including itself if that case is explicitly defined.

30.2.5 List types 10

This subclause defines the list types.

30.2.5.1 rf_list 15

This method has *like* inheritance from **rf_type** (see [30.2.1.2](#)).

Lists are multi-purpose containers in *e*. The different list types are all instances of a generic definition similar to a parameterized type (template) in other OO languages. Any non-list type (scalars, strings, or structs) can serve as the element type of a list. 20

- a) **rf_list.get_element_type()**: rf_type
Returns the element type of this list, e.g., the element type of `list of big packet` is `big packet`.
- b) **rf_list.is_packed()**: bool 25
Return TRUE if this list is *packed* (a list with an element type whose size in bits is 16 or less). Otherwise, returns FALSE.

30.2.5.2 rf_keyed_list 30

This method has *like* inheritance from **rf_list** (see [30.2.5.1](#)).

rf_keyed_list.get_key_field(): rf_field

Returns the field by which the list is mapped or NULL if the key is the object itself (i.e., when the list is defined as `(key: it)`). 35

30.2.6 Scalar types

Scalars in *e* have value semantics in assignment, parameter passing, equivalence, operators, etc. They are either enumerated, numeric, or Boolean types. 40

30.2.6.1 rf_scalar

This method has *like* inheritance from **rf_type** (see [30.2.1.2](#)).

- a) **rf_scalar.get_size_in_bits()**: int 45
Returns the size of this scalar type in bits.
- b) **rf_scalar.get_range_string()**: string 50
Returns a string representation of the scalar range of values in the format of range modifiers (e.g., the string "[1..4, 7, 9..10]").

30.2.6.2 rf_numeric

This method has *like* inheritance from **rf_scalar** (see [30.2.6.1](#)). 55

1 **rf_numeric.is_signed():** bool

 Returns `TRUE` if the numeric type is signed. Otherwise, returns `FALSE`.

5 **30.2.6.3 rf_enum**

 This method has *like* inheritance from **rf_scalar** (see [30.2.6.1](#)).

10 a) **rf_enum.get_items():** list of `rf_enum_item`

 Returns the set of named values for this type. The legal values of an **enum** type (see [4.3.2.3](#)) are not restricted by a range declaration, e.g., the type introduced by the statement `type my_color: color [red..blue]` has the same items as `type color`. Such declarations only effect generation properties of the type.

15 b) **rf_enum.get_item_by_value(value: int):** `rf_enum_item`

 Returns the named value object for *value* or `NULL` if no such value exists in this type's range.

20 c) **rf_enum.get_item_by_name(name: string):** `rf_enum_item`

 Returns the named value object for *name* or `NULL` if no value by such name exists in this type's range.

25 **30.2.6.4 rf_enum_item**

 Enum items are pairs of identifier-integer, which are the possible values of a variable of that **enum** type. The integer values of **enum** items are the numbers assigned to them explicitly in the declaration (e.g., [`red = 3`, `green = 17`]) or the default (consecutive) numbers.

30 a) **rf_enum_item.get_name():** string

 Returns the identifier associated with this item as a string.

35 b) **rf_enum_item.get_value():** int

 Returns the integer value associated with this item as a signed integer.

Issue
728

40 **30.2.6.5 rf_bool**

 This method has *like* inheritance from **rf_scalar** (see [30.2.6.1](#)).

45 Boolean types in *e* are the predefined type **bool** and its (possibly user-defined) width derivatives such as `bool (bits:8)`. Boolean types have no special features others than those declared for **rf_scalar**.

50 **30.2.6.6 rf_string**

 This method has *like* inheritance from **rf_type** (see [30.2.1.2](#)).

55 Strings in *e* are instances of a special built-in type, which is neither scalar nor compound (a struct or list). Unlike the other three types (enumerated, numeric, or Boolean), there is only one string type; thus, the meta-type **rf_string** is a singleton. It does not have any special features other than those declared for **rf_type**.

Issue
728

55 **30.2.7 Port types**

 Ports in *e* are special purpose objects that serve to bind different units in the verification environment and specifically to interconnect with the DUT. Each port is an instance of one the four port types. There are three kinds of parameterized port types: `simple_port`, `buffer_port`, and `method_port`, and a non-parameterized kind: `event_port`.

30.2.7.1 rf_port	1
This method has <i>like</i> inheritance from rf_type (see 30.2.1.2).	
a) rf_port.is_input(): bool	5
Returns TRUE if this port type is declared as an input with the in or the inout specifier.	
b) rf_port.is_output(): bool	10
Returns TRUE if this port type is declared as an output with the out or the inout specifier.	
30.2.7.2 rf_simple_port	
This method has <i>like</i> inheritance from rf_port (see 30.2.7.1).	
rf_simple_port.get_element_type(): rf_type	15
Returns the element type of this port type.	
30.2.7.3 rf_buffer_port	
This method has <i>like</i> inheritance from rf_port (see 30.2.7.1).	20
rf_buffer_port.get_element_type(): rf_type	
Returns the element type of this port type.	
30.2.7.4 rf_event_port	25
This method has <i>like</i> inheritance from rf_port (see 30.2.7.1).	
30.2.7.5 rf_method_port	30
This method has <i>like</i> inheritance from rf_port (see 30.2.7.1).	
rf_method_port.get_element_type(): rf_type	
Returns the method type of this method port.	
30.2.8 Querying for types: rf_manager	35
The starting point in every query into the type information is a set of services that are not related to any specific kind of meta-objects. They are scoped together as methods of a singleton class named rf_manager , the instance of which is under global . This struct has other general services that are defined in 30.3.5 , 30.4.1 , and 30.4.4 .	40
a) rf_manager.get_type_by_name(name: string): rf_type	
Returns the type with <i>name</i> or NULL if no type by that name exists in the system.	
b) rf_manager.get_user_types(): list of rf_type	45
Returns a list of all the types declared in the user modules.	
30.3 Aspect information	
The structure and behavior of objects at runtime consists of fields, methods, events, etc. In <i>e</i> , the definition of these constituents can be separated into different modules of the software and extended on a per-type basis as part of the aspect-oriented modeling paradigm (i.e., decomposed as different concerns). Thus, the mapping between how the code is laid out and imported, and the end-result of accumulated software layers once all of the extensions have been resolved, is non-trivial. The structure of the definitions in the code is modeled in the Reflection API by a new kind of meta-information.	55

1 Reflecting this information is an important component of the API. It can be viewed as the mapping between
 named entities and the structure of their definitions in the source code. Meta-objects that represent the
 elements of the definition of named entities are called *definition elements*.

5 30.3.1 Definition elements

This describes the definition elements.

Issue
729

During the Draft 3 Review, compare this material against other parts of IEEE Std 1647-2005

10 30.3.1.1 Extensible entities and layers

15 In common OO languages, the definition of a class begins and ends in one single stretch of code. Conversely
 the definition of structs in *e* can be separated between different locations in the source files. It is introduced
 and initially defined by a **struct** statement and then possibly further defined by later **extend** statements.
 Each such “piece” of definition is called a *struct layer*. An enumerated type can similarly be initially defined
 with some set of named values and extended later with more named values. Each of these is an *enum layer*.

20 Methods can be overridden or refined not only in subtypes, but also later in the same struct (by **is also/first/
 only** constructs (see [17.1.3](#)). Thus, the definition of a method for some given object is a series of one or
 more definition “pieces” which are called *method layers*. Events, like methods, are declared once in some
 struct and are possibly overridden later in the same struct or in subtypes. These concepts are explained
 below (see [30.3.3](#)).

25 Generally speaking, entities that can be declared at one location in the source code and extended in later
 locations, such as struct types, enum types, methods and events, are called extensible entities. The definition
 of *extensible entities* consists of a series of one or more elements (layers), the first of which is the
declaration and the rest are *extensions*.

30 30.3.1.2 Anomalies of definition elements

35 The separation between a named entity and its definition is natural where extensible entities are concerned.
 However, it is somewhat artificial for non-extensible entities, e.g., numeric types and fields. Nevertheless,
 the same scheme applies trivially to non-extensible entities. Their definition consists of exactly one
 element—the *declaration*. For example, the **rf_field** object (see [30.4.3](#)) that represents the field `size` of the
 struct `packet` can be queried for the source location of its declaration, not directly, but through a different
 object (of type **rf_definition_element** (see [30.3.1.3](#)) which represents its declaration.

40 Moreover, some named entities are not explicitly defined by *e* code at all and so have no definition elements
 whatsoever, not even a declaration. For example, list types are instantiations of a parameterized built-in
 type. They are used in *e* code and represented in the type system just as any other type, but they are never
 defined by *e* code itself. See also: [30.3.2](#).

Issue
730

45 30.3.1.3 rf_definition_element

- a) **rf_definition_element.get_defined_entity():** `rf_named_entity`
 Returns the named entity that is being defined by this definition element; i.e., this element is part of
 the definition of the returned named entity.
- b) **rf_definition_element.get_module():** `rf_module`
 Returns the module where this definition element appears.
- c) **rf_definition_element.get_source_line_no():** `int`
 Returns the line number of the beginning of the clause in the source file.

- d) **rf_definition_element.is_before**(*rf_definition_element*): bool 1
Returns TRUE if this definition element appears before *rf_definition_element* in the load order. Otherwise, returns FALSE. This is based on a *full-order relation* on definition elements, which is defined as the ordinal number of modules and then the line number in the file. Issue 730 5
- e) **rf_definition_element.get_documentation**(): string
Returns the inline documentation of this definition element. *Inline documentation* is the comment in the consecutive lines directly preceding the definition in the source files. An empty string is returned if the source file is not found. 10
- f) **rf_definition_element.get_documentation_lines**(): list of string
Returns the inline documentation of this definition element as a list of strings separated by new-line characters in the source file. An empty list is returned if the source file is not found.
- g) **rf_named_entity.get_declaration**(): rf_definition_element 15
Returns the declaration (the first definition element) of this entity or NULL for any types defined implicitly as variants of existing types (see [30.3.2](#)).

30.3.2 Type layers 20

****During the Draft 3 Review, compare this material against other parts of IEEE Std 1647-2005****

enum and struct types are extensible entities, so their definition can consist of one or more layers. Other kinds of types are not extensible and so have only the declaring layer. However, not all types have explicit definitions. Some of these types can be used in context, without being previously declared: 25

- Numeric types can be used in context with a size modification (e.g., `uint (bits: 16)`). The size modification implies a different type, but one which has no explicit declaration.
- **enum** types can be spelled out inline, they have no separate declarations or explicit names.
- List types are instances of predefined parameterized types; they **are not declared or defined** in *e*.
- Not all **when** subtypes are explicitly defined, but they can still be used in context as types.

The first two cases are scalar types that could have been declared and given an explicit name by a **type** statement. In the other two cases, there is no way to make the type declaration explicit. Some **when** subtypes are explicitly defined in a different sense by using **when** or **extend** constructs. Even then, the layers of the **when** subtypes cannot always be separated from those of other subtypes or the when base. From the viewpoint of aspect information, all these cases are treated in the same way: All types that fall under one of the above cases do not have any layers. Calling the method **get_declaration**() (see [30.3.1](#)) on them returns NULL and for implicitly defined **enum**, calling **get_layers**() (see [30.3.2.1](#)) returns an empty list. As for structs, the service **get_layers**() is restricted to **like** structs. Issue 737 Issue 730 35

30.3.2.1 rf_type_layer 40

This method has *like* inheritance from **rf_definition_element** (see [30.3.1.3](#)). Issue 746 35

Structs and **enums** are extensible entities; they are defined in layers. Struct and enum layers are both type layers. This abstraction does not have features of its own, but is used by other services (see **get_type_layers**() in [30.3.4](#)). 40

30.3.2.2 rf_enum_layer 50

This method has *like* inheritance from **rf_type_layer** (see [30.3.2.1](#)).

- a) **rf_enum_layer.get_added_items**(): list of rf_enum_item 55
Returns the named values added by this enum layer.

- 1 b) **rf_enum.get_layers()**: list of `rf_enum_layer`
Returns all enum layers that constitute this enum type.

5 30.3.2.3 `rf_struct_layer`

This method has *like* inheritance from `rf_type_layer` (see [30.3.2.1](#)).

- 10 a) **rf_struct_layer.get_field_declarations()**: list of `rf_definition_element`
Returns the field declarations added to the struct by this struct layer.
- 15 b) **rf_struct_layer.get_method_layers()**: list of `rf_method_layer`
Returns the method layers added to the struct by this struct layer.
- c) **rf_struct_layer.get_event_layers()**: list of `rf_event_layer`
Returns the event layers added to the struct by this struct layer.
- d) **rf_like_struct.get_layers()**: list of `rf_struct_layer`
Returns all struct layers that constitute this struct type.

20 30.3.3 Method and event layers

****During the Draft 3 Review, compare this material against Clause 17, Methods****

25 **Issue
738**

Once a method in *e* is declared for a given struct, it can never be replaced by a different method in a subtype or a later extension. Rather, all later modifications of the definition, in all three modes: **also**, **first**, and **only**, in extensions as well as in *when* subtypes and *like* heirs, are definition elements of the same method—they are *method layers*. For example, the method `bark()`, once declared for struct `dog`, is one and the same for all kinds of dogs. But different method layers may be executed upon calling `bark()` for different dog objects, so they display different behaviors.

30 The reason for this deviation from standard OO terminology is *e* can be used to modify the behavior in derived structs, as well as when variants, and in later extensions of that same struct. Therefore, the need to distinguish between the method (the common semantics or message) on the one side and the definition of the behavior associated with it for some set of objects on the other side is more acute. These same considerations and terminology also apply to *events*.

35 30.3.3.1 `rf_method_layer`

40 This method has *like* inheritance from `rf_definition_element` (see [30.3.1.3](#)).

- 45 a) **rf_method_layer.get_context_layer()**: `rf_struct_layer`
Returns the struct layer where this method layer appears.
- b) **type `rf_extension_mode`**: [`empty`, `undefined`, `is`, `also`, `first`, `only`]
rf_method_layer.get_extension_mode(): `rf_extension_mode`
Returns one of the values: `empty`, `undefined`, `is`, `also`, `first`, or `only`, according to how this method layer was declared.
- 50 c) **rf_method.get_layers()**: list of `rf_method_layer`
Returns a list of all layers of this method in all struct types where it is defined. The returned list is ordered by load order from early to late.
- 55 d) **rf_method.get_relevant_layers(*rf_struct*)**: list of `rf_method_layer`
Returns a list of all layers of this method that apply to *rf_struct*. If *rf_struct* does not have this method at all, an empty list is returned. For example, the method **to_string()** (see [27.4.4](#)) is defined for every struct in *e*, so calling **get_layers()** returns all extensions of this method in the system.

However, calling `get_relevant_layer()` for the struct `packet` only returns the extensions of `to_string()` defined in the context of the struct `packet` and its subtypes. 1

30.3.3.2 rf_event_layer 5

This method has *like* inheritance from `rf_definition_element` (see [30.3.1.3](#)).

- a) `rf_event_layer.get_context_layer(): rf_struct_layer`
Returns the struct layer where this event layer appears. 10
- b) `rf_event_layer.get_extension_mode(): rf_extension_mode`
Returns either `is` or `only`, according to how this event layer was declared (see **type `rf_extension_mode`** in [30.3.3.1](#)). Other extension modes, such as `first` or `also`, are not applicable to events. 15
- c) `rf_event.get_layers(): list of rf_event_layer`
Returns a list of all layers of this event in all struct types where it is defined. The returned list is ordered by load order from early to late. 20

30.3.4 Modules and packages 20

This describes the modules and packages. 25

30.3.4.1 rf_module 25

****During the Draft 3 Review, compare this material against other parts of IEEE Std 1647-2005****

Modules are simply *e* files. However, with the ability to extend structs and separate different concerns or crosscuts of a system, modules play an important role in organizing the program. If a struct may be considered the vertical encapsulation principle, then modules are the horizontal one. A struct consists of a number of related layers of definition in different modules and, symmetrically, the module consists of a number of related layers of different structs—it can be thought of as a layer of the entire system. Thus, modules can be queried for their overall contribution to the structure of a system in the reflection API. 30

- a) `rf_module.get_name(): string`
Returns the name of this module, basically the name of the *e* file without the `.e` extension. 35
- b) `rf_module.get_index(): int`
Returns this module's ordinal number in the load order. 40
- c) `rf_module.get_type_layers(): list of rf_type_layer`
Returns a list of all the type layers defined in this module. A module's overall contribution to the structure of a system is the set of declarations of new types and extensions of existing types. 45
- d) `rf_module.get_package(): rf_package`
Returns the *e* package with which this module is associated. Any modules which are not explicitly associated with some package (using the **package** statement (see [22.1](#))) are implicitly part of the package `main`. 50
- e) `rf_module.is_user_module(): bool`
Returns `TRUE` if the module is user-defined. Otherwise, returns `FALSE`.
- f) `rf_module.is_encrypted(): bool`
Returns `TRUE` if the module is encrypted. Otherwise, returns `FALSE`. 55

30.3.4.2 rf_package

A *package* (see [Clause 1](#)) is a set of one or more *e* modules which together implement some closely related functionality. This package defines a scope for restricting the access of named entities. It also is represented in the reflection API by a meta-object.

- a) **rf_package.get_name()**: string
Returns the name of this package.
- b) **rf_package.get_modules()**: list of rf_module
Returns the set of modules associated with this package.

Issue
739

30.3.5 Querying for aspects

Similar to the services for type information queries (see [30.2.8](#)), the following services can be used to perform aspect information queries.

- a) **rf_manager.get_module_by_name(name: string)**: rf_module
Returns the module with the *name* or NULL if no module by this name is currently loaded.
- b) **rf_manager.get_module_by_index(index: int)**: rf_module
Returns the module with the given *index* in the load order.
- c) **rf_manager.get_user_modules()**: list of rf_module
Returns a list of all user modules that are currently loaded.
- d) **rf_manager.get_package_by_name(name: string)**: rf_package
Returns the package with the *name* or NULL if no package by this name is currently loaded.

30.4 Value query and manipulation

The parts of the API described in previous subclauses, type information and aspect information, both reflect static features of the program. During a run of the program, values are being manipulated. Each of those values is an instance of a type and all operations carried upon them are defined by their type. This part of the API enables the user to query and manipulate values using the representations of types. This feature is known as *meta-programming*. It can serve to construct data browsers, debugging aids, and other generic runtime features.

Issue
746

See also: [5.2](#) and [value_unsafe \(30.4.3\)](#).

30.4.1 Types of objects

The natural entry point for querying or manipulating objects is getting a representation of their type. Each object has exactly one representative struct type: a **when** subtype or a **like** struct. A query can be generated for the **like** struct of an instance and any **when** variants discarded, or the query can be for the specific **when** subtype. The **when** subtype of an instance depends on its state, which may change with the course of the run.

Issue
740

- a) **rf_manager.get_struct_of_instance(instance: base_struct)**: rf_like_struct
Returns the most specific **like** struct of the struct *instance* and disregards any **when** variants, even if they apply to the instance. To query for the specific **when** subtype of an object, use: **get_exact_subtype_of_instance()**.
- b) **rf_manager.get_exact_subtype_of_instance(instance: base_struct)**: rf_struct
Returns the type of *instance*. The returned meta-object represents the most specific significant type that applies to the *instance*, i.e., the one containing all other types that apply to the instance. For example, if the parameter is a `packet`, which has the defined subtypes `big packet` and `cor-`

rupt packet, and a particular packet happens to be both corrupt and big, then the returned type would be corrupt big packet, even though it is not a *defined* subtype. 1

The static type of a field is sometimes more specific than the exact subtype of the object which is the field's actual value: This happens when the static type is an insignificant **when** subtype (see [30.2.4.2](#)). 5

30.4.2 Values and value holders

When dealing with values in a generic way (meta-programming), there needs to be some safe way to refer to values of all types: struct instances, lists, strings, and scalars. Since these values are very different in their semantics and there is no abstract type common to all, the reflection API wraps values of all types with an object called **rf_value_holder**. This object holds a value together with its type, and guarantees its consistency and continuity when the original variable goes out of scope, and across garbage collections. 10

Value holders are returned from value queries or explicitly created by the user. They are used in setting values or calling methods. Actual uses of the value itself, however, involve passing through an *untyped* value and brute casting, which is not type-safe. Two operators are implemented generically for all values—equating and getting a string representation. 15

- a) **rf_value_holder.get_type():** rf_type 20

Returns the type of this value. When the value is a struct instance, the type of the holder is not necessarily the most specific subtype of that instance (e.g., a legal value holder whose type is **any_struct** can hold an instance of `packet`).

- b) **rf_value_holder.get_value():** untyped 25

Enables (by using the **value_unsafe** operator (see [30.4.3](#))) assignment of the value into a typed variable. The variable shall be a type to which this value is assignable according to *e* casting rules; however, this cannot be enforced and the user's responsibility to confirm. **Issue 746**

- c) **rf_type.create_holder(value: untyped):** rf_value_holder 30

Returns a value holder of this type for *value*, which shall be an instance of this type (or of a subtype in case it is a struct type). Very simple sanity checks are performed on the value; if they fail, an exception is thrown. These checks are by no means exhaustive; it is the user's responsibility to create the right holder for a value.

- d) **rf_type.value_is_equal(value1: untyped, value2: untyped):** bool 35

Returns TRUE if *value1* and *value2* are equivalent or identical (using the same semantics as that of the `==` operator (see [4.10.2](#))). Otherwise, returns FALSE. The behavior is not defined if one of the two values is not of this type.

- e) **rf_type.value_to_string(value: untyped):** string 40

Returns a string representation of *value* (this is the same as the **to_string()** operator). The behavior is not defined if the value is not of this type.

30.4.3 Object operators

Object operators include reading and writing fields, calling methods, and emitting or monitoring events. These operators are available with meta-objects that represent struct members. Value holders are the safe way to handle values in a generic way (see [30.4.2](#)). However, using them involves the dynamic allocation of memory, which can impact performance where this feature is heavily used. 45

Each object operator has two versions: One uses value holders and makes some checks, throwing exceptions in the cases where preconditions do not hold. The other uses bare *untyped* values (see [5.2](#)) and skips checks. This brute version of each operator is marked as `unsafe` and should be avoided where possible. 50

55

- 1 a) **rf_field.get_value**(*instance*: base_struct): rf_value_holder
Returns the value of this field in the struct *instance*. If the struct type of the instance does not have this field, an exception is thrown.
- 5 b) **rf_field.get_value_unsafe**(*instance*: base_struct): untyped
Returns the value of this field in the struct *instance*. If the struct type of the instance does not have this field, the behavior is undefined.
- 10 c) **rf_field.set_value**(*instance*: base_struct, *value*: rf_value_holder)
Sets the value of this field in the struct *instance* to the new *value*. If the struct type of the instance does not have this field, an exception is thrown.
- 15 d) **rf_field.set_value_unsafe**(*instance*: base_struct, *value*: untyped)
Sets the value of this field in the struct *instance* to the new *value*. If the struct type of the instance does not have this field, the behavior is undefined.
- 20 e) **rf_method.call**(*instance*: base_struct, *parameters*: list of rf_value_holder): rf_value_holder
Calls this method on the struct *instance*, using the list of (zero or more) values as the method's *parameters*, and returns a value holder of the method's return value (or NULL if the method has none). If the struct type of the instance does not have this method or there is a mismatch in the number and types of parameters, an exception is thrown.
- 25 f) **rf_method.call_unsafe**(*instance*: base_struct, *parameters*: list of untyped): untyped
Calls this method on the struct *instance*, with the list of (zero or more) values as the method's *parameters*, and returns the method's return value. If this method does not return a value, the value returned from **call_unsafe** is undefined. If the struct type of the instance does not have this method or there is a mismatch in the number and types of parameters, the behavior is undefined.
- 30 g) **rf_event.is_emitted**(*instance*: base_struct): bool
Returns TRUE if this event of the *instance* was emitted so far in the current cycle. Otherwise, returns FALSE. If the struct type of the instance does not have this event, an exception is thrown.
- 35 h) **rf_event.is_emitted_unsafe**(*instance*: base_struct): bool
Returns TRUE if this event of the *instance* was emitted so far in the current cycle. Otherwise, returns FALSE. If the struct type of the instance does not have this event, the behavior is undefined.
- 40 i) **rf_event.emit**(*instance*: base_struct)
Emits the event on the *instance*. If the struct type of the instance does not have this event, an exception is thrown.
- j) **rf_event.emit_unsafe**(*instance*: base_struct)
Emits the event on the *instance*. If the struct type of the instance does not have this event, the behavior is undefined.

30.4.4 List operators

45 The three main list operators: reading an element, writing to an index, and querying the size are available as general services, i.e., methods of **rf_manager**. Two versions of the operators are available: the safe version, using value holders, and the brute one, using untyped values. See also: [30.4.3](#).

- 50 a) **rf_manager.get_list_element**(*list*: rf_value_holder, *index*: int): rf_value_holder
Returns the value of the given *list* at the given *index*. If the value is not a list or the index is out of bounds, an exception is thrown.
- 55 b) **rf_manager.get_list_element_unsafe**(*list*: untyped, *index*: int): untyped
Returns the value of the given *list* at the given *index*. If the value is not a list or the index is out of bounds, the behavior is undefined.

- | | | |
|----|---|----|
| c) | rf_manager.set_list_element (<i>list</i> : rf_value_holder, <i>index</i> : int, <i>new_value</i> : rf_value_holder) | 1 |
| | Sets the value of the given <i>list</i> at the given <i>index</i> . If the first parameter is not a list, the index is out of bounds, or the new value is not of an instance of the list's element type, an exception is thrown. | |
| d) | rf_manager.set_list_element_unsafe (<i>list</i> : untyped, <i>index</i> : int, <i>new_value</i> : untyped) | 5 |
| | Sets the value of the given <i>list</i> at the given <i>index</i> . If the first parameter is not a list, the index is out of bounds, or the new value is not of an instance of the list's element type, the behavior is undefined. | |
| e) | rf_manager.get_list_size (<i>list</i> : rf_value_holder): int | 10 |
| | Returns the number of elements currently in the given <i>list</i> . If the value is not a list, an exception is thrown. | |
| f) | rf_manager.get_list_size_unsafe (<i>list</i> : untyped): int | 15 |
| | Returns the number of elements currently in the given <i>list</i> . If the value is not a list, the behavior is undefined. | |
| | | 20 |
| | | 25 |
| | | 30 |
| | | 35 |
| | | 40 |
| | | 45 |
| | | 50 |
| | | 55 |

Issue 743

1

5

10

15

20

25

30

35

40

45

50

55