

# 6 Reflective Facility for e

## 6.1 Introduction

This section covers the following topics:

- “What Is Reflection?” on page 6-1
- “Uses of Reflection” on page 6-1
- “Overview” on page 6-2
- “This Document” on page 6-2
- “Terminology and Conventions” on page 6-3

### 6.1.1 What Is Reflection?

Reflection (sometime called introspection) is a programmatic interface into the meta-data of a program. Most object oriented 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 reflective facility than that of other languages.

### 6.1.2 Uses of Reflection

Reflection interfaces are typically used for constructing external developer aid tools such as class browsers, data browsers, debugging aids, source browsers, and the like. Other uses can be for the implementation of generic utilities, where the modeling powers of the language such as inheritance and polymorphism are not strong enough. Examples are object serialization utilities like packing or register filling. These utilities need to deal with all types of data, primitive (machine supported) types as well as different compound types, and cannot be implemented with the standard terms of the language. The reflection interface in *e* is designed to be used by advanced programmers, and to facilitate third-party

applications and tools, to which these problems are typical. The Specman built-in data-browser, for instance, is implemented using it. It is not meant to be used by an *e* end user—a test writer—where it could be tempting to use it in order to bypass the static type checks.

### 6.1.3 Overview

The *e* reflective facility is a class API, similar in principle to Java's Reflection API (and is also called Reflection API henceforth). 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* meta-types form a hierarchy of abstractions, and show the different relations between such entities. A representation of the struct-type *any\_struct*, for example, can return a list of representations of *any\_struct*'s methods, among them is the representation of the method *to\_string()*. All of these meta-types have a common prefix to their name—the letters *rf*.

The definition of this interface should also serve as an agreed-upon conceptual model of types and structural elements in *e*, and supply a unified terminology for talking about them and the relations between them.

Here is a trivial use example of the proposed API, which gives an idea of how it works. It prints all the fields in struct *packet*:

```
var rfs: rf_struct = rf_manager.get_struct_by_name("packet");
print rfs.get_fields().apply(it.get_name());
```

The first line assigns to a variable the object representing the struct *packet* (a meta-object). The second obtains from it the set of objects representing *packet*'s fields, and prints their names.

The Reflection API as described by this document is focused on type-related meta-data of a program. There are, however, more structural elements in *e* software system that go beyond types and their constitution. Examples are *eRM* packages, sequences, *message* actions, *check* actions, coverage-related entities, expects, and more. Reflecting such entities and others is a natural future enhancement of the API. The Reflection API should eventually become a comprehensive programmatic interface into every structural aspect of system.

### 6.1.4 This Document

This document defines the Reflection API in detail. The interface is divided into three main parts—*type information*, *aspect information*, and *value query and manipulation*, which correspond to chapters 2, 3 and 4. 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 (for example, some methods of struct *rf\_event*

are described in chapter 2, others in chapter 3 and the rest in chapter 4). Nevertheless, meta-types are introduced in order from the abstract to the concrete, and where one meta-type depends on another, it is introduced later.

Each meta-type is introduced in a separate numbered section. The location in the type hierarchy is fixed by a ‘like’ clause immediately after the heading. This has the usual inheritance implications (for example, the method *is\_private()*, which is defined for struct *rf\_struct\_member*, exists also for *rf\_field*, which is like *rf\_struct\_member*). The concept behind each meta-type is first explained, and then its methods are listed and explained. The method signature is introduced in pseudo *e* code and its semantics is explained in free text.

The document aims to explain the model of types in *e* as it is captured by Reflection API, as well as to define the API in details. These two tasks are interwoven together in the mode of presentation described above. Therefore, some parts are simply “reference manual” style definitions, while others require more elaborate explanations that go beyond the scope of API documentation.

At the end of each chapter there is a use example—a little utility that is implemented in terms of the relevant part of the API. These examples are based on possible real-life needs, but they are very simplistic. They are written with the only purpose to illustrate how this of the API might be used.

## 6.1.5 Terminology and Conventions

- Many technical terms are defined and used in the body of this document. It would be unhelpful to list them and define them in advance, since they require the context to be understood. In order to identify these terms easily they are italicized throughout, and their first appearance, where they are contextually defined, is bold as well.

This said underlined, but I changed the italic underlined text to **bi\_bolditalics**. If you want to get rid of this convention, it will be easy to find and change them. The only other use of bold italics is in the *e* variable. --Linda

- There is a danger of confusion when dealing with meta-data and meta-types, as objects are used to represent types, methods and so forth, 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 used as short for ‘an object representing the type/method/field’. For example, the method *rf\_struct.get\_methods()* is said to return 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* inheritance and *when* subtyping is a major concern of the reflection model and of this document. Therefore two trivial and well known examples are used in many places to illustrate the definitions. One is the hierarchy of *dog*, with *bulldog* and *poodle* as its like heirs. The other is struct *packet* having an enumerated field *size*, and a boolean field *corrupt*, which allow for *when* subtypes such as *small packet* or *big corrupt packet*.

## 6.2 Type Information

The core of 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. Note that units are simply structs from Reflection API's point of view and throughout this document (see "rf\_struct" on page 6-5 for how to query if a struct is a unit).

This section covers the following topics:

- "Named Entities" on page 6-4
- "Struct Types" on page 6-5
- "Struct Members" on page 6-6
- "Inheritance and When Subtypes " on page 6-9
- "List Types" on page 6-12
- "Scalar Types" on page 6-12
- "Querying for Types" on page 6-14
- "Use Example" on page 6-14

### 6.2.1 Named Entities

#### 6.2.1.1 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 all are either visible or hidden. The importance of this abstraction is related to the next chapter—aspect information.

- **rf\_named\_entity.get\_name(): string**  
Returns the name of this entity.
- **rf\_named\_entity.is\_visible(): bool**  
Returns TRUE if this entity is visible. Invisible named entities are Specman internal entities, including fields and methods of user defined structs, that are not shown in printing and visualization tools.

## 6.2.1.2 rf\_type

### like rf\_named\_entity

Not sure what that line above should be. I've made them all 4heads, but if so, the “like” s/b capped. -L

Each value that is manipulated during the run of a program is an instance of some type. A type is either built in or user defined, explicitly defined or implicitly inferred. Types in *e* are divided into three categories—scalar-types, list-types, and struct-types.

- **rf\_type.is\_public(): bool**

Returns TRUE if this type is not declared with *package* modifier, that is, it has unrestricted access.

## 6.2.2 Struct Types

### 6.2.2.1 rf\_struct

#### like rf\_type

Structs in *e* are equivalent to classes in other OO languages. Struct types have three kinds of members—fields, method and events, which are represented in Reflection API as named entities. There are other kinds of struct members in *e*, like constraints, cover-groups, expects and more, which are currently not reflected in the interface (it is a possible future enhancement of the API). Each of these members is either declared in the context of the struct itself or inherited from a supertype (and possibly refined in the struct). For example, the method *to\_string* is declared in the context of *base\_struct* and inherited by all other structs. Units are also represented by *rf\_struct* objects, and being a unit is a property that can be queried.

- **rf\_struct.get\_fields(): list of rf\_field**

Returns a list containing all fields of this struct (declared by it or inherited from its supertype).

- **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()*).

- **rf\_struct.get\_field(name: string): rf\_field**

Returns the field of this struct with the name given as parameter, or NULL if no such field exists. Note that field names are unique in the context of a struct.

- **rf\_struct.get\_methods(): list of rf\_method**

Returns a list containing all methods of this struct (declared by it or inherited from its supertypes).

- **rf\_struct.get\_declared\_methods(): list of rf\_method**

Returns a list containing all methods declared in the context of this struct (a subset of *rf\_struct.get\_methods()*). Note that methods that are declared by a supertype and extended or overridden in the context of this struct are not returned (see further discussion of method extension in section 6.3.3).

- **rf\_struct.get\_method(name: string): rf\_method**

Returns the method of this struct with the name given as parameter, or NULL if no such method exists. Note that method names are unique in the context of a struct.

- **rf\_struct.get\_events(): list of rf\_event**

Returns a list of all events of this struct (declared by it or inherited from its supertype).

- **rf\_struct.get\_declared\_events(): list of rf\_event**

Returns a list of all events declared in the context of this struct (a subset of *rf\_struct.get\_events()*). Note that events that are declared by a supertype and overridden in the context of this struct are not returned (see further discussion of overriding events in section 6.3.3).

- **rf\_struct.get\_event(name: string): rf\_event**

Returns the event of this struct with the name given as parameter, or NULL if no such event exists. Note that event names are unique in the context of a struct.

- **rf\_struct.is\_unit(): bool**

Returns TRUE if this struct is a unit. This is the only indication that this meta-object represents a unit rather than a regular struct type.

## 6.2.3 Struct Members

### 6.2.3.1 rf\_struct\_member

#### like rf\_named\_entity

Struct members are represented by instances of the meta-types *rf\_field*, *rf\_method* and *rf\_event*. 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 the default—public) are assigned to it upon its declaration

- **rf\_struct\_member.get\_declaring\_struct(): rf\_struct**

Returns the struct in which this member was introduced. This applies also to empty definition of methods or declarations of undefined methods.

- **rf\_struct\_member.applies\_to(rf\_struct): bool**

Returns TRUE if this struct member applies to instances of the struct given as parameter, that is, if it was declared by that struct or by a different struct that contains it (see section 6.2.4 for definition of containment).

- **rf\_struct\_member.is\_private(): bool**

Returns TRUE if this struct member was declared with *private* access modifier, which means it is accessible only within the context of both its package and its declaring struct or its subtypes.

- **rf\_struct\_member.is\_protected(): bool**

Returns TRUE if this struct member was declared with *protected* access modifier, which means it is accessible only within the context of the declaring struct or its subtypes.

- **rf\_struct\_member.is\_package\_private(): bool**

Returns TRUE if this struct member was declared with *package* access modifier, which means it is accessible only within the context of the package in which it was declared.

- **rf\_struct\_member.is\_public(): bool**

Returns TRUE if this struct member was declared without an access modifier, that is, its access is not restricted.

### 6.2.3.2 rf\_field

#### like rf\_struct\_member

- **rf\_field.get\_type(): rf\_type**

Returns the declared type of the field.

- **rf\_field.is\_physical(): bool**

Returns TRUE if the field is declared physical (that is, with '%' modifier). Physical fields are the ones packed when the struct is packed.

- **rf\_field.is\_ungenerated(): bool**

Returns TRUE if the field is declared as ungenerated (that is, with '!' modifier). Ungenerated fields are not generated automatically when the struct is generated.

- **rf\_field.is\_unit\_instance(): bool**

Returns TRUE if the field is an instance of a unit (declared as 'is instance').

### 6.2.3.3 rf\_method

#### like rf\_struct\_member

- **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.

- **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.

- **rf\_method.is\_tcm(): bool**

Returns TRUE if this method is time consuming, that is, if it has a sampling event.

- **rf\_method.is\_inline(): bool**

Returns TRUE if the method is declared as *inline*.

- **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.

### 6.2.3.4 rf\_parameter

- **rf\_parameter.get\_name(): string**

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

- **rf\_parameter.get\_type(): rf\_type**

Returns the type of this parameter.

- **rf\_parameter.is\_by\_reference(): bool**

Returns TRUE if this parameter is passed by reference (that is, declared with '\*').

### 6.2.3.5 rf\_event

like rf\_struct\_member

## 6.2.4 Inheritance and When Subtypes

There are two mechanisms for subtyping in *e*. One is OO style single inheritance, in which a struct is declared as deriving from another—*like* inheritance. The other is closer to predicate classes, in which a behavioral or structural feature of an object is determined by some state or attribute of it. This is called *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 are represented by different kinds of meta-objects. Thus there are two kinds of structs types: *like structs* and *when subtypes*.

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 *when base*. Note that a *when* subtype can be determined by a field that is declared in the context of another *when* subtype. Such subtypes are, however, part of the same *when family* with the same *when base*.

There are a number of generalized relations which apply both to *when* subtypes and *like* structs. These are general set relations such as containment and mutual exclusion. Note that regular inheritance relations, such as whether a struct is a direct supertype or a direct subtype of another, are applicable only to *like* structs.

- **rf\_struct.is\_contained\_in(other: rf\_struct): bool**

Returns TRUE if every instance of this struct is necessarily an instance of the struct given as parameter. For example, *bulldog* is contained in *dog*, *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*.

- **rf\_struct.is\_disjoint(other: rf\_struct): bool**

Returns TRUE if every instance of this struct is necessarily not an instance of the struct given as parameter and vice versa, that is, the two types are mutually exclusive. Note that *like* structs are disjoint if they are not identical and neither one is contained in the other, for example, *bulldog* and *poodle* are disjoint, *big packet* and *small packet* are disjoint, but *big packet* and *corrupt packet* are not.

- **rf\_struct.is\_independent(other: rf\_struct): bool**

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

- **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 has *when* subtypes).

### 6.2.4.1 rf\_like\_struct

#### like rf\_struct

- **rf\_like\_struct.get\_supertype(): rf\_like\_struct**

Returns the immediate like supertype of this struct.

- **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.

- **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.

- **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 (that is, it has like subtypes), the method returns an empty list. Note that subtypes that are significant but not defined are not returned (see next section).

### 6.2.4.2 rf\_when\_subtype

#### like rf\_struct

#### Canonical Names

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—with fully qualified determinants or without them (for example, “big size packet” versus “big packet”), and in different determinant order (for example, “big corrupt packet” versus “corrupt big packet”). However, the method

`get_name()` will return a canonical name, the fully qualified determinants in the reverse order of the declaration of the fields. For example, “corrupt’TRUE big’size packet” is a canonical name of one of *packet*’s subtypes, given that field *corrupt* was defined after field *size*.

## Explicit and Significant Subtypes

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. Subtypes can be true variants of a struct, that is, have distinct content, even when they are not explicitly defined, in the case that 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 *exact subtype of instance* in section 6.4.2).

For example, struct *packet* with enumerated field *size* (either *big* or *small*) and boolean field *corrupt* has four possible *when* subtypes. If only *big packet* and *corrupt packet* were defined as explicit variants of *packet* (by the constructs “when big packet {...}” and “when corrupt packet {...}”) then only they are *explicit subtypes*. In this case also *corrupt big packet* is 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*.

- **rf\_when\_subtype.get\_short\_name(): string**

Returns a short version of the canonical name, that is, without determinant qualification unless ambiguity requires. 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 in case the same value name is a possible value of more than one field.

- **rf\_when\_subtype.is\_explicit(): bool**

Returns TRUE if this ‘when’ subtype is explicitly defined in the program (by a ‘when’ or an ‘extend’ construct). FALSE is returned both for *significant* and *insignificant* subtypes.

- **rf\_when\_subtype.get\_contributors(): list of rf\_when\_subtype**

Returns the set of subtypes that contribute to the definition of this subtype, that is all the explicit subtypes in which this subtype is contained, and possibly itself in case it is explicitly defined itself.

## 6.2.5 List Types

### 6.2.5.1 rf\_list

#### like rf\_type

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.

- **rf\_list.get\_element\_type(): rf\_type**

Returns the element type of this list. For example, the element type of *list of big packet* is *big packet*.

- **rf\_list.is\_packed(): bool**

Return TRUE if this list is packed (lists with element type whose size in bits is 16 or less are packed).

### 6.2.5.2 rf\_keyed\_list

#### like rf\_list

- **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)”).

## 6.2.6 Scalar Types

### 6.2.6.1 rf\_scalar

#### like rf\_type

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

- **rf\_scalar.get\_size\_in\_bits(): int**

Returns the size of this scalar type in bits.

- **rf\_scalar.get\_range\_string(): string**

Returns a string representation of the scalar range of values in the format of range modifiers (for example, the string “[1..4,7,9..10]”). Note that range modification of types does not affect type checking. It is rather a generation constraint on all instances of the type.

### 6.2.6.2 **rf\_numeric**

#### like **rf\_scalar**

- **rf\_numeric.is\_signed(): bool**

Returns TRUE if the numeric type is signed.

### 6.2.6.3 **rf\_enum**

#### like **rf\_scalar**

- **rf\_enum.get\_items(): list of rf\_enum\_item**

Returns the set of named values for this type. Note that the legal values of an enum type are not restricted by a range declaration. For example, the type introduced by the statement “type my\_color: color [red..blue]” has the same items as type color. Such declaration affects only generation properties of the type.

- **rf\_enum.get\_item\_by\_value(val: int): rf\_enum\_item**

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

- **rf\_enum.get\_item\_by\_name(name: string): rf\_enum\_item**

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

### 6.2.6.4 **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 (such as ‘[red = 3, green = 17]’) or the default (consecutive) numbers.

- **rf\_enum\_item.get\_name(): string**

Returns the identifier associated with this item as a string.

- **rf\_enum\_item.get\_value(): int**

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

### 6.2.6.5 rf\_bool

#### like rf\_scalar

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*.

### 6.2.6.6 rf\_string

#### like rf\_type

Strings in *e* are instances of a special built in type, which is neither scalar, nor compound (struct or list). Unlike the other three kinds, there is only one string type, and thus the meta-type *rf\_string* is a singleton. It does not have any special features other than those declared for *rf\_type*.

## 6.2.7 Querying for Types

### 6.2.7.1 rf\_manager

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 the next two chapters (see section 6.3.5, 6.4.1 and 6.4.4).

“scoped”? See second line of paragraph above.

- **rf\_manager.get\_type\_by\_name(name: string): rf\_type**

Returns the type with the name given as parameter, or NULL if no type by that name exists in the system.

- **rf\_manager.get\_user\_types(): list of rf\_type**

Returns a list of all the types that are declared in user modules.

## 6.2.8 Use Example

The following is an example of a very simple use of the type information interface. It is the implementation of a method that receives a struct name as parameter, and prints out the fields with their types, and the methods with their parameter and return types.

```
print_struct(name: string) is {  
    var s: rf_struct = rf_manager.get_type_by_name(name)
```

```
.as_a(rf_struct);

    outf("struct - %s\n",s.get_name());
    if s is a rf_like_struct (ls) {
        outf("\t inherits from - %s\n",
            ls.get_supertype().get_name()
        );
    };

    for each (f) in s.get_declared_fields() do {
        outf("\t field - %s: %s\n",
            f.get_name(),
            f.get_type().get_name()
        );
    };

    for each (m) in s.get_declared_methods() do {
        outf("\t method - %s()\n",m.get_name());
        for each (p) in m.get_parameters() do {
            outf("\t\t parameter - %s: %s\n",
                p.get_name(),
                p.get_type().get_name()
            );
        };
        if m.get_result_type() != NULL {
            outf("\t\t result type - %s\n",
                m.get_result_type().get_name()
            );
        };
    };
};
```

The following two files serve as a trivial design in order to show the output of the *print\_struct* utility (the same code will be referred to in the examples in the next chapters).

### file1.e

```
<'
type size_t: [big, medium, small];

struct packet {
    size: size_t;
    data: int (bits: 256);

    foo(id: int, name: string) is {
    };
};
```

```
extend sys {
    packets: list of packet;

    keep packets.size() > 3 and packets.size() < 7;
};
'>
```

## file2.e

```
<'
import file1.e;

extend packet {
    corrupt: bool;

    foo(id: int, name: string) is also {
    };

    bar(): int is {
    };
};
'>
```

This is output of running the utility on the code above:

```
Specman file2> print_struct("packet")
struct - packet
    inherits from - any_struct
    field - size: size_t
    field - data: int (bits: 256)
    field - corrupt: bool
    method - foo()
        parameter - id: int
        parameter - name: string
    method - bar()
        result type - int
```

## 6.3 Aspect Information

The definition of the structure and behavior of objects at runtime involves different concerns. In *e* these concerns can be separated into different modules of the software, as a part of the aspect oriented modeling paradigm. Breaking up the tie between the “layout” of the source code on the one hand, and the accumulated structure of the types on the other, brings with it a new kind of meta-information.

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*.

This section covers the following topics:

- “Definition Elements” on page 6-17
- “Type Layers” on page 6-19
- “Method and Event Layers” on page 6-20
- “Modules and Packages” on page 6-22
- “Querying for Aspects” on page 6-23
- “Use Example” on page 6-23

## 6.3.1 Definition Elements

### Extensible Entities and Layers

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*.

Methods can be overridden or refined not only in subtypes but also later in the same struct (by ‘is also/first/only’ construct). 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 (section 3.3).

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—*extensions*.

### Anomalies of Definition Elements

The separation between a named entity and its definition is natural where extensible entities are concerned. It is, however, somewhat artificial for non-extensible entities, such as numeric types and fields. Nevertheless, out of uniformity considerations the same scheme applies trivially to

non-extensible entities. Their definition consists of exactly one element—the *declaration*. For example, the *rf\_field* object that represents field *size* of struct *packet* can be queried for the source location of its declaration, not directly, but rather through a different object (of type *rf\_definition\_element*) which represents its declaration.

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 it. See further discussion of implicitly defined types in the next section.

### 6.3.1.1 **rf\_definition\_element**

- **rf\_definition\_element.get\_defined\_entity(): rf\_named\_entity**  
Returns the named entity that is being defined by this definition element. In other words, this element is part of the definition of the returned named entity.
- **rf\_definition\_element.get\_module(): rf\_module**  
Returns the module in which this definition element appears.
- **rf\_definition\_element.get\_source\_line\_no(): int**  
Returns the line number of the beginning of the clause in the source file.
- **rf\_definition\_element.is\_before(other: rf\_definition\_element): bool**  
Returns TRUE if this definition element appears before the one given as parameter in the load order. This is a full order relation on definition elements, which is defined as the ordinal number of modules and then the line number in the file (a definition element is not before itself).
- **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. Note that this method involves opening and reading the source file. If the source file is not found (according to the standard file search algorithm) an empty string is returned.
- **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 (see above). If the source file is not found (according to the standard file search algorithm) an empty list is returned.
- **rf\_named\_entity.get\_declaration(): rf\_definition\_element**

Returns the declaration (first definition element) of this entity. Note that every explicitly defined named entity has exactly one declaration. Types that are defined implicitly as variants of existing types do not have a declaration. In these cases NULL is returned (see next section for a list of such cases).

## 6.3.2 Type Layers

Enum types and struct types are extensible entities, and so their definition can consist of one or more layers. Others kinds of types are not extensible and so have only the declaring layer.

Not all types, however, have explicit definitions. Some types can be used in context, without being previously declared. Here is a list of the cases:

- Numeric types can be used in context with a size modification (such as ‘uint (bits: 16) ‘). The size modification implies a different type, but one which has no explicit declaration.
- Enum types can be spelled out inline, and have no separate declaration or explicit name.
- List types are instances of predefined parameterized type and so never have a definition in *e*.
- Not all *when* subtypes are explicitly defined, but they can still be used in context as types.

The first two are cases of 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—using ‘when’ or ‘extend’ constructs (see section 6.2.4.2). Even then the layers of the *when* subtypes are not always separable from those of other subtypes or of the *when base*. In order to reduce complexity, all these cases are treated in the same way from the point of view of aspect information. All types that fall under one of the above cases **do not have any layers**. Calling the method *get\_declaration()* on them would return NULL, and for implicitly defined enum, calling *get\_layers()* would return an empty list. As for structs, the service *get\_layers()* is restricted to *like* structs.

### 6.3.2.1 rf\_type\_layer

#### like rf\_definition\_element

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()* section 6.3.4.1).

### 6.3.2.2 rf\_enum\_layer

#### like rf\_type\_layer

- **rf\_enum\_layer.get\_added\_items(): list of rf\_enum\_item**  
Returns the named values that are added by this enum layer.
- **rf\_enum.get\_layers(): list of rf\_enum\_layer**  
Returns all enum layers that constitute this enum type.

### 6.3.2.3 rf\_struct\_layer

#### like rf\_type\_layer

- **rf\_struct\_layer.get\_field\_declarations(): list of rf\_definition\_element**  
Returns the field declarations that are added to the struct by this struct layer.
- **rf\_struct\_layer.get\_method\_layers(): list of rf\_method\_layer**  
Returns the method layers that are added to the struct by this struct layer.
- **rf\_struct\_layer.get\_event\_layers(): list of rf\_event\_layer**  
Returns the event layers that are added to the struct by this struct layer.
- **rf\_like\_struct.get\_layers(): list of rf\_struct\_layer**  
Returns all struct layers that constitute this struct type.

## 6.3.3 Method and Event Layers

In standard OO literature the term *method* is used to denote a certain behavior of an object. According to such understanding of the term a class inherits a method from its super class, or overrides it, that is, replaces it by a **different** method. Something, however, must stay fixed once a method is declared, and that is the common semantics of the operation it defines—the name, the parameter types, the return type, and some general intention of the operation. This fixed semantics is sometimes called a *message*. In *e* the term *method* is used in this sense.

A method in *e*, once declared for some struct, is never 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 it on different dog objects, so that they display different behavior.

The reason for this deviation from standard OO terminology is that in *e* one can modify the behavior not only in derived structs, but also in 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 apply to events too.

### 6.3.3.1 rf\_method\_layer

#### like rf\_definition\_element

- **rf\_method\_layer.get\_context\_layer(): rf\_struct\_layer**  
Returns the struct layer in which this method layer appears.
- **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 the extension mode by which this method layer was declared.
- **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.
- **rf\_method.get\_relevant\_layers(s: rf\_struct): list of rf\_method\_layer**  
Returns a list of all layers of this method that apply to the struct-type given as parameter. If the parameter struct-type does not have this method at all an empty list is returned. For example, method *to\_string()* is defined for every struct in *e*, so *get\_layers()* would return all extensions of this method in the system (including Specman internal). However calling *get\_relevant\_layer()* for struct *packet* would return only the extensions of *to\_string()* defined in the context of struct *packet* and its subtypes.

### 6.3.3.2 rf\_event\_layer

#### like rf\_definition\_element

- **rf\_event\_layer.get\_context\_layer(): rf\_struct\_layer**  
Returns the struct layer in which this event layer appears.
- **rf\_event\_layer.get\_extension\_mode(): rf\_extension\_mode**  
Returns either *is* or *only*, according to the extension mode by which this event layer was declared. Other extension modes such as *first* or *also* are not applicable to events.

- **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.

## 6.3.4 Modules and Packages

### 6.3.4.1 rf\_module

Modules are simply *e* files. However, with the ability to extend structs and in this way to separate different concerns or crosscuts of a system, they play an important role in organizing the program. If struct may be considered the vertical encapsulation principle then modules are the horizontal. 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. Modules are represented in reflection API and can be queried for their overall contribution to the structure of a system.

- **rf\_module.get\_name(): string**

Returns the name of this module—basically the name of the *e* file without the ‘.e’ extension.

- **rf\_module.get\_index(): int**

Returns this module's ordinal number in the load order (the first couple of hundred are Specman internal modules).

- **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.

- **rf\_module.get\_package(): rf\_package**

Returns the *e* package with which this module is associated. Note that modules that are not explicitly associated with some package (with *package* statement) are implicitly part of package *main*.

- **rf\_module.is\_user\_module(): bool**

Returns TRUE if the module is added by the user (all modules outside Specman are user modules).

- **rf\_module.is\_encrypted(): bool**

Returns TRUE if the module is encrypted.

### 6.3.4.2 rf\_package

A package is a set of one or more *e* modules that together implement some closely related functionality. It defines a scope to which the access of named entities can be restricted. It too is represented in the reflection API by a meta-object.

- **rf\_package.get\_name(): string**  
Returns the name of this package.
- **rf\_package.get\_modules(): list of rf\_module**  
Return the set of modules that are associated with this package.

### 6.3.5 Querying for Aspects

Analogous to the services that are the entry point in type information queries (see section 6.2.7) is the following set of services. It is the natural way to start aspect information queries.

- **rf\_manager.get\_module\_by\_name(name: string): rf\_module**  
Returns the module with the name given as parameter, or NULL if no module by this name is currently loaded.
- **rf\_manager.get\_module\_by\_index(index: int): rf\_module**  
Returns the module with the given ordinal number in the load order.
- **rf\_manager.get\_user\_modules(): list of rf\_module**  
Returns a list of all user modules that are currently loaded.
- **rf\_manager.get\_package\_by\_name(name: string): rf\_package**  
Returns the package with the name given as parameter, or NULL if no package by this name is currently loaded.

### 6.3.6 Use Example

The following code illustrates the way aspect information interface might be used. It is an implementation of a method that print out the content of modules in terms of the type layers that they add to the overall design, a set of modules that are loaded (compiled) on top of Specman.

```
print_user_modules() is {
    for each (m) in rf_manager.get_all_user_modules() do {
        outf("module - %s\n", m.get_name());
        for each (tl) in m.get_type_layers() do {
            if tl is a rf_struct_layer (sl) {
```

```
        outf("struct layer - %s\n",
            sl.get_defined_entity().get_name()
        );
        for each (fd) in sl.get_field_declarations() do {
            outf("\t\t field declaration - %s\n",
                fd.get_defined_entity().get_name()
            );
        };
        for each (ml) in sl.get_method_layers() do {
            outf("\t\t method layer - %s (%s)\n",
                ml.get_defined_entity().get_name(),
                ml.as_a(rf_method_layer).get_method_kind()
            );
        };
    };
};
};
};
```

Here is a possible output of this utility. In this case it runs on the trivial design from the previous example (see section 6.2.8), namely module ‘file1.e’ and ‘file2.e’.

```
Specman file2> print_user_modules()
module - file1
struct layer - packet
    field declaration - size
    field declaration - data
    method layer - foo (is)
struct layer - sys
    field declaration - packets
module - file2
struct layer - packet
    field declaration - corrupt
    method layer - foo (also)
    method layer - bar (is)
```

## 6.4 Value Query and Manipulation

The parts of the API described in previous chapters—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 that are 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.

This section covers the following topics:

- “Types of Objects” on page 6-25
- “Values and Value Holders” on page 6-26
- “Object Operators” on page 6-27
- “List Operators” on page 6-28
- “Use Example” on page 6-29

## 6.4.1 Types of Objects

The natural entry point for querying or manipulating objects is getting a representation of their type. Each object has exactly one most specific struct type that represents it, be it some *when subtype*, or just a regular *like struct*. One can ask for the like struct of an instance and disregard when variants, or for the specific *when* subtype. The *when* subtype of an instance depends on its state and may change with the course of the run.

- **rf\_manager.get\_struct\_of\_instance(instance: base\_struct): rf\_like\_struct**

Returns the most specific *like* struct of the struct instance given as parameter 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()*.

- **rf\_manager.get\_exact\_subtype\_of\_instance(instance: base\_struct): rf\_struct**

Returns the type of the instance given as parameter. The returned meta-object represents the most specific *significant* type that applies to the instance given as parameter, that is the one that contains all other types that apply to the instance. For example, if the parameter is a *packet*, which has the defined subtypes *big packet* and *corrupt packet*, and assume this packet happens to be both corrupt and big. In this case the returned type would be *corrupt big packet*, even though it is not a *defined* subtype. Note that the static type of a field, for example, can be in some cases 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 section 6.2.4.2).

## 6.4.2 Values and Value Holders

### 6.4.2.1 rf\_value\_holder

When dealing with values in a generic way (meta-programming), there must 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, 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.

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, must 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.

- **rf\_value\_holder.get\_type(): rf\_type**

Returns the type of this value. Note that in case the value is a struct instance, the type of the holder is not necessarily the most specific subtype of that instance (for example, a legal value holder whose type is *any\_struct* can hold an instance of *packet*).

- **rf\_value\_holder.get\_value(): untyped**

Enables (with the use of ‘unsafe()’ operator) assignment of the value into a typed variable. The variable should be of a type to which this value is assignable according to *e* cast rules. However, this cannot be enforced, and is the responsibility of the user. Note that the value should never be held by an untyped variable (this is what value holders are for).

- **rf\_type.create\_holder(value: untyped): rf\_value\_holder**

Returns a value holder of this type for the value given as parameter. The value must be an instance of this type (or of a subtype in case it is a struct type). Very simple sanity checks are being performed on the value, and if they fail an exception is thrown. These checks are by no means exhaustive, and it is the user’s responsibility to create the right holder for a value.

- **rf\_type.value\_is\_equal(val1: untyped, val2: untyped): bool**

Returns TRUE if the two instances given as operands are equivalent or identical (same semantics as that of the operator ‘==s’). The behavior is not defined if one of the two values is not of this type.

- **rf\_type.value\_to\_string(value: untyped): string**

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

### 6.4.3 Object Operators

The types of operators that are available for objects are the reading and writing to 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 previous section). However, using them involves the dynamic allocation of memory, and can therefore have impact on performance where this feature is heavily used. Thus, 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 just bare *untyped* values and skips checks. The brute version of each operator is marked as “unsafe”, and should be avoided where possible.

- **rf\_field.get\_value(instance: base\_struct): rf\_value\_holder**

Returns the value of this field in the struct instance given as parameter. If the struct type of the instance does not have this field an exception is thrown.

- **rf\_field.get\_value\_unsafe(instance: base\_struct): untyped**

Returns the value of this field in the struct instance given as parameter. If the struct type of the instance does not have this field the behavior is undefined.

- **rf\_field.set\_value(instance: base\_struct, value: rf\_value\_holder)**

Sets the value of this field in the struct instance given as parameter to the new value. If the struct type of the instance does not have this field an exception is thrown.

- **rf\_field.set\_value\_unsafe(instance: base\_struct, value: untyped)**

Sets the value of this field in the struct instance given as parameter to the new value. If the struct type of the instance does not have this field the behavior is undefined.

- **rf\_method.call(instance: base\_struct, parameters: list of rf\_value\_holder): rf\_value\_holder**

Calls this method on the struct instance given as parameter, with 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 in case of a mismatch in the number and types of parameters, an exception is thrown.

- **rf\_method.call\_unsafe(instance: base\_struct, parameters: list of untyped): untyped**

Calls this method on the struct instance given as parameter, 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 in case of a mismatch in the number and types of parameters, the behavior is undefined.

- **rf\_event.is\_emitted(instance: base\_struct): bool**

Returns TRUE if this event of the instance given as parameter was emitted so far in the current cycle. If the struct type of the instance does not have this event an exception is thrown.

- **rf\_event.is\_emitted\_unsafe(instance: base\_struct): bool**

Returns TRUE if this event of the instance given as parameter was emitted so far in the current cycle. If the struct type of the instance does not have this event the behavior is undefined.

- **rf\_event.emit(instance: base\_struct)**

Emits the event on the instance given as parameter. If the struct type of the instance does not have this event an exception is thrown.

- **rf\_event.emit\_unsafe(instance: base\_struct)**

Emits the event on the instance given as parameter. If the struct type of the instance does not have this event the behavior is undefined.

## 6.4.4 List Operators

The three main list operators—reading an element, writing to an index, and querying the size—are available as general services, that is, methods of `rf_manager`. Here, again, two versions of the operators are available—the safe version, using value holders, and the brute one, using untyped values.

- **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 bound an exception is thrown.

- **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 bound the behavior is undefined.

- **rf\_manager.set\_list\_element(list: rf\_value\_holder, index: int, new\_val: rf\_value\_holder)**

Sets the value of the given list at the given index. If the first parameter is not a list, the index is out of bound or the new value is not of an instance of the list's element type an exception is thrown.

- **rf\_manager.set\_list\_element\_unsafe(list: untyped, index: int, new\_val: untyped)**

Sets the value of the given list at the given index. If the value is not a list, the index is out of bound or the new value is not of an instance of the list's element type the behavior is undefined.

- **rf\_manager.get\_list\_size(list: rf\_value\_holder): int**

Returns the number of elements currently in the given list. If the value is not a list an exception is thrown.

- **rf\_manager.get\_list\_size\_unsafe(list: untyped): int**

Returns the number of elements currently in the given list. If the value is not a list the behavior is undefined.

## 6.4.5 Use Example

It is hard to find an intuitive use for the object query interface, which is simple enough to serve as an example. The following code is a very simple utility that prints out the state of objects recursively. It is somewhat artificial since it prints out only enumerated and boolean fields, other scalar types being more complicated to handle.

```
print_struct_recursive(obj: any_struct) is {
    var s: rf_struct = rf_manager.get_struct_of_instance(obj);
    outf("instance of %s\n", s.get_name());
    for each (f) in s.get_fields() {
        if f.get_declaration().get_module().is_user_module() {
            outf("field %s - ", f.get_name());
            var vh: rf_value_holder = f.get_value(obj);
            if vh.get_type() is a rf_scalar (e) {
                outf("%s", vh.get_type()).
value_to_string(vh.get_value().unsafe());
            } else if vh.get_type() is a rf_struct (s) {
                print_struct_recursive(vh.get_value().unsafe());
            } else if vh.get_type() is a rf_list (l) and
l.get_element_type() is a rf_struct {
                outf("\n");
                var size: int = rf_manager.get_list_size(vh.get_value());
                for i from 0 to size-1 do {
                    outf("%d: ", i);
                    print_struct_recursive(rf_manager.
get_list_element(vh, i).get_value().unsafe());
                };
            };
            outf("\n");
        };
    };
};
```

Here is the result of calling this method, given the little design of the previous examples (see section 2.8—the design consisting of ‘file1.e’ and ‘file2.e’).

```
Specman file2> gen -seed = 5
Doing setup ...
Generating the test using seed 5...
Specman file2> print_struct_recursive(sys)
instance of sys
field packets -
0: instance of packet
field size - small
field data - -5319061515555392341
field corrupt - TRUE
```

```
1: instance of packet
field size - small
field data - 3230878320328792872
field corrupt - FALSE
2: instance of packet
field size - medium
field data - 2775930122720983980
field corrupt - TRUE
3: instance of packet
field size - big
field data - 2044827916054152830
field corrupt - FALSE
```