

Annex C 1

(normative) 5

Name spaces 5

This annex defines an extension to the encapsulation scheme — the scoping of names. The naming problem in *e* is addressed by using packages as *name spaces* for types. This overloading of the notion of a package is natural and common in other languages. 10

C.1 The naming problem in *e* 15

Currently, all types and structs in *e* share one global name space, in which every name needs to be unique. This problem is more acute in *e* than in other programming languages, because there are no separate binary libraries or object files. As verification projects grow in size and depend more on components developed externally, the risk of name collisions increases. Such name clashes can be hard to work around, especially when they occur during integration of code from different parties. 20

Fields, methods, and events shall have a unique name in the context of their *structs*, including those added in distant extensions of the *struct*. Similarly, **enum** items shall have a unique identifier within a single **enum** type, even when they are added to later extensions of it. Name collisions might occur also for these entities, although they are much less likely. 25

In *e*, this problem is addressed by using a naming convention that assigns a globally unique name to every type and a unique name to every *struct* member or enum item in extensions outside of the declaring package. The unique name consists of the simple name of the entity prefixed by the package name, e.g., the unit that represents a monitor in the `vr_xbus` package is named `vr_xbus_monitor_u`. 30

The problem with this convention is most names become very long; so the code can become cumbersome and unreadable. The readability of the output can also be impacted. Another problem is the convention is enforced by an *e* compiler; thus, it becomes the developer's responsibility not to pollute the global name space with unqualified names. 35

C.2 Resolution overview 40

This material previews the resolution for using name spaces in *e*.

C.2.1 Packages as name spaces 45

Packages, being the major encapsulation vehicle in *e*, are the natural candidates for serving as name spaces. As in the *e* naming convention, package names serve to qualify type names declared within their context. But, unlike the *e* naming convention, this solution involves full linguistic support for name qualification. The support consists of syntactic difference between qualified and unqualified type names, and semantic rules for resolving the reference of unqualified names. 50

C.2.2 Name spaces for other named entities 55

According to *e*, fields, methods, events, and **enum** items shall qualify their names with the package name only when declared in extensions outside the package where their context type is declared. These cases are

1 relatively rare, so keeping to the convention does not affect the readability of the code significantly.
 Furthermore, even when the convention is not strictly kept, the probability of a name collision is low. On the
 other hand, having packages serve as name spaces for *struct* members or **enum** items is unintuitive and hard
 to define. Therefore, the name space scheme described here is restricted to types only.

5 **C.2.3 Name resolution**

10 Types can be referenced by their given name or a qualified name. Referencing a type by a qualified name
 succeeds given a type by that name exists in the right package (in code that is already loaded). As is required
 by the compatibility constraint, unqualified references succeed also if only one type by that name exists in
 some package. In the case of unqualified references where more than one type by the same name exists in
 15 different packages, the compiler tries to resolve the reference according to set priorities. A type declared
 within the context package has the highest priority. Public types declared in packages used by the context
 package are next in priority. Only then come all the rest, i.e., types in packages not used by the context
 package. See also: C.3.

20 **C.2.4 The use relation**

import statements are used to declare dependencies between different parts of a design; they determine if a
 package uses another package. A package uses another package when one of its modules directly imports
 one of the other package's modules. Another important consequence of this definition is the use relation is
 25 defined in terms of packages, which puts more weight on the package as a whole. This goes together well
 with the idea of using packages as encapsulation units and name spaces as encapsulation mechanisms. The
 down side of this is every inter-package **import** statement affects the name resolutions of the whole package.
 This may force qualification of names in the modules, which is unnecessary from the module's perspective.
 For more details, see C.4.

30 **C.2.5 Reserved type names**

One set of types in *e* is considered essential or “core”, e.g., **int**, **string**, and **sys**. No reasonable *e* program
 would assign the name of any of those types to a user-defined type. Core types are declared under a special
 35 package called **e_core**. Within the name space model, **e_core** type names are reserved. Unlike other type
 names, they cannot be used in another package. For more information, see C.3.3.

40 **C.3 Qualified and unqualified names**

Types (scalars or *structs*) declared inside a package belong to the name space of that package. Type names
 shall be unique in the context of the package, but types in different packages can have the same name.
 Fields, methods, and events shall have unique names within their context *struct* even if they are declared in
 45 extensions in different packages. The same goes for **enum** items in the context of the **enum** type. Many other
 built-in derivatives of explicitly defined types, such as lists and size-modified scalars are available to a
 program. These are not associated directly with a package, but rather through their explicit base type.

50 **C.3.1 Name rules**

- An explicitly declared type can be referenced using an *unqualified name* (an *e* identifier) or a *quali-
 fied name* in the form:

```
id :: id
```

The second identifier is the type's given name, and the first is the package in which it was declared.
 The double colon (::) is called the *scope operator*.

NOTE—The scope operator does not relate directly to built-in derivatives of a type. Rather, it relates to the explicitly declared type that serves as the base for the derivatives. For example, the qualified name of `list of packet` would be `list of vr_xbus::packet` and not `vr_xbus::list of packet`. Similarly, in the case of **when** qualifiers, `big corrupt packet` would be `big corrupt vr_xbus::packet`.

- Only a simple name can be used in the declaration of a new type (a **type**, **struct**, or **unit** statement).
- In any other construct where type names matter, both qualified and unqualified names are syntactically legal.
- Whether the module is associated explicitly with a package (by a **package** statement) or implicitly with package **main** (if there is no **package** statement) is irrelevant to naming, i.e., types declared in modules that do not explicitly associate themselves with some package are part of the name space of package **main**.
- The names of types declared in the package **e_core** are reserved and cannot be used in the declaration of new types anywhere.

C.3.2 Type reference resolution

- A qualified name always fully determines the reference. If there is no type by the given name in the given package, a *type not found* error is issued.
- The reference of unqualified names is determined on the basis of the following three priorities (listed in order of priority):
 - 1) A type declared within the context package
 - 2) A public type declared in a package that is used by the context package (see C.4)
 - 3) A public type declared in packages outside of the current context (priorities 1 and 2)
- If more than one public type with the same name and priority is found during the search process (this is only possible for priorities 2 and 3), then an ambiguity error is issued. In so, the user needs to resolve the ambiguity by qualifying the type name.

NOTE—Only actual ambiguities are reported, i.e., ambiguities found during resolution of an actual reference. There is no problem with potential ambiguities.

C.3.3 e_core types

The set of types declared in the built-in package **e_core** is privileged. Unlike all other types, the names of the **e_core** types remain unique. When a user tries to define a type using the name of an **e_core** type, an error is reported. However, these names are not reserved keywords; they can be used as identifiers in any other context (variable names, method names, and so on).

The **e_core** types are: **int**, **uint**, **byte**, **bit**, **bool**, **string**, **sys**, **global**, **base_struct**, **any_struct**, **any_unit**, and **event_port**.

C.4 Use relation

An **import** statement declares the dependency of one module on another. These dependencies determine the load order of modules in a single **load command or compilation** (see Annex A). As part of the name space scheme, a use relation between packages is defined in terms of *module dependency*. A package uses another if any of its modules directly imports any of the other's modules. As explained in the C.3.2, packages used by the current module's package are second in the search list for type resolution, after the context package itself, but before the rest of the packages.

1 C.4.1 Load clusters

5 The definition of the use relation requires some amplification. Packages are not necessarily loaded all at once, new modules of already known packages can be loaded at any stage. So not all modules of a package can be taken into account when determining the used packages of the currently loading package at some given stage.

10 On the other hand, it is not enough to take into account only modules that have already been loaded. A typical case would be when the top file of package *A* imports the top file of package *B* and then imports the rest of package *A*'s files. Package *A*'s files are actually loaded before the top file is, but would naturally presuppose package *B* as being used already in their context. For this reason the concept of a load cluster is needed.

15 A *load cluster* is the set of modules that are loaded by a single **load** command (or a single compilation). All modules that together form a load cluster depend on a common root in the dependency graph — the module explicitly mentioned in the **load** command. Consequently, package *A* uses package *B* at a given stage in the load process if there is a module *m* that is either already loaded or is part of the current load cluster and *m* directly imports some module of package *B*.

NOTE 1—A module that is imported by a module in the current load cluster might be already loaded by previous **load** commands. Whether or not it is already loaded does not affect the use relation.

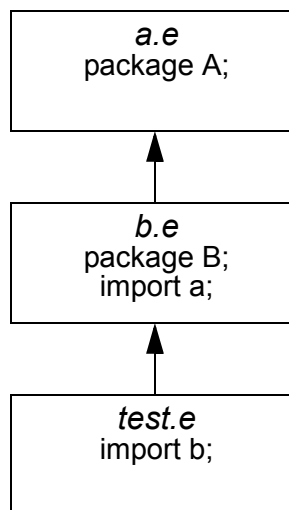
25 NOTE 2—If more than one module is named by a **load** command or a single compilation (e.g., `load a.e, b.e`), the two modules with all of their dependencies are treated as a single load cluster.

30 C.4.2 Examples

These examples demonstrate various use relationships.

35 C.4.2.1 Example 1

This example illustrates a simple use relation, as shown in Figure C.1.



40
45
50
55
Figure C.1—Simple use relation

Since module *test.e* is the one explicitly loaded, and, thus is the root of the load cluster, package *B* uses package *A* and package *main* (the package of module *test.e*) uses package *B*. Type references would be resolved accordingly.

C.4.2.2 Example 2

This example (see Figure C.2) shows how the use relation takes into account the whole load cluster.

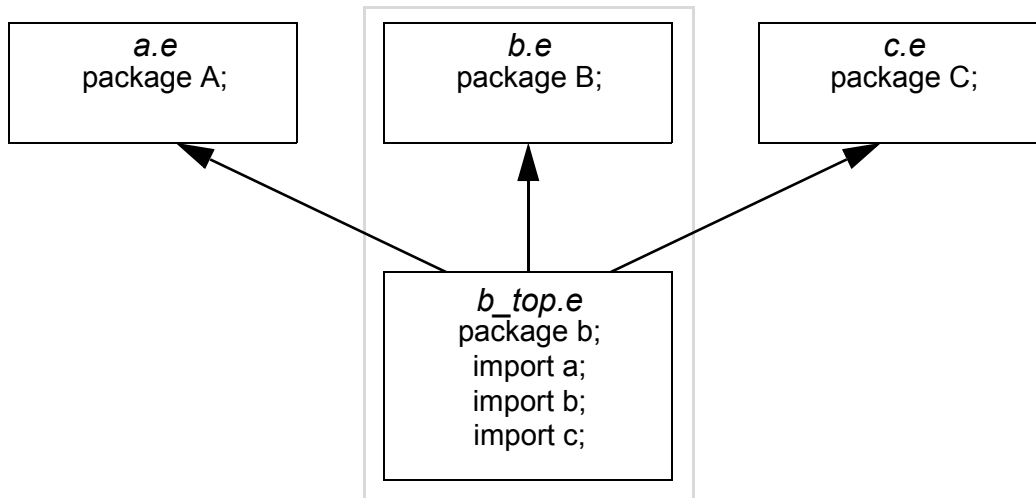


Figure C.2—Use relation for a load cluster

Since module *b_top.e* is the root of the load cluster, package *B* uses both package *A* and package *C*. Thus, in resolving type names in module *b.e*, both *A* and *C* are name spaces with priority, even though this module is loaded before module *top_b.e*.

NOTE — Types (like any other named entities) declared in *c.e* could not be used in *b.e*, because they are still not loaded. Of the named entities declared within package *C*, only those in modules that are previously loaded can be referenced at all, that is, with or without qualification.

C.4.2.3 Example 3

This example (see Figure C.3) shows how previous **load** commands affect the use relation in the current load cluster, but not vice-versa.

1

5

10

15

20

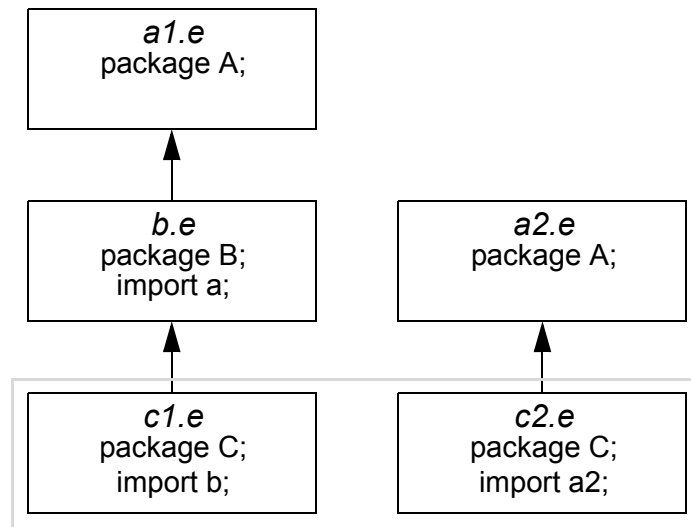


Figure C.3—Use relation dependencies

25

Module *c1.e* is explicitly loaded by a **load** command and module *c2.e* is loaded by a subsequent command. In this case, when *c1.e* is loaded, package *C* is only using package *B* and names in *B* shadow names in *A* in the context of *c1.e*. When module *c2.e* is loaded, both package *A* and *B* are used by package *C*; therefore, a type name declared in both with the same name would be ambiguous in the context of *c2.e*.

30

C.5 Built-in APIs

This section defines how name spaces can impact (using) APIs.

35

C.5.1 Reflection and name spaces

The reflection API deals with type names in two ways: getting a type with a given name and getting the name for a given type. Both need to address name spaces.

40

rf_manager.get_type_by_name() can be used to get a representation of a type with a given name. If the parameter uses the scope operator, the qualified name's type is returned (or NULL if no such type exists). If the parameter is a simple name, the requested type is determined similar to the resolution process for interactive scope described in C.3.2. If more than one type with the same unqualified name exists with the same priority, NULL is returned.

45

The interactive context package determines which types have priority for resolution. Therefore, meta-programming code using reflection might behave differently when the context package changes. The context for name resolution has nothing to do with the module in which the call to **get_type_by_name()** actually appears, but only with the context package when the call is executed.

50

The method **rf_type.get_name()** continues returning the type's unqualified name. A **new method** called **rf_type.get_qualified_name()** returns a string with the fully qualified name (in `::` format).

55

See also: xref the new [Reflection clause](#) as appropriate (both ways).

C.5.2 Coverage callback support

As a part of the coverage API, use of type names is made programmatically as user input (as parameters to methods) or as output (a field set by the service).

Methods that take type names (possibly with wildcards) as parameters are **user_cover_struct.scan_cover()**, **covers.set_weight()**, and **covers.set_at_least()**. The same rules for commands in interactive scope explained in C.3.2 hold here. When a type name without wildcards is used, the usual resolution process of using the priorities with respect to the interactive context package applies. When a wildcard is used, the reference is taken to be all types whose name matches the pattern, regardless of the package qualifier.

The field **user_cover_struct.struct_name** can be used to make a type name an output. For compatibility, this field can only be a simple name. A **new field** of **user_cover_struct** called **package_name** holds the name of the package in which the output type was declared.

See also: xref the [Coverage clause](#) (Clause 14) as appropriate (both ways).

C.6 Code comparison

This section uses example code for three *e* verification components: `vr_xbus`, `vr_xserial`, and `vr_xsoc` to compare how typical code looks with the *e* naming conventions and how it would be written with name space support (note the **highlighted** differences shown in Table C1). The only places where qualified names are required by the compiler are places where they are also needed for the understandability of the code (e.g., the scoping operator is needed in the `vr_xsoc_* .e` example).

Table C1—Comparing code with and without name spaces

Typical <i>e</i> code	Modified to use name spaces
<pre> vr_xbus_*.e package vr_xbus; type vr_xbus_env_name_t: []; unit vr_xbus_env_u { name: vr_xbus_env_name_t; monitor: vr_xbus_monitor_u}; unit vr_xbus_monitor_u {...}; </pre>	<pre> vr_xbus_*.e package vr_xbus; type env_name_t: []; unit env_u { name: env_name_t; monitor: monitor_u}; unit monitor_u {...}; </pre>
<pre> vr_xserial_*.e package vr_xserial; type vr_xserial_env_name_t: []; unit vr_xserial_env_u { name: vr_xserial_env_name_t; agent: vr_xserial_agent_u is instance;}; unit vr_xserial_agent_u {...}; </pre>	<pre> vr_xserial_*.e package vr_xserial; type env_name_t: []; unit env_u { name: env_name_t; agent: agent_u is instance;}; unit agent_u {...}; </pre>
<pre> vr_xsoc_*.e package vr_xsoc; type vr_xsoc_env_name_t: []; unit vr_xsoc_env_u { name: vr_xsoc_env_name_t; xbus_evc: vr_xbus_env_u; xserial_A_evc: vr_serial_env_u; xserial_B_evc: vr_serial_env_u}; </pre>	<pre> vr_xsoc_*.e package vr_xsoc; type env_name_t: []; unit env_u { name: env_name_t; xbus_evc: vr_xbus::env_u; // No qualification would result // in an ambiguity error xserial_A_evc: vr_serial::env_u; xserial_B_evc: vr_serial::env_u}; </pre>