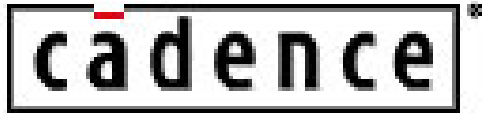


Namespaces for Types in e

Version 1.1

© 2006 Cadence Design Systems, Inc. All rights reserved worldwide

Contents

1 Introduction . . . 1

1.1 This Document . . . 1
1.2 The Naming Problem in ***e*** . . . 1
1.3 Considerations . . . 2
1.4 Namespace Solution Overview . . . 2

2 Qualified and Unqualified Names . . . 4

2.1 Names Overview . . . 4
2.2 Name Rules . . . 4
2.3 Type Reference Resolution . . . 5
2.4 e_core Types . . . 5
2.5 Code Comparison . . . 5
2.5.1 Example Code Adhering to ***e***RM Naming Conventions . . . 6
2.5.2 Example Code with Namespace Support . . . 7

3 The Use Relation . . . 8

3.1 Use Relation Definition . . . 8
3.2 Load Clusters . . . 8
3.3 Examples . . . 9

4 Non-Load-Time (NLT) Type Names . . . 11

4.1 NLT Type Names Overview . . . 11
4.2 NLT Type Name Rules . . . 12
4.3 Interactive Scope Commands . . . 12

© 2006 Cadence Design Systems, Inc. All rights reserved worldwide

4.4 Printing and Data Presentation 13
4.5 Debugger Scope 13

5 Built-in APIs 13

5.1 Reflection and Namespaces 13
5.2 Coverage Callback Support 14
5.3 C Interface Support 14

6 Compatibility 15

6.1 Compatibility Overview 15
6.2 Strict Compatibility 16
6.3 Output Compatibility 16
6.4 ***e***RM Compatibility 16

© 2006 Cadence Design Systems, Inc. All rights reserved worldwide

1 Introduction

1.1 This Document

This document defines an extension to the encapsulation scheme described in the document named *Support for Encapsulation in e*, version 1.1. The enhancement of ***e*** described in that document involved introducing the notion of packages as access scopes into the language. You can make types, structs, fields, methods and events inaccessible outside the package. The enhancement to the language described in this document addresses another aspect of encapsulation — scoping of names. The naming problem in ***e*** is addressed by using packages as namespaces for types. This overloading of the notion of a package is natural and common in other languages.

1.2 The Naming Problem in *e*

Currently, all types and structs in ***e*** share one global namespace, in which every name must 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.

Fields, methods, and events must have a unique name in the context of their structs, including those added in distant extensions of the struct. Similarly enum items must 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.

In ***e***RM, this problem is addressed by 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. For example, the unit that represents a monitor in the vr_xbus package must be named vr_xbus_monitor_u.

© 2006 Cadence Design Systems, Inc. All rights reserved worldwide

The problem with this convention is that most names become very long, and so code can become cumbersome and unreadable. Where type names figure in log files and the debugging environment, the readability of the output is also hurt. Another problem is the fact that the convention is enforced by an ***e*** compiler. It is the developer's responsibility not to pollute the global namespace with unqualified names, and this responsibility is a burden.

1.3 Considerations

This section explains the major considerations that underlie the solution presented in this document.

Compatibility

Packages were already introduced without affecting naming. Existing code may refer to entities using their given name without any limitation, and specifically with no regard to the package in which they were declared. All legal code must remain legal with this enhancement. This requirement implies that wherever a name is used and the reference is unique (as it necessarily has been before namespaces were introduced), the resolution of the reference must be carried automatically. Chapter 6 "Compatibility" discusses the definition of namespaces from this perspective.

Simplicity

The solution should employ as few new concepts and as few new linguistic constructs as possible. The semantic rules that govern the new syntax should also be few and simple.

Usability

The linguistic support for namespaces should take care of the bottleneck of names in a way that is intuitive and easy to handle by the average user, while not limiting advanced or more esoteric uses.

1.4 Namespace Solution Overview

Packages as Namespaces

Packages, being the major encapsulation vehicle in ***e***, are the natural candidates for serving as namespaces. As in the ***e***RM naming convention, package names serve to qualify type names declared within their context. But unlike the ***e***RM 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.

© 2006 Cadence Design Systems, Inc. All rights reserved worldwide

Namespaces for Other Named Entities

According to ***e***RM, fields, methods, events and enum-items must qualify their names with the package name only when declared in extensions outside the package in which their context type is declared. These cases are relatively rare, and 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 namespaces for struct members or enum-items is unintuitive and hard to define. Therefore the namespace scheme described here is restricted to types only.

Name Resolution

Types can be referenced by their given name, or by a qualified name. Referencing a type by a qualified name succeeds given that 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 too if only one type by that name exists in some package. In case of unqualified references where more than one type by the same name exists in 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 that are used by the context package are next in priority. Only then come all the rest, that is, types in packages not used by the context package. This subject is elaborated in Chapter 2 “Qualified and Unqualified Names”.

The Use Relation

Whether or not a package uses another package is determined by **import** statements. A package uses another package if one of its modules directly imports one of the other package’s modules. Import statements in general are used to declare dependencies between different parts of a design. The fact that import statements are overloaded with this extra semantics is consistent with the general intention. Another important consequence of this definition is the fact that the use relation is defined in terms of packages, which puts more weight on the package as a whole. This goes together well with the idea of packages as encapsulation units, and namespaces as encapsulation mechanisms. The down side of this is that every inter-package import statement affects the name resolutions of the whole package. It may force qualification of names in modules, which is unnecessary from the module’s perspective. For details, see Chapter 3 “The Use Relation”.

Reserved Type Names

One set of types in ***e*** is considered essential or “core”. No reasonable ***e*** program would assign the name of any of those types to a user-defined type. Examples of these core types are **int**, **string**, and **sys**. These types are declared under a special package called **e_core**. With the namespace model, **e_core** type names are reserved. Unlike all other type names, they cannot be used in another package. For more information, see “e_core Types” on page 5.

© 2006 Cadence Design Systems, Inc. All rights reserved worldwide

Non-Load-Time (NLT) Type Names

Namespaces are designed to improve the convenience of interactive sessions with Specman Elite and to improve readability of ***e*** command scripts and log files. Name resolution for NLT ***e*** (type names that are not resolved when loading or compiling ***e*** modules, for example, code in an .ecom file or at the Specman> prompt) is similar to that of code in ***e*** modules. The context package for resolving type names is, in most cases, set manually by users. In this way, users get the look and feel of code inside ***e*** modules of interest. For details, see Chapter 4 “Non-Load-Time (NLT) Type Names”.

2 Qualified and Unqualified Names

2.1 Names Overview

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

2.2 Name Rules

- An explicitly declared type can be referenced using an *unqualified name* (an ***e*** identifier) or a *qualified 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”.

- In the declaration of a new type (**type**, **struct**, or **unit** statement), only a simple name can figure.

- In any other construct where type names figure, both qualified and unqualified names are syntactically legal.

© 2006 Cadence Design Systems, Inc. All rights reserved worldwide

- It is irrelevant to naming whether the module is associated explicitly with a package (by a **package** statement) or implicitly with package **main** (if there is no **package** statement). In other words, types declared in modules that do not explicitly associate themselves with some package are part of the namespace of package **main**.
- The names of types declared in package **e_core** are reserved and cannot be used in the declaration of new types anywhere.

2.3 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 Chapter 3 "The Use Relation")
  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 is found with the same priority during the search process (possible only in priority 2 and 3), then an ambiguity error is issued. In that case, users must resolve the ambiguity by qualifying the type name. (Only actual ambiguities are reported, that is, ambiguities found during resolution of an actual reference. There is no problem with potential ambiguities.)

2.4 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 with a name identical to an **e_core** type, an error is reported. However, these names are not reserved keywords. They can, at least as far as this aspect of the language goes, 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**.

2.5 Code Comparison

This section takes an example from the ***e***RM golden ***e***VCs and compares such typical code looks with the ***e***RM naming conventions and how it would be written with namespace support. In the example, there are three ***e***VCs — vr_xsoc, vr_xbus, and vr_xserial.

© 2006 Cadence Design Systems, Inc. All rights reserved worldwide

2.5.1 Example Code Adhering to *e*RM Naming Conventions

In 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 {...};
```

In 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 {...};
```

In 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;
};
```


© 2006 Cadence Design Systems, Inc. All rights reserved worldwide

2.5.2 Example Code with Namespace Support

With namespace support, the following code would be the equivalent of the code found in "Example Code Adhering to eRM Naming Conventions". Note that the only places where qualified names are required by the compiler are places where they are also needed for the understandability of the code.

In vr_xbus_*.e

```
package vr_xbus;

type env_name_t: [];

unit env_u {
    name: env_name_t;
    monitor: monitor_u;
};

unit monitor_u {...};
```

In 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 {...};
```

In 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;
};
```


© 2006 Cadence Design Systems, Inc. All rights reserved worldwide

3 The Use Relation

3.1 Use Relation Definition

Dependency of one module on another is declared by an **import** statement. Dependencies determine in turn the load order of modules in a single **load** command or compilation. As part of the namespace 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 was explained in the "Type Reference Resolution" on page 5, 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.

3.2 Load Clusters

The definition of the use relation in "Use Relation Definition" on page 8 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 in figuring out the used packages of the currently loading package at some given stage.

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.

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 directly or indirectly on a common root in the dependency graph — the module that is explicitly mentioned in the **load** command.

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*.

Notes

- 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.
- If more than one module is named by a **load** command or a single compilation (for example, "load a.e, b.e"), the two modules with all of their dependencies are treated as a single load cluster.

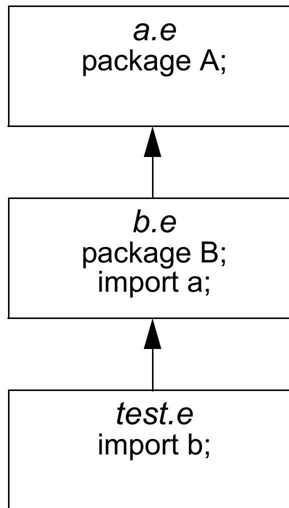
© 2006 Cadence Design Systems, Inc. All rights reserved worldwide

3.3 Examples

Example 1

This example illustrates a simple use relation.

Figure 1 Simple Use Relation



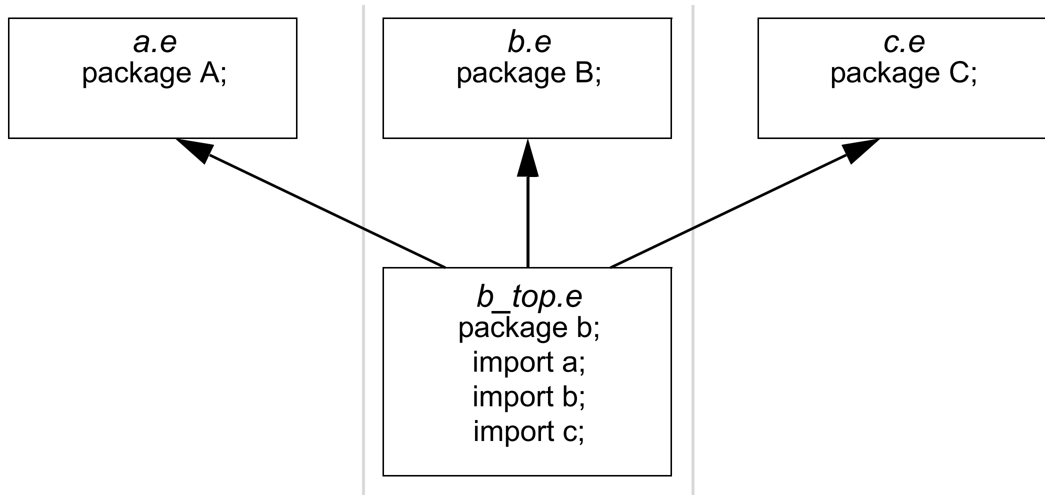
Given that module *test.e* is the one explicitly loaded, and is thus the root of the load cluster, package *B* uses package *A*, and package *main* (the package of module *test.e*) uses package *B*. Type references would be resolved accordingly.

© 2006 Cadence Design Systems, Inc. All rights reserved worldwide

Example 2

This example shows how the use relation takes into account the whole load cluster.

Figure 2 Use Relation for Load Cluster



Given that 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 namespaces with priority, even though this module is loaded before module *top_b.e*.

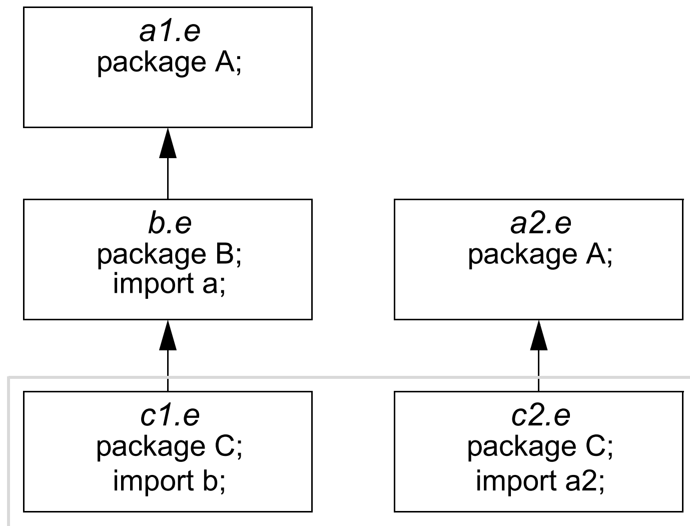
Note Types (like any other named entities) declared in *c.e* could not be used in *b.e*, because they are still not loaded. Of the named entities declared within package *C*, only those in modules that are previously loaded can be referenced at all, that is, with or without qualification.

© 2006 Cadence Design Systems, Inc. All rights reserved worldwide

Example 3

This shows how previous **load** commands affect the use relation in the current load cluster but not vice versa.

Figure 3 Use Relation Dependency



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*, and therefore a type name that is declared in both with the same name would be ambiguous in the context of *c2.e*.

4 Non-Load-Time (NLT) Type Names

4.1 NLT Type Names Overview

So far the focus was on code within ***e*** modules. However, Specman Elite includes development-environment tools (debugger, data browser, etc.), and a rich interactive language. It involves type names that users input, and type names that are output. In both cases names can be qualified or unqualified. Whenever unqualified type names are used, the reference must be fixed by rules. For the most part, the rules are the same as before.

© 2006 Cadence Design Systems, Inc. All rights reserved worldwide

4.2 NLT Type Name Rules

- Whenever user input involves type names, qualified or unqualified names are allowed.
- There is always a context package for the evaluation of NLT type references, even though the **package** statement is not part of the command language. The package can be thought of as the current focus of the interactive session.
- The resolution of unqualified names is done in exactly the same way as in ***e*** modules (see “Type Reference Resolution” on page 5). Considered first are types declared in the context package, then types in packages used by it, and only then the rest.
- By default the context package is **main** (just like in ***e*** modules).
- The context package can be set manually by a new command. The syntax is:

```
set package package-name
```

4.3 Interactive Scope Commands

Many commands involve reference to types. Examples are **show cover**, **echo event**, and **trace ports**. Type names in all of these cases may involve wildcards. Normal type names, that is, names without wildcards, are subject to the regular resolution rules explained in “NLT Type Name Rules” on page 12. An error is issued when no type by the specified name exists, or when the name is ambiguous given the context packages. The reference of the name in the non-wildcard case is evaluated at the moment the command is entered.

Types names can have wildcards. The asterisk (*) stands for any number of non-blank characters. When wildcards are used, the input does not stand for a specific type anymore, but rather for a pattern of type names. In this case, no resolution rules are taken into consideration. Unless a scope operator appears explicitly in the pattern, simple names of types from all packages are considered. In addition, the applicability of the pattern is determined only when the command is actually executed during the test. For example, if the type argument to a **trace ports** command contains wildcards, matching ports are traced even if their declaration is loaded after the command is entered. The same goes for other variations of **trace**, as well as **echo** and **break** commands.

For example, the command “echo event monitor_u” will only consider events on *vr_xsoc::monitor_u*, given that the context package is *vr_xsoc*. However, with the command “echo events monitor_*”, events on both *vr_xsoc::monitor_u* and *vr_xbus::monitor_u* would be considered, regardless of the context package, because the expression “monitor_*” is a global pattern of type names. You can restrict the wildcard with a scope operator, as in “echo event vr_xbus::*”. It should be noted that wildcards can be used anywhere within the type name, including the package qualifier and **when** subtype qualifiers, as in “*’size vr_*::packet_s”.

© 2006 Cadence Design Systems, Inc. All rights reserved worldwide

4.4 Printing and Data Presentation

Print commands on struct instances and calls on **any_struct.do_print()** also output the instance's type name. It is either qualified or unqualified. There are three related printing modes configurable by the **set config** command or through the standard dialog box. The options for printing qualified names are: **always**, **never**, and **upon_ambiguity**. The default is **upon_ambiguity** (printing qualified type names upon ambiguity).

Ambiguity for output (the third printing mode) is considered globally. A qualified name is printed for a type whenever there is another type by the same name declared anywhere. This is a more restrictive policy than the one imposed on input, as it does not consider the context package. The reason for this is that during a single interactive session, the context package might change, and printing/logging should not depend on such contingencies.

The same policy applies across all Specman Elite built-in development tools, like the data-browser and the debugger. It also applies to the printed struct format (the *type-name@number* notation). Printed structs can also be referred to with the same *@number* notation. Such expression can be prefixed by a type name (just as it appears in the output), and this type name can be qualified or unqualified. When unqualified, the type name in the expression is subject to the regular resolution rules.

4.5 Debugger Scope

When stopped on a breakpoint in the debugger, the context package is automatically set to the package of the module in the debugger's frame. The resolution process of unqualified type names is then identical to the one actually used to analyze the source code of the frame, so that constructs with type names can be copied from the code and pasted to the interpreter prompt.

5 Built-in APIs

5.1 Reflection and Namespaces

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 must address namespaces.

You can get a representation of a type with a given name by calling **rf_manager. get_type_by_name()**. If the parameter is a qualified name (with double-colon "::" operator), the type by that name would be returned or NULL if no such type exists. If the parameter is a simple name, the requested type would be figured out in a way similar to the resolution process for interactive scope described in "NLT Type Name Rules" on page 12. If more than one type with the same unqualified name exists with the same priority, NULL is returned, just like when no type by that name exists.

© 2006 Cadence Design Systems, Inc. All rights reserved worldwide

The interactive context package determines which types have priority for resolution. Therefore, metaprogramming 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.

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 the double-colon "::" format).

5.2 Coverage Callback Support

As a part of the coverage API, use of type names is made programmatically either 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 "Interactive Scope Commands" on page 12 hold here. When a type name without wildcards is given, the usual resolution process according to the priorities with respect to the interactive context package applies. When a wildcard is given, the reference is taken to be all types whose name matches the pattern, regardless of the package qualifier.

A type name is the output of the service through field **user_cover_struct.struct_name**. For compatibility considerations, this field will continue to hold only simple names. A new field of **user_cover_struct** called **package_name** holds the name of the package in which the output type was declared.

5.3 C Interface Support

C code can make use of ***e*** types using the C interface. Such use is made by C macros that take the name of the type in ***e*** as an argument. Some examples are declaring variables of ***e*** types (using the macro SN_TYPE or SN_LIST) and calling a method on an ***e*** object (using the macro SN_DISPATCH). Types that are used in this way must be explicitly exported in the ***e*** code with the **C export** statement.

Currently, all C interface macros that take type name as argument are used with ***e*** simple names (C identifiers). With the introduction of namespaces into the language, the same type can also be referenced in C with the qualified name, using the C macro:

```
SN_QUALIFIED_NAME(package-name,simple-type-name)
```

in place of the simple identifier where the name would go previously. So, for example, a variable of the type named vr_xbus::monitor_u could be declared in C in this way:

```
SN_TYPE(SN_QUALIFIED_NAME(vr_xbus,monitor_u)) my_monitor;
```


© 2006 Cadence Design Systems, Inc. All rights reserved worldwide

The **C export** statement determines the way the type is exported. When a simple name is used for exporting, both qualified and unqualified references can be made in C. When a qualified name is used in the **C export** statement, only a qualified name can be used in C.

In C, just like in ***e***, using qualified names is always possible, but it is required only when there is actual ambiguity. However, unlike ***e***, there is only one resolution rule throughout the whole C compilation unit. Therefore, ambiguity is immediately identified by the ***e*** compiler when two **C export** statements mention two different types with the same simple name. In that case, there would be no way for the C compiler to resolve a simple name, and so an error is issued. Resolving this ambiguity requires a qualified name in at least one of the **C export** statements and then a qualified name in all references to that type in C.

The semantics for the type name syntax in the **C export** statement is overloaded. Reference of a type name is resolved in the same way as any other context. Only after the reference is resolved, the syntactic distinction is used to determine how the type is exported, that is, as qualified only or both unqualified and qualified. This might have the odd consequence that C export statements in which a type is exported by its simple name must be in contexts where the name is not ambiguous (which might be tricky).

Many types that are not explicitly exported to C are nevertheless available for user C code. This is because all types needed for C code generated by Specman Elite are automatically exported in the header file. For this reason, there can be a lot of C code that presupposes ***e*** types and refers to them by their short name. For compatibility, all types that are automatically exported by code generation will be exported with both their simple and their qualified name in all cases where no conflict would arise for the C compiler. In particular, **e_core** types, which are customarily used by C interface without an explicit export statement, would still be available to C code with unqualified names.

6 Compatibility

6.1 Compatibility Overview

Compatibility is the main constraint on this enhancement to ***e***. Compatibility considerations must be examined on three levels:

- "Strict Compatibility" — Existing legal source code must load/compile and behave logically in the same way as before.
- "Output Compatibility" — Printing and data presentation format must remain the same. This is important for tests based on *diff* and master logs.
- "eRM Compatibility" — Previous conventions and coding styles must be kept. There are prevailing naming conventions in ***e***RM and an increasingly large amount of existing code that conforms to those naming conventions.

© 2006 Cadence Design Systems, Inc. All rights reserved worldwide

6.2 Strict Compatibility

All simple type names are resolved in the same way as before if there is only one type with that name in modules already loaded. This is true for both ***e*** code and commands (see "Type Reference Resolution" on page 5 and "Interactive Scope Commands" on page 12). Prior to this language enhancement, no two loaded types could have the same name. Therefore this rule guarantees that all previously legal code is still legal and has the same semantics.

Code in C using ***e*** types by C-interface might have previously referred to types without explicitly exporting them. As no qualified names exist today, all these references from C are with the type's short name. For this reason, the policy for automatically exporting types to C (see "C Interface Support" on page 14) is very permissive. Only when two types with the same name are needed in C during a single compilation phase are they exported exclusively with qualified names. This policy guarantees full backward compatibility for exiting C code in exiting designs.

The unified semantics for type reference in commands (defined in "Interactive Scope Commands" on page 12) implies a change of behavior. Where a name was given as a command argument, for which no type by that name existed, most commands previously passed silently without notification. In case of debugging commands like **break**, **trace**, and **echo event**, if a struct by the required name was later introduced by new loads, the command was taken to relate to it. According to the new semantics, an error is reported in this case. No specific reference to non-existent or future structs members can be made. This is an esoteric incompatibility, involving only scripts with debugging commands. It can be worked around easily. Note that the wildcard semantics is still delayed and can be used to support such a use model.

6.3 Output Compatibility

As long as no two loaded types have the same name, the rules for data presentation in "Printing and Data Presentation" on page 13 guarantee that only a simple name is printed. So here too, full compatibility is attained. User-controlled configuration on type name printing, which would force all printing to be in terms of simple names, can also be supported. This might come in handy, for example, in a new environment using a heritage log base.

6.4 eRM Compatibility

Currently many existing ***e***VCs are ***e***RM-compliant, and a very large amount of code also complies with the ***e***RM naming conventions. Eventually, with the linguistics support for namespaces, ***e***RM naming conventions with regard to types might be relaxed. Other ***e***RM naming rules would still have to be kept, that is, the ones concerning packages, C-like defines, struct-members declared outside the struct's declaring package, and enum-items outside the enum-type's declaring package. However, relaxing the naming conventions would be up to ***e***VC developers, and in any case old ***e***VC code and old ***e***VC user code would remain indefinitely.

© 2006 Cadence Design Systems, Inc. All rights reserved worldwide

There will be no problem to continue working with ***e***VCs implemented with the old-style naming rules, even in environments that contain new-style ***e***VCs. Mixed environments might, however, cause some confusion; because the naming style of the different ***e***VCs would have to be kept in mind by users.

In principle, different degrees of compiler support can be implemented to facilitate the style change. For example, reference to types named with the old ***e***RM style might be made by the new-style qualified name, and possibly even with an unqualified name and some resolution process. The need for this feature and its possible draw-backs have not yet been investigated.

© 2006 Cadence Design Systems, Inc. All rights reserved worldwide

© 2006 Cadence Design Systems, Inc. All rights reserved worldwide

Index

A

APIs, built-in 13

C

C interface support 14
code comparison 5
compatibility 2, 15
    *e*RM 16
    output 16
    strict compatibility 16
considerations 2
coverage callback support 14

D

debugger scope 13

E

e naming problem 1
e_core types 5
*e*RM 16
examples 9, 10, 11
    *e*RM naming conventions 6
    namespace support 7

I

interactive scope commands 12
introduction 1

L

load clusters 8

N

names 4
    NLT type
        overview 11
    qualified 4
    reserved type names 3
    resolution 3
    rules 4
        NLT type 12
    unqualified 4
namespace solution, overview 2
namespaces, for other named entities 3
naming problem in *e* 1
NLT type names 4
    overview 11
    rules 12
non-load-time
    See NLT

O

output compatibility 16

© 2006 Cadence Design Systems, Inc. All rights reserved worldwide

P

packages as namespaces 2
presentation, data 13
printing 13

Q

qualified names 4

R

reflection 13
reserved type names 3

S

scope commands, interactive 12
simplicity 2

T

type names
    NLT 4
    reserved 3
type reference resolution 5

U

unqualified names 4
usability 2
use relation 3, 8
    definition 8

V

vr_xbus_*.e 6, 7
vr_xserial_*.e 6, 7
vr_xsoc_*.e 6, 7

W

wildcards 12

© 2006 Cadence Design Systems, Inc. All rights reserved worldwide