Dependency Utility

Introduction 1

Important note: 1

Print 1

Print direct dependencies 1

Syntax: 1

Usage example: 1

Queries 1

find\_module\_dependencies\_recursively 1

Syntax: 1

Usage example 1

find\_all\_dependencies\_by\_pattern 1

Syntax: 1

Usage example: 1

Data types used in the queries 1

rf\_structural\_element 1

direct\_dependency\_info 1

dependency\_info 1

module\_dependencies 1

# Introduction

The purpose of the Dependency Utility is to provide information for understanding dependencies between files. Based on this information, the users can restructure the environments – create sub environments omitting files that are not required under a specific configuration, create elibs so that build performance will be improved, etc.

The utilities provide two kinds of methods:

* **Print** – print out the requested data.
* **Queries** – return data that can be further analyzed by the user.

The file **example\_report\_dependencies.e** contains examples of some advanced analysis, calling the queries that are implemented in **e\_util\_dependency\_util.e.** Users can use **example\_report\_dependencies.e,** copy and edit as required,for creating their own report in preferred format.

## Important note:

For the tool to work, must activate the lint\_mode. **Before** loading the files, issue this command:

**Specman> config misc -lint\_mode;**

For example:

specman -command ‘**config misc -lint\_mode**; **load e\_util\_dependency\_util**; **load** my\_env\_top; **dependencies\_query::print\_all\_dependencies\_by\_pattern**(module, "checker", type, "\*", TRUE)’

# Print

## Print direct dependencies

The method **dependencies\_query::print\_all\_dependencies\_by\_pattern** prints all direct dependencies between two elements.

The element can be a directory, a file or a type.

This method prints a list of reasons why the dependent depends on the ‘dependee’. If details == TRUE, then a link to the relevant line of code in the dependee is printed as well.

### Syntax:

print\_all\_dependencies\_by\_pattern(   
 dependent\_kind : element\_kind,   
 dependent\_pattern : string,   
 dependee\_kind : element\_kind,   
 dependee\_pattern : string,  
 detailed : bool,  
 first\_module\_name : string = “”,  
 last\_module\_name : string = “”);

dependent\_kind : element\_kind - one of - directory, module, type  
 dependent\_pattern : string - can contain wildcards   
 dependee\_kind : element\_kind - one of - directory, module, type  
 dependee\_pattern : string - can contain wildcards   
 detailed : bool - when TRUE, prints detailed info of   
 all dependencies.  
 when FALSE, prints count of dependencies  
 first\_module\_name : string - start the analysis from this module.   
 If empty string – start with first   
 loaded/compiled module  
 last\_module\_name : string - this is the last module to analyze.   
 If empty string – analyze until the   
 last loaded/compiled module

### Usage example:

Assume the module *top* calls a method that was defined in module *env\_1*, and also uses a type that was defined in module *env\_2*.

Calling this method:

dependencies\_query::print\_all\_dependencies\_by\_pattern(module, "top", module, "\*", TRUE)

Results with:

direct\_deps =

Dependent element Dependee Count

------------------------------------------------------------------------------------

rf\_method\_layer, line 12 in @top rf\_method 'foo', line 3 in @env\_1 1

rf\_module 'top' rf\_enum 'my\_kind', line 2 in @env\_2 1

rf\_field\_declaration, line 10 in @top rf\_enum 'my\_kind', line 2 in @env\_2 1

Explaining this report:

1. A method defined in *top.e line 12*, calls the method *foo()* that was define at *line 2 in env\_1.e*
2. The module *top.e* uses the enum *my\_kind* that was defined *at line 2 in env\_2.e*
3. The field defined at *line 10 in top.e* uses the enum *my\_kind* that was defined at *line 2 in env\_2.e*

# Queries

Using the queries, one can implement their own code, for getting exact information of interest.

## find\_module\_dependencies\_recursively

The method **dependencies\_query::find\_module\_dependencies\_recursively**() returns a list of[**module\_dependencies**](#_module_dependencies)**.**

The first item in the list contains a list of all the modules that *module\_name* depends on. For each of these modules – there is another item in the list, of all the modules this module depends on, etc.

If, for example, module A depends on module B and C, and module C depends on module D, then this method will return this list of 4 **modules\_dependencies**:

1. Module is A, **all\_deps** contains module B and module C
2. Module is C, **all\_deps** contains module D
3. Module B, **all\_deps** is empty
4. Module D, **all\_deps** is empty

The [**module\_dependencies**](#_module_dependencies)does not contain information of the causes of the dependency. For getting more information, you should further investigate using find\_all\_dependencies\_by\_pattern().

### Syntax:

dependencies\_query::find\_module\_dependencies\_recursively(  
 module\_name: string,  
 first\_module\_name : string = “”,  
 last\_module\_name : string = “”);

module\_name : string, can contain wildcards  
first\_module\_name : string, start the analysis from this module.  
 If empty string – start with first  
 loaded/compiled module  
last\_module\_name : string, this is the last module to analyze.   
 If empty string – analyze until the last   
 loaded/compiled module

### Usage example

This code implements a method that gets the **module\_rdependencies** of a given file, and prints the names of all the dependee files. It analyzes only the modules starting with *my\_base\_env.e*, assuming that all the modules before it are common modules, used by all, no need to analyze them

report\_all\_dependencies(dependent\_name : string,   
 report\_style : report\_style,   
 “my\_env\_base”, “”) is {

var rf\_module\_l : list of rf\_module;

// Get the info - module\_dependencies - of all   
 // modules that dependent\_name depends on  
  
 var module\_dependencies\_l :=   
 **dependencies\_query::find\_module\_dependencies\_recursively**(  
 dependent\_name);   
 if module\_dependencies\_l is empty {  
 out("\n\nThe module " , dependent\_name,   
 " does not depend on any other module in this env");  
 };   
   
 // each **module\_dependencies** contain:  
 // the dependent module  
 // list of depdencee modules  
 for each (one\_module\_dependencies) in module\_dependencies\_l {  
 // From the module\_dependencies, get the list of   
 // all the modules  
 rf\_module\_l =   
 one\_module\_dependencies.get\_all\_deps();  
 out(one\_module\_dependencies.get\_dependent().get\_name(),  
 " depends directly on:");  
 for each in rf\_module\_l {  
 out(" ", it.get\_name());  
 };  
 };  
};

## find\_all\_dependencies\_by\_pattern

This method returns a list of [dependency\_info](#_dependency_info). Each **dependency\_info** holds information of dependencies between two elements.

### Syntax:

find\_all\_dependencies\_by\_pattern (  
 dependent\_kind : element\_kind,  
 dependent\_pattern : string,  
 dependee\_kind : element\_kind,  
 dependee\_pattern : string,  
 first\_module\_name : string,  
 last\_module\_name : string): list of dependency\_info

dependent\_kind : element\_kind - one of - directory, module, type  
dependent\_pattern : string - can contain wildcards   
dependee\_kind : element\_kind - one of - directory, module, type  
dependee\_pattern : string - can contain wildcards  
first\_module\_name : string – ignore modules loaded/compiled   
 before this module  
last\_module\_name : string – ignore modules loaded/compiled   
 before this module

### Usage example:

The following code looks for all the dependencies of two files, and after getting the information from the dependency info structs – prints the details.

report\_dependencies\_two\_files( dependent\_name : string,  
 dependee\_name : string) is {

var one\_dependency\_info : dependency\_info;  
 var direct\_dependency\_info\_l : list of direct\_dependency\_info;

// Get the list of all dependencies of dependent\_name  
 // and dependee\_name.  
 // The list will be of size of one because we call with   
 // exact names.  
 // There is one dependency\_info for each pair of two elements.  
 // When calling this method with wildcards - the return list   
 // can contain multiple items,  
 // each representing dependencies of two modules  
  
 var dependency\_info\_l : list of dependency\_info;  
 dependency\_info\_l =  
 **dependencies\_query::find\_all\_dependencies\_by\_pattern**(  
 module, dependent\_name,   
 module, dependee\_name);

if dependency\_info\_l is empty {  
 out("\nThere is a limitation - we know that ", dependent\_name,  
 " depends on ", dependee\_name, " but info is missing");  
 return;  
 };  
  
 one\_dependency\_info = dependency\_info\_l[0];   
 out("\n\nThe dependencies between ",  
 dependent\_name, " and ", dependee\_name, " :");   
  
 // The dependency\_info contains two elements -   
 // one dependent and one dependee,  
 // and a list of direct\_dependency\_info.   
 // Get the exact reason/s for the dependency.   
 // What line in dependent\_name  
 // uses what line in the one\_rf\_module module  
  
 direct\_dependency\_info\_l =   
 one\_dependency\_info.get\_direct\_dependencies();  
 for each (one\_direct\_dependency\_info) in   
 direct\_dependency\_info\_l {  
 outf("%s depends on %s\n",   
 one\_direct\_dependency\_info.get\_dependent().  
 get\_printed\_lines(),  
 one\_direct\_dependency\_info.get\_dependee().  
 get\_printed\_lines());  
 };   
 };

# Data types used in the queries

The queries return lists of **module\_dependencies** and **dependency\_info.** You can use the api of these types, to retrieve detailed information about the dependency nature.

### rf\_structural\_element

This is the basic struct. Contains information about one element (a directory, a file or a type).

**rf\_structural\_element** api:

* get\_element\_name(): string
* get\_element\_source\_line(): string
* get\_element\_module\_name(): string
* get\_element\_container(): string

### direct\_dependency\_info

This struct contains information about the dependencies between two elements, each represented by a [rf\_structural\_element](#_rf_structural_element). The count is the number of times in which the dependent uses the dependee. If, for example, method *foo()* calls method *kuku()* twice, then the dependent is the element of method *foo*, the dependee is the element of method *kuku*, and the count is 2.

**direct\_dependeny\_info** api:

* information about the dependent:
  + get\_dependent() : rf\_structural\_element
* information about the dependee
  + get\_dependee() : rf\_structural\_element
* information about the dependencies
  + get\_dependency\_count(): int

If you call a query that returns a struct of type **direct\_dependeny\_info**, you can further query this type for getting the required information, for example:

var one\_dpi : direct\_dependency\_info;  
print one\_dpi.get\_dependent().get\_element\_name();  
print one\_dpi.get\_dependent().get\_element\_source\_line();

### dependency\_info

This struct contains information of all dependencies between two elements. It contains information about the elements, and also – a list of [direct\_dependency\_info](#_direct_dependency_info).

**dependeny\_info** api:

* information about the dependent:
  + get\_dependent\_element\_name(): string
  + get\_dependent\_source\_line(): string
  + get\_dependent\_container(): string is
* information about the dependee
  + get\_dependee\_element\_name(): string
  + get\_dependee\_source\_line(): string
  + get\_dependee\_container(): string
  + get\_dependee\_module\_name(): string
* information about the dependencies:
  + get\_dependency\_count(): int
  + get\_direct\_dependencies(): list of direct\_dependency\_info

### module\_dependencies

This struct contains a rf\_module – the dependent, and a list of rf\_moduels - all the modules that it depends on.

**module\_dependencies** api:

* get\_dependent() : rf\_module - the dependent
* get\_all\_deps() : list of rf\_module – all the modules that the dependent depends on