Bell-Northern Research
VHDL Group

Internal Workings of vmkr
Application Note

Himanshu M. Thaker
Dept. 5S71

May 7, 1991

1. Introduction

This document has not been modified since May 1991, and is out of date with respect to the actual software. However, it serves as a good introduction to anyone who wishes to modify vmkr to suit their own needs.

2. Dependency Rules

Primary units are : entity declarations, package declarations, and configuration declarations. Secondary units are : package body, and architecture body. Order of analysis is governed by three rules. These rules are :

1) A secondary unit can be analyzed only after its corresponding primary unit has been analyzed.

2) Any library unit that references a primary unit can only be analyzed only after that primary unit has been analyzed.

3) An architecture body referenced in a configuration declaration must be analyzed before the configuration declaration; likewise, one configuration declaration referenced in a second configuration declaration must be analyzed before that second configuration declaration.

- above quoted from VHDL : Hardware Description and Design
- by Lipset et al, pg 40

Intermetrics VHDL design Environment (VDE) User's Manual gives the following dependency rules regarding the order of analysis:

1) A package body depends on its package declaration. A package body may not be analyzed into a library unless the package declaration is present in the same library, although the package declaration and the package body do not have to be analyzed at the same time.

2) An architecture body depends on its entity declaration. An architecture body may not be analyzed into a library unless the entity declaration is present in the same library, but the entity declaration and the architecture body do not have to be analyzed at the same time.

3) Any unit that contains a reference to a primary unit (package declaration, entity declaration or configuration declaration) depends on that primary unit. The referenced primary unit will have been made visible by a VHDL use clause or a VHDL library clause and a VHDL use clause.

Intermetrics also gives the following rules for the order of model generation

1) A package declaration must be model generated before its corresponding package body can be model generated.

2) A package declaration must be model generated before any unit that uses the package declaration (including an architecture body of an entity declaration that uses the package declaration) can be model generated.

3) All the architecture bodies and configuration declarations implied by a configuration

declaration must be model generated before the configuration declaration itself can be model generated.

2.1. Some Definitions

There are four phases required in executing VHDL code. They are :

- analysis -this is where syntactic and static semantic analysis occurs
- model generation - high level code is generated from VHDL, then compiled and placed into the design database
- build phase - a simulation kernel is created by linking the object modules (created by the model generation phase)
- simulation phase - execution of the kernel created by the build phase

The VHDL toolset does not affect the timestamps of the VHDL source codes. Thus, it is not possible to determine in what state of execution the toolset is, simply from examining the timestamps of the source files. One solution around this, to keep the state information, is to create 'dummy' files, whose existence (along with their timestamps) store this information (even though these 'dummy' files are empty).

Vmkr creates and depends on these files for such information. By definition, these files will begin with a '.', so that when the 'ls' UNIX command is executed (with no options, of course), the 'dummy' files are not listed. Three extensions are defined. A file ending in :

- ".ana" indicates that the corresponding unit has been analysed
- ".mg" indicates that the corresponding unit has been model generated, and
- ".bd" indicates that the corresponding unit has been built.

2.2. A Quick Review of the Unix make command

Vmkr examines the VHDL files presented to it, and after building an internal representation of the dependencies, generates statements that are compatible with the UNIX make command. The make command requires statements to be of the following format :

```
TARGET : dependency1 dependency2 ... dependencyn
        command1
        ...
        commandn
```

When asked to make the TARGET (either directly from the user, or from another statement within the makefile), make examines the timestamp of TARGET, and compares it to the timestamps of its dependencies. If the TARGET is older than any of the dependencies, then the commands are executed. There is a special command called "touch $@" which touches, or changes the timestamp of its TARGET to the current time. Thus, if the touch is the last command in the list, after executing all the commands, TARGET will be made up to date relative to its dependencies (because its timestamp now will be newer than the rest).

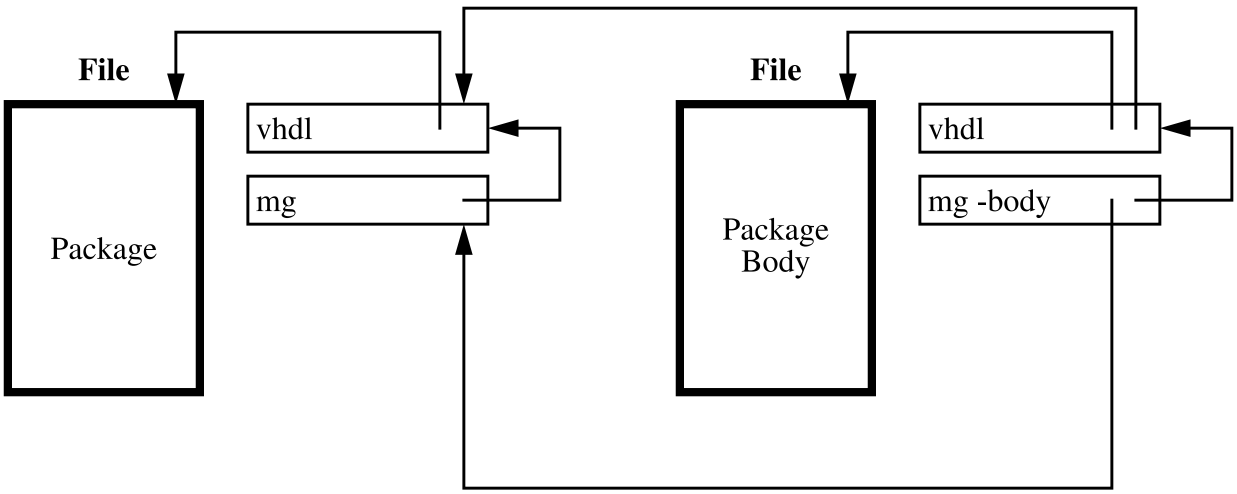
Any command that is preceeded by an @ will not be echoed to the terminal. Thus "@touch $@" executes silently.

2.3. Graphical Representation of the Dependency Rules

The dependencies are created based on the rules presented in section 2. on page 2. The following illustrates graphically what those rules indicate.

2.3.1 Package and Package Bodies

A package and package body have the following dependencies.



Package and Package Body Dependencies

Figure 1

This diagram indicates that :

1) the analysis (vhdl) of the package depends on the file that the package resides in.

2) the model generation (mg) of the package depends on the analysis.

3) the analysis of the package body depends on the file that the package body resides in, as well as the anaylsis of the package.

4) the model generation of the package body depends on the analysis of the package body, as well as the model generation of the package.

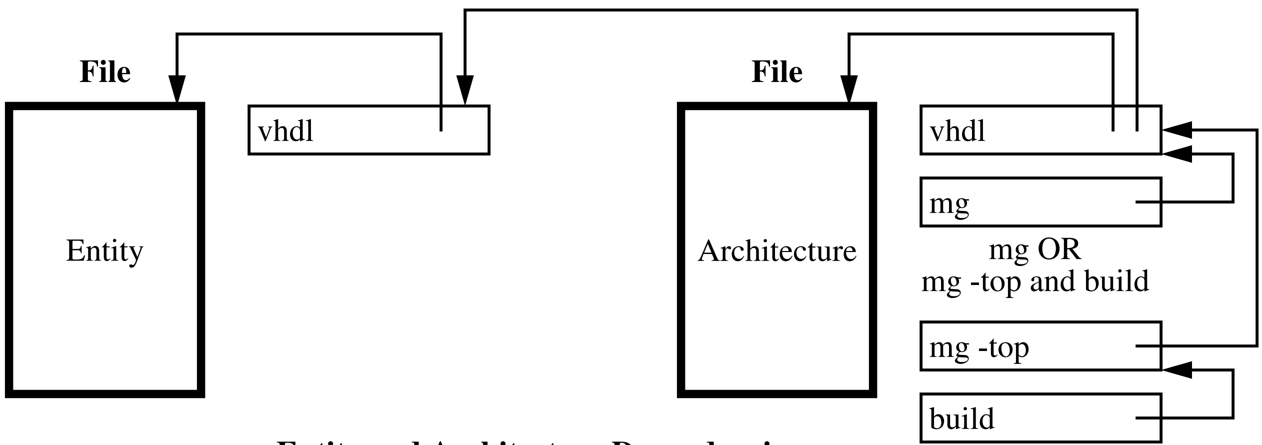
As an example of dependency changes, the above diagram indicates that if the package should change, then the dependency rules would indicate :

- the package should be reanalysed
- then, the package body should be analysed, and the package should be model generated (it does not matter, without additional dependency rules, which occurs first)
- finally, the package body should be model generated.

There are a few variations as to how package and package bodies can exist. For example, the package and package body may reside in the same file. Additionally, a package body may not exist. Modification of the rules are straightforward to accomodate these variations.

2.3.2 Entities and Architectures

Entity and architecture dependencies are shown in Figure 2. Note that an entity could reside in the same file as the architecture. Additionally, there could be multiple architectures for every entity



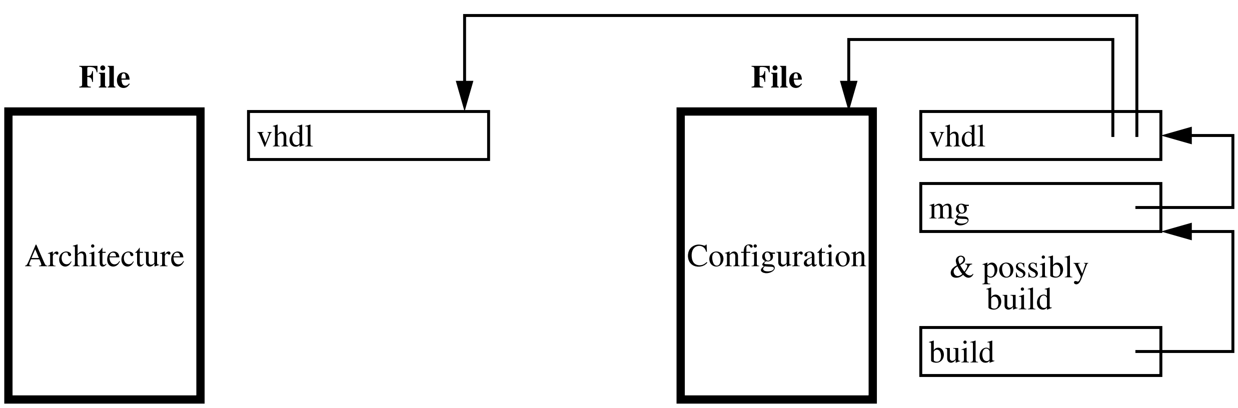
Entity and Architecture Dependencies

Figure 2

(but not vice versa).

2.3.3 Configurations

Configuration dependencies are shown in Figure 3. A configuration could reside in the same file as the entity or architecture. These variations have simple changes to the dependency diagrams, and are not illustrated here.

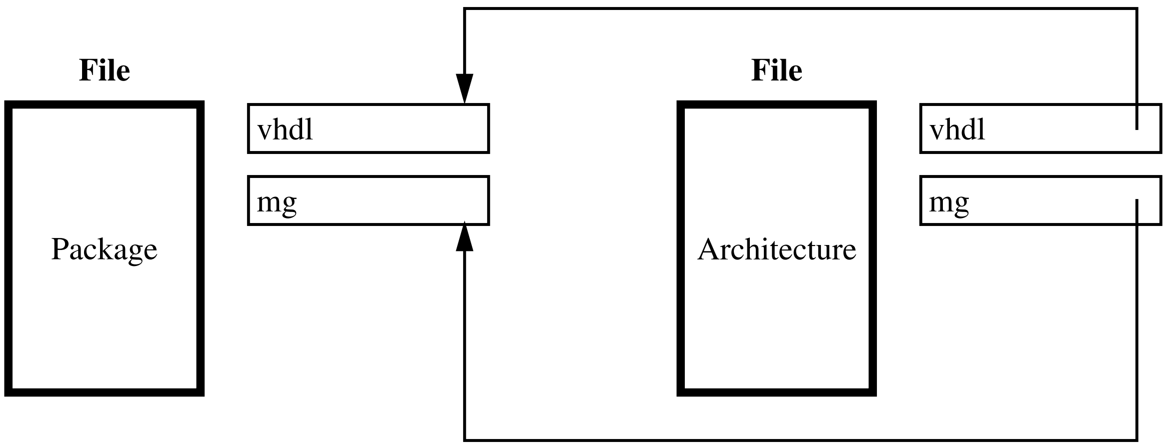


Configuration Dependencies

Figure 3

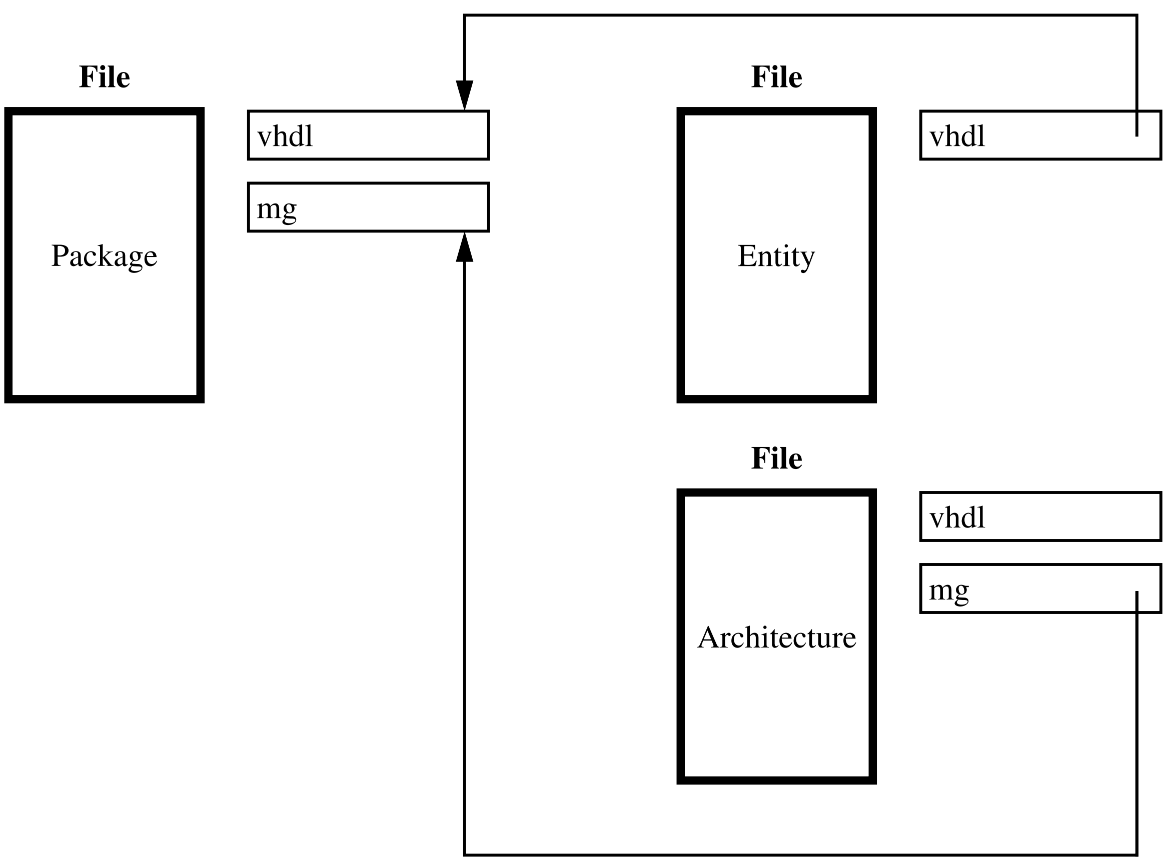
2.3.4 Additional Dependencies - Architectures, Entities and Packages

If an architecture uses a package, then the analysis of the architecture is dependent on the analysis of the package. Similarly for the model generation, the architecture is dependent on the model generation of the package. If an entity uses a package, then the analysis of the entity is dependent on the analysis of the package, and all of the architectures of the entity (recall that there can be more than one architecture per entity) depend on the package being model generated before they can be model generated. Note that the package body does not come into play, whether it exists or not. The graphical depiction of these dependencies are shown in Figure 4 and Figure 5.



Architecture Depends on Package

Figure 4

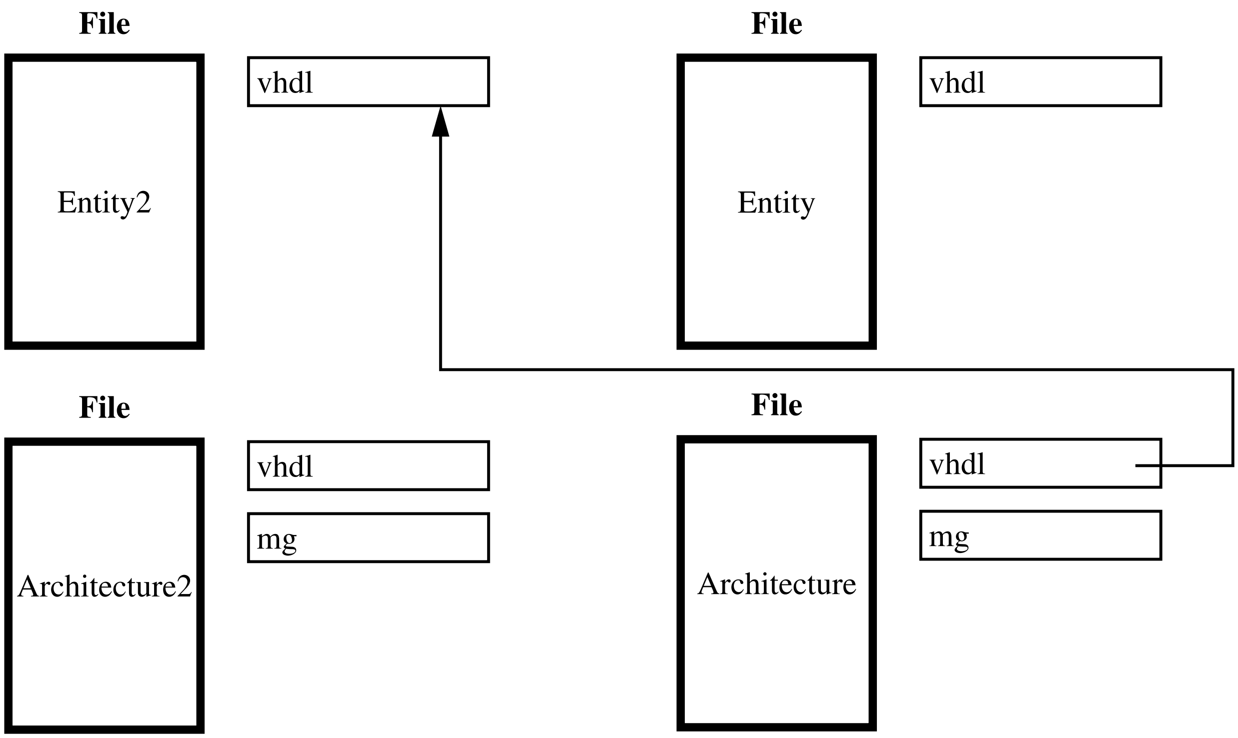


Entity Depends on Package

Figure 5

2.3.5 Additional Dependencies - Architectures and other Entity-Architecture Pairs

If an architecture depends on another entity architecture pair (i.e. the USE ENTITY clause, when binding a component to a specific entity-architecture), then the model generation of the architecture is dependent on the analysis of the entity being used, as shown in Figure 6.



Architecture Depends on an Entity-Architecture Pair

Figure 6

2.3.6 Additional Dependencies - top level kernals on Architectures and Configurations

Before a top level kernal can be built for an archtecture (or configuration, though not depicted in the graphical representation), all of the architectures and configurations used (recursively, that is, not only the ones used, but the ones used by the ones used, etc.) must be model generated. This is shown in Figure 7.

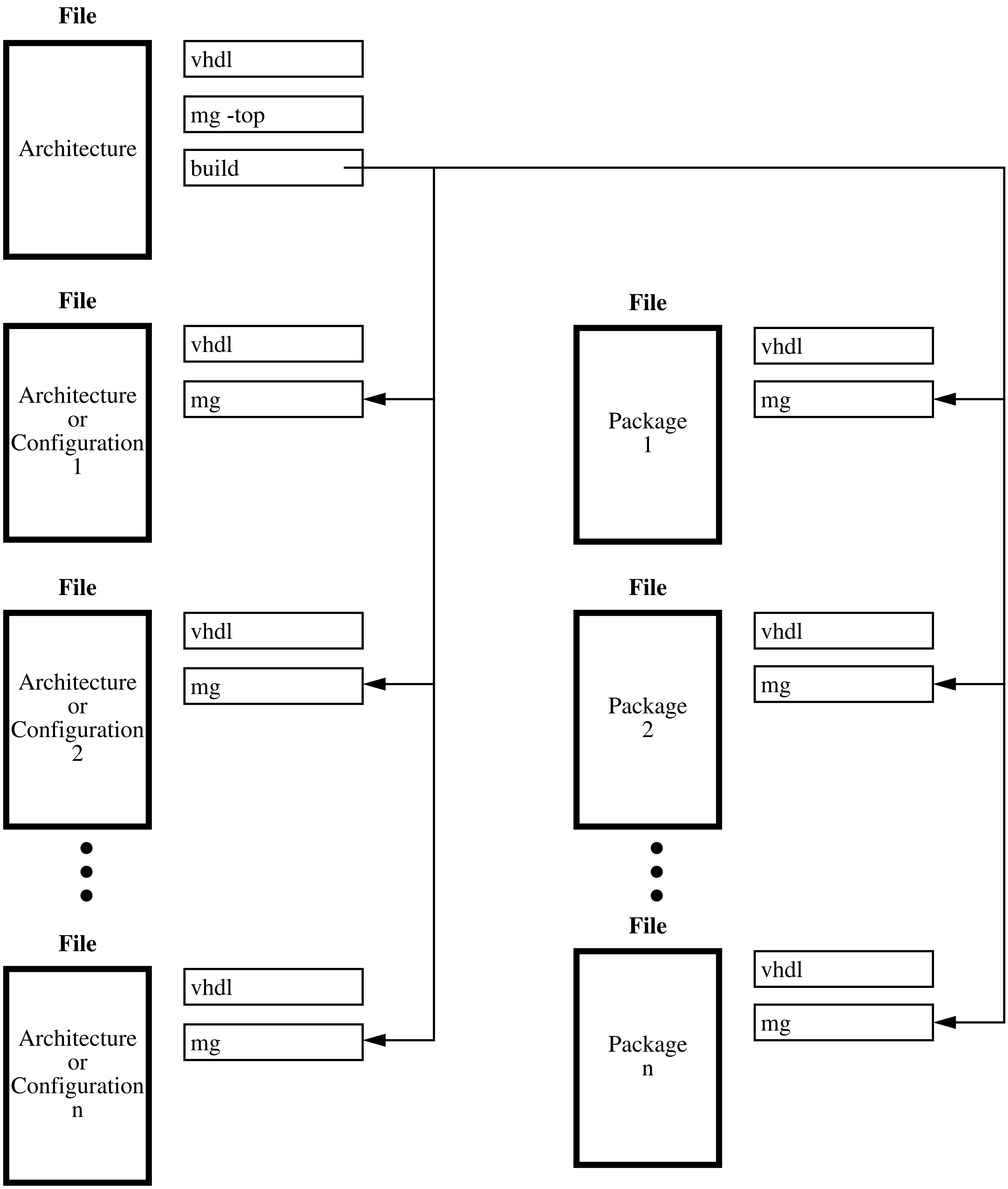
2.3.7 Additional Dependencies - Configurations on its components

Before a configuration can be analysed, all its components must be analysed, recursively. Additionally, before a configuration can be model generated, all its components must also be model generated, recursively. This is shown in Figure 8.

3. The VHDL Makefile Program

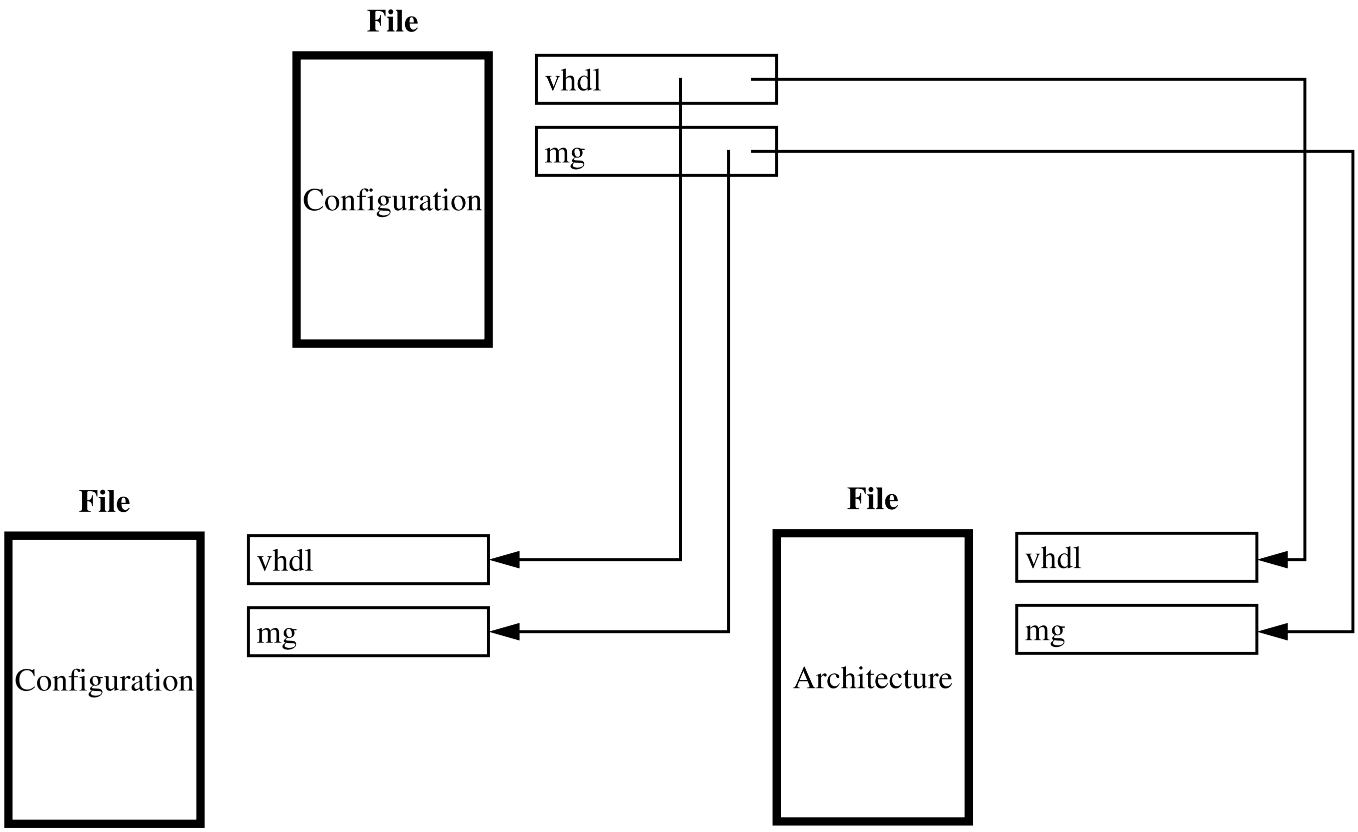
The VHDL Makefile program (or vmkr) builds information about dependencies of the files that it is asked to, and creates a make compatable output that will indicate the order of compilation etc. required. A block level description of how this program functions is shown in Figure 9. This figure is to be interpreted as follows :

- repeatedly execute
    get keywords



A simulation kernel depends on the model generation of all its components, and any components that its components depend on, recursively.

Figure 7



Before a configuration can be model generated, all the architectures and components used must be model generated (recursively, for configurations).

Figure 8

update linked lists
- then back annotate
- then print out the makefile

The module "get keywords" functions as follows :

open the next file
repeatedly execute
get next token
check for keywords

The functionality of the blocks is :

"open next file" - this block does just that, namely opens the next file for parsing

"get next token" - this block examines the file just opened, and parses tokens from that file.

"check for keywords" - from all of the tokens parsed, check for keywords that vmkr requires, and construct a linked list of these tokens.

"get keywords" - the parent block that calls "open next file", "get next token", and "check for keywords".

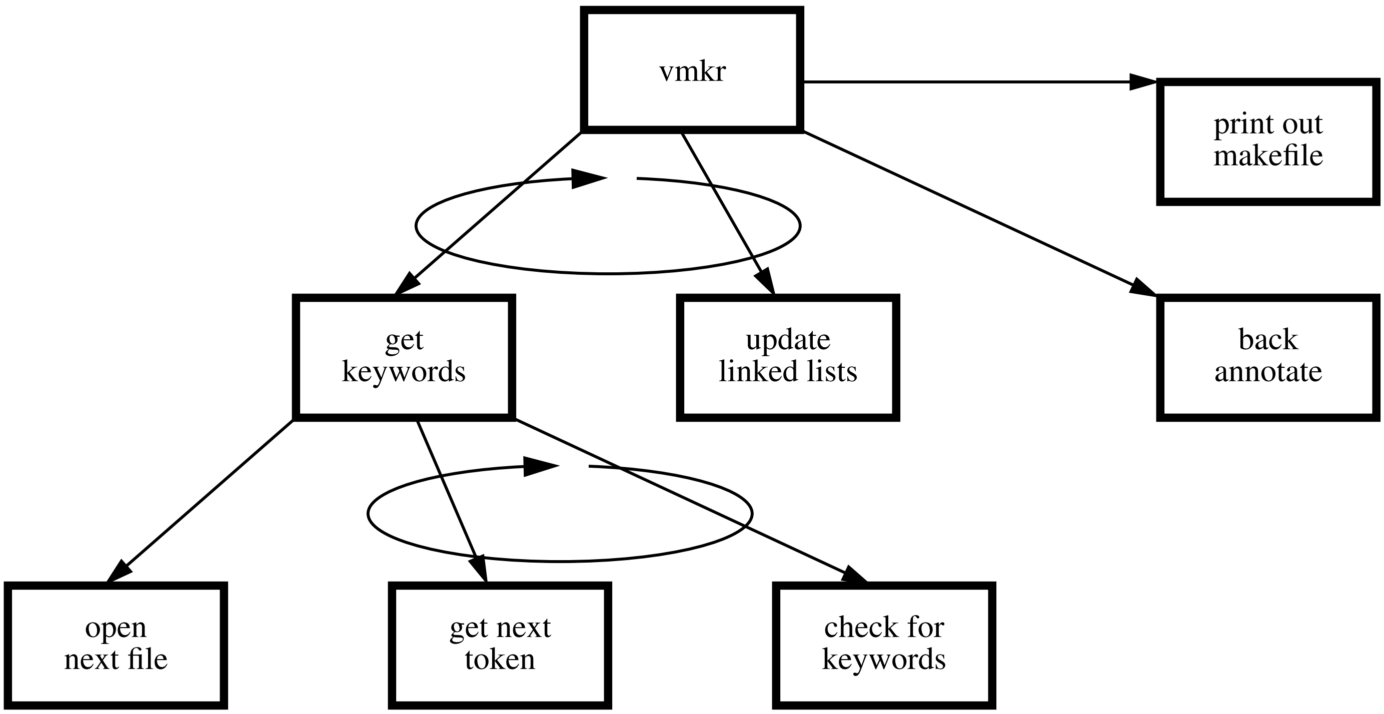


Figure 9

"update linked lists" - this block takes the linked list of tokens passed to it, and updates the dependency linked list (the list that maintains the dependency information).

"back annotate" - this block back annotates the various fields in the dependency list, after all of the files have been parsed and their information added to the list.

"print out makefile" - after the back annotation, it is now possible to print out the dependencies, conforming to the make command format.

"vmkr" - the VHDL Makefile maker

3.1. Source Code

Source code for vmkr currently resides in /usr/users/hemi/Source/Vhdlmaker on the machine bck-rd12. The source is divided up among five files, corresponding to the five top blocks (vmkr and the four immediate blocks below it). Additionally, there is one include file defining the structures of various records.

3.2. Block "get next token"

This block is implemented using a Mealy-type Finite State Machine, getting the next token from the file indicated. In vmkr, a token is defined as all the characters between two characters that delimit tokens. This rule is a general statement, and needs a few more definitions to precisely define a token for our needs. For example, the statement

VARIABLE current_frame : integer := 0;

has five tokens ("VARIABLE", "current_frame", "integer", "0", ";").

Thus "get next token" parses the input file character by character until the next token is found, or the end of the file is reached. The FSM controlling the parsing is depicted in Figure 10. This FSM has three special definitions, namely T, TT, and TS. T is defined to be a terminating token, one of " []()+:|#^*,<>/", and tab and end of line. TT or Terminating character that is also a token, is defined here to be ";". TS, or Terminating character that needs to be Sent back to the input stream is defined to be "-" or ". The reason for TS characters being sent back to the input stream is that they could form the beginning of a comment (two "-"'s in a row), or a literal quote. Any comments, and any literal quotes are to be ignored during parsing. Thus, to keep the FSM simple, it was decided to return any TS characters encountered while parsing a token. Note that it is possible to create an FSM that does not need to send these characters back, but it would complicate the FSM unnecessarily.

3.3. Block "check for keywords"

Once a token is extracted from the input stream, it is sent to the block "check for keywords". This block filters the keywords, and for those that are desired, creates a linked list of tokens. Again, an FSM is employed to assist in this task. This FSM is shown in Figure 10. This FSM is used to group the keywords desired, and their appropriate modifiers, together. For example, the keyword "entity" is followed by the entity name. Thus, when "entity" is encountered, it must be paired with its entity name. Different keywords can have a variable number of such modifiers. A keyword and its modifiers is known as a token_holder. A linked list of token holders is created for the entire file parsed (see Figure 11), and this linked list is then passed on to the block "update linked lists". The different keywords and their modifiers are shown below :

keyword	var1	var2	token
------	----	----	-----
ARCHITECTURE	architecture name	-	entity name
CONFIGURATION	configuration name	-	entity name
ENTITY	-	-	entity name
FOR	-	-	name
PORT	-	-	name
PACKAGE	-	-	package name
PACKAGE	BODY	-	package name
USE	-	-	package name
USE	ENTITY	entity name	architecture name
USE	CONFIGURATION	-	configuration name

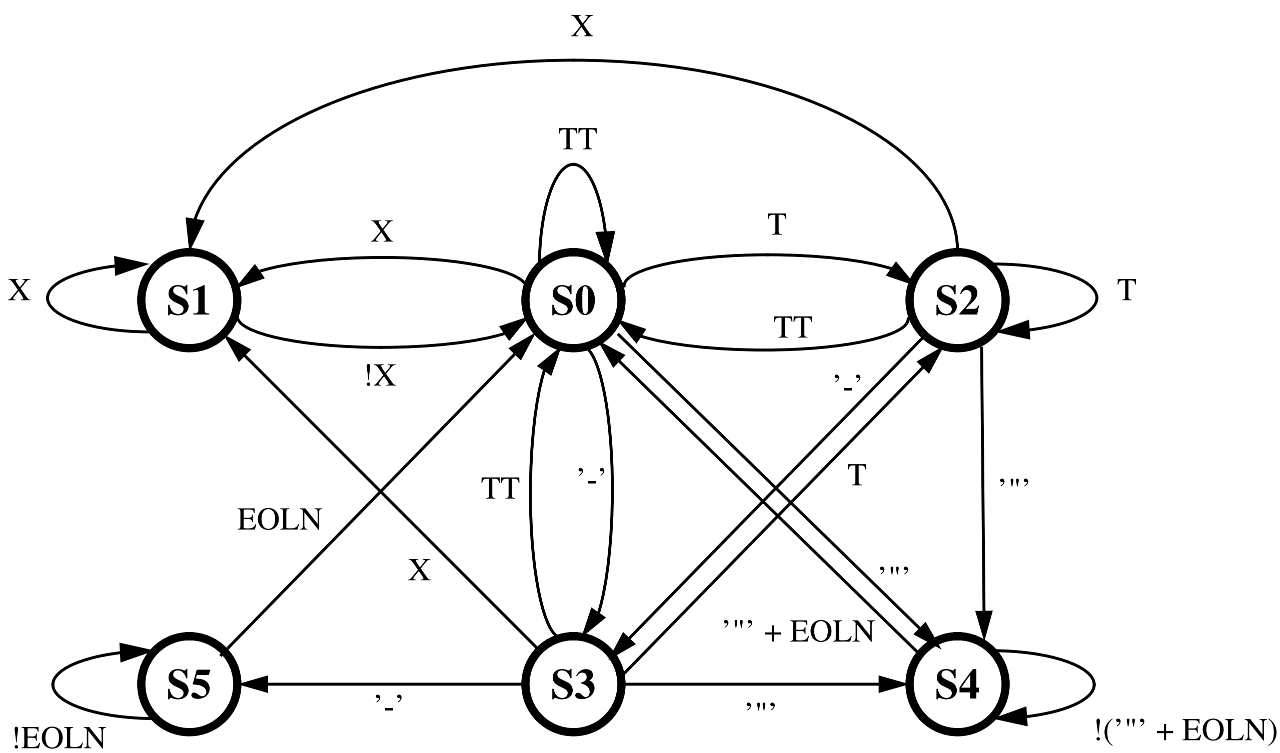
3.4. Block "update linked lists"

Using the linked list of tokens extracted from a file, "update linked lists" does essentially that. That is, there is a database of linked lists maintained for all of the files parsed by vmkr, that tracks the dependencies between the different files. There are four main linked lists (see Figure 12), namely one for packages (and their bodies, if they exist), one for entities, one for architectures, and one for configurations. For example, when an architecture token holder is encountered, "update linked lists" will create a new arch_struct, that contains the description of the architecture, and append it to the architecture linked list.

3.4.1 arch_struct

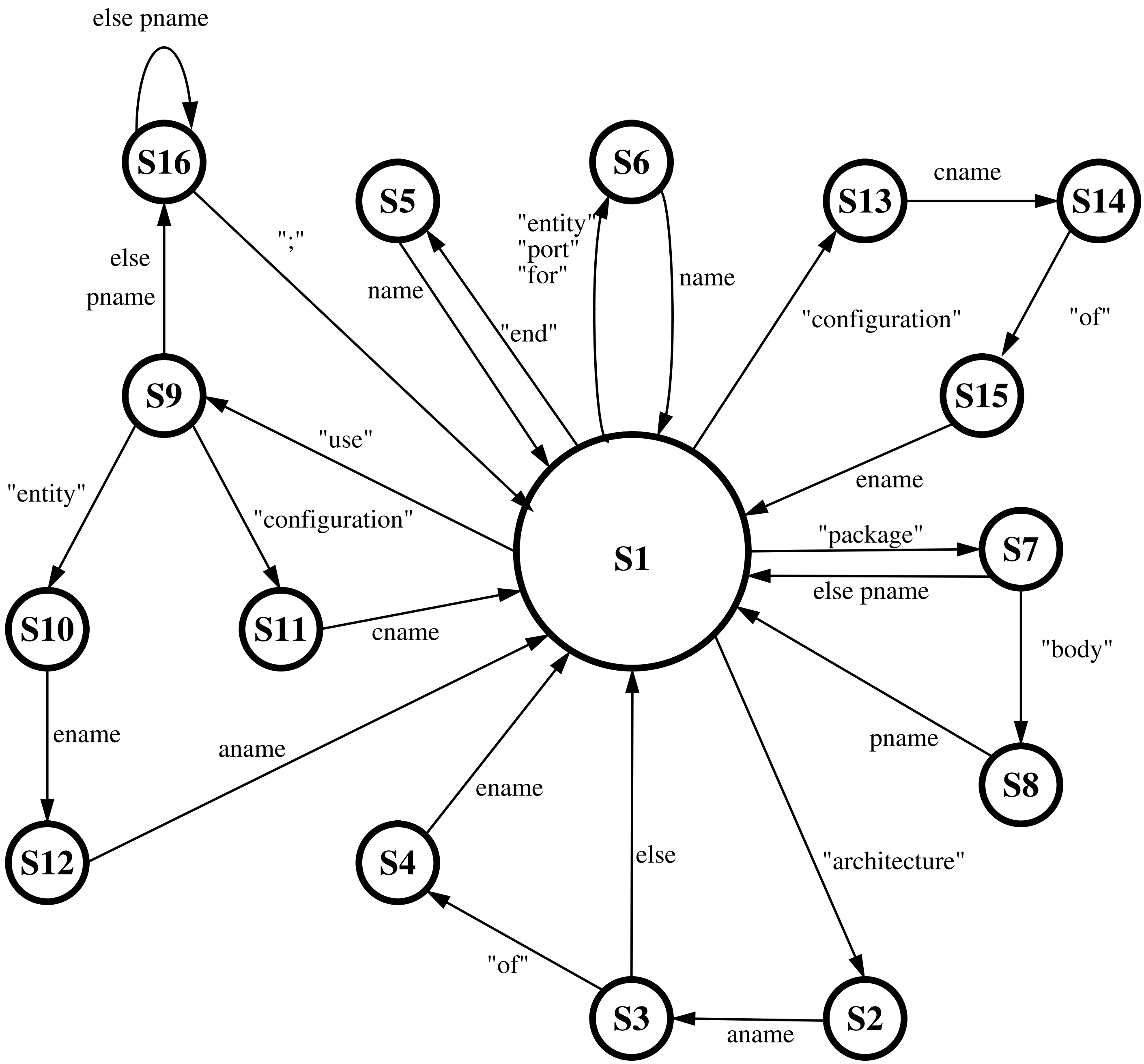
The arch_struct structure holds descriptions of the architecture encountered. There are eight fields that comprise arch_struct. Aname holds the name of the architecture (actually, all text, eg. names,

T = Terminating character
TT = Terminating character that is also a Token
(here, ';' is a TT)
TS = Terminating character that needs to be Sent back to the
input stream (here, '-' and '"' are TS's)
X = !T & !TT & !TS



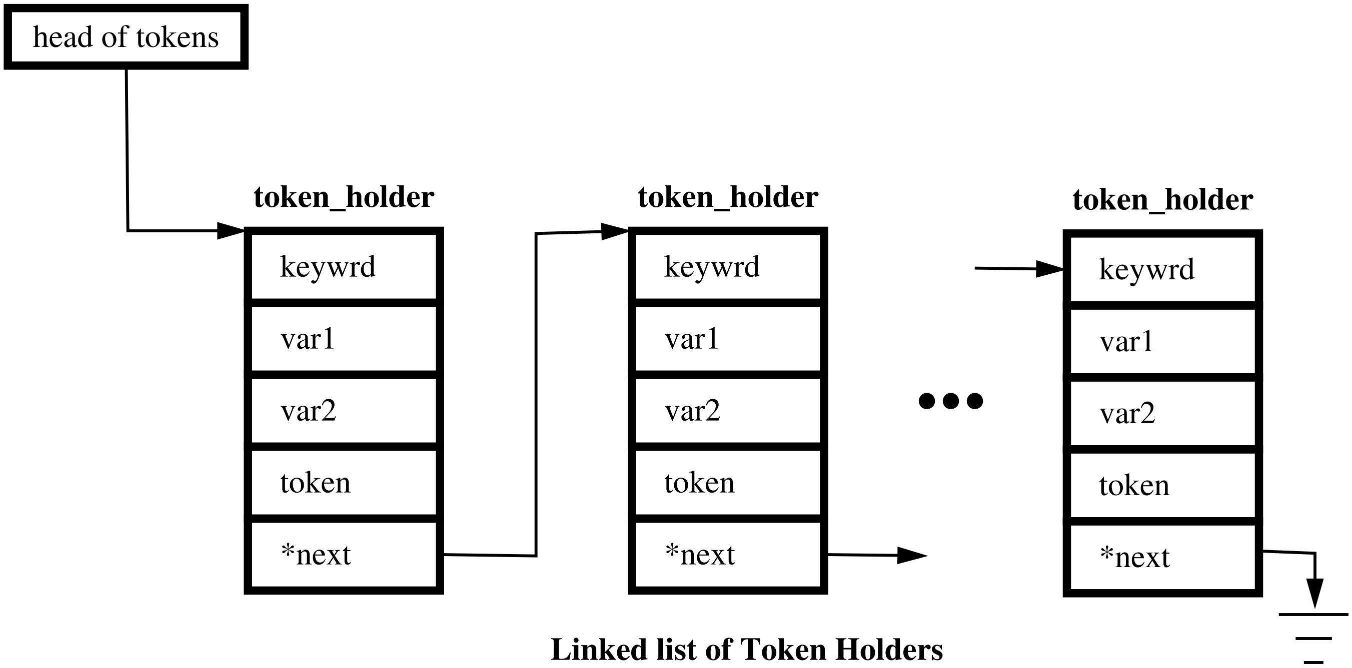
Mealy FSM to parse tokens from an input stream

Figure 10



Mealy FSM to extract keywords from tokens

Figure 11



Linked list of Token Holders

Figure 12

etc., are implemented as pointers to a variable to type char, though it is just referred to as the name). Ename holds the name of the corresponding entity, and faname is the name of the file the architecture is in. Entity is a pointer into the entity linked list that is linked up, in the block "back annotate". This pointer points to the architecture's entity. Likewise config is a pointer into the configuration linked list for the architecture's configuration (if it exists), and this link is also make, in the block "back annotate". Next point to the next arch_struct (creating the linked list). There are two other fields in arch_struct. These are pkg and bind. These are pointers to packages and binding units used by the architecture. While evaluating the token holder linked list, if a package is encountered, a temporary linked list of packages is created, and if an architecture is found, the temporary linked list of packages is appended to the pkg field. Similarily, for all "use" clauses (eg. use entity_name (arch_name)), a temporary binding structure is created, and linked to bind. See Figure 13 for a pictoral representation.

3.4.2 entity_struct

Similar to arch_struct, entity_struct has 7 fields. Ename, f_ename, pkg, and next have similar functions to those described in arch_struct. Ports_exist is a boolean used to determine if the entity in question has any ports. This is useful in determining which entities could possibly be top level entities. That is, an entity without any ports could be a top level entity (and any corresponding architectures top level architectures). Boolean built is used by block "print out makefile" to determine if this entity has been built before (to prevent printing out of the make statement to build the entity twice or more).

3.4.3 packages

Packages has 5 fields. Pname holds the package name, f_pname the file name that package resides in, fpame the file name the package body resides in and next points to the next package in the linked list. Printed is a boolean whose use is similar to that of boolean built for entity_struct.

3.4.4 config_struct

Config_struct has 8 fields. Cname, ename, aname, fcname, entity, arch, bind, and next have similar functionalities to fields previously discussed.

3.4.5 package_ptrs

Package_ptrs has 5 fields. Libname holds the name of the library this package resides in (if any), pname the name of the package, pkg_ptr is used to point to the corresponding package (bound in the block "back annotate", and next points to the next package_ptrs. Package_ptrs are used to indicate what packages entities and architectures use. There is one more boolean field called work that is used to indicate if this package is from the library work. This information is required when printing out the makefile.

3.4.6 binding_struct

Binding_struct has 9 fields. Libname, ename, aname, cname, work, entity, arch, config, and next have similar functionalities to fields previously discussed.

3.5. Block "back annotate"

After all of the files have been parsed, and the linked list database created, it is necessary to back annotate pointers. For example, the entity pointer in every arch_struct must be updated to point to their corresponding entity_struct. This is shown in Figure 14.

Additionally, pkg_ptr links for all of the package_ptrs must be updated to point to their corresponding packages. This is shown in figure Figure 15.

After all of the links have been back annotated, an example of a linked lists could be as shown in Figure 16.

3.6. Block "print out makefile"

Using the linked lists, and the update rules presented in section 2., a makefile is printed out.

Do a man vmkr for information on how to use vmkr.

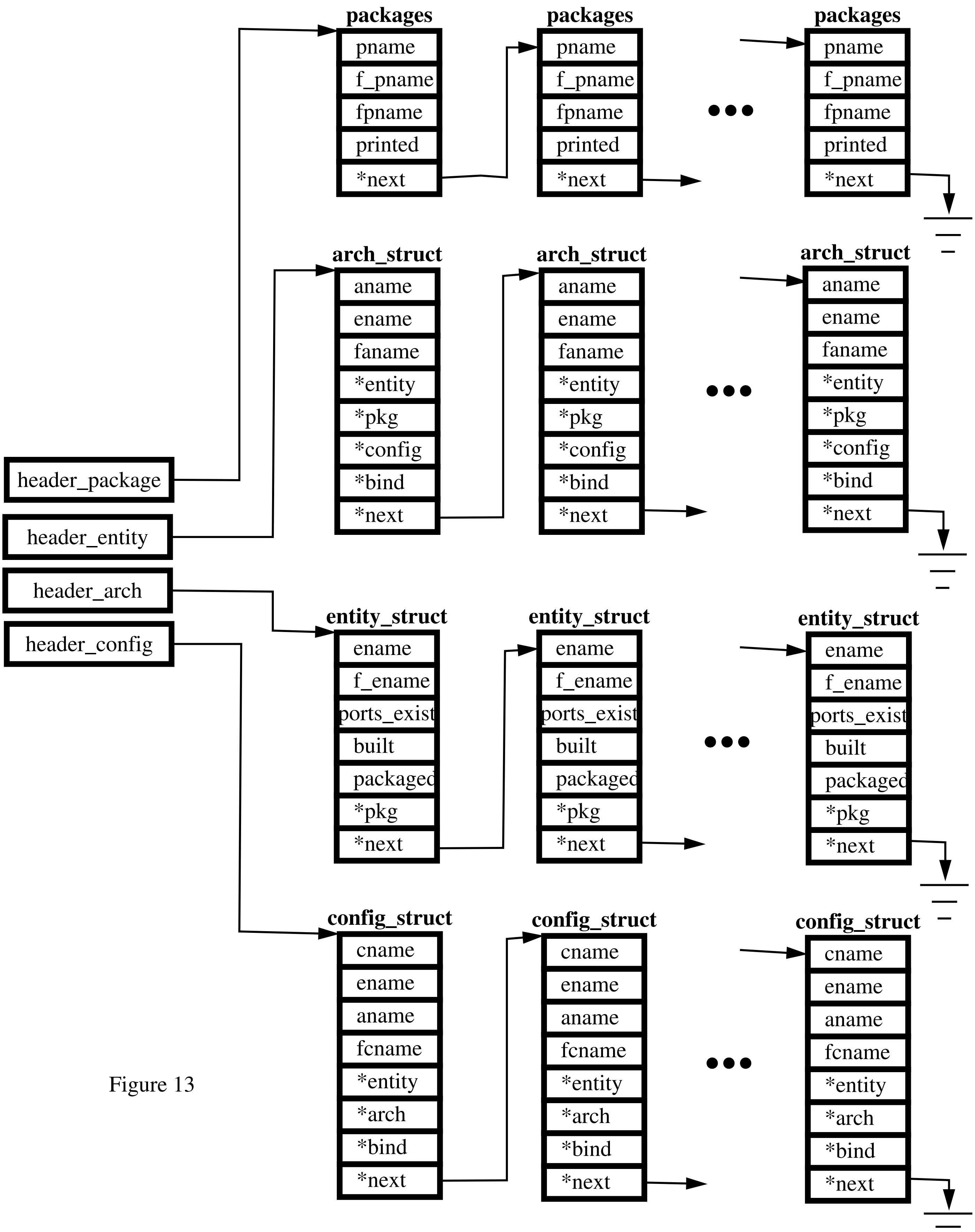
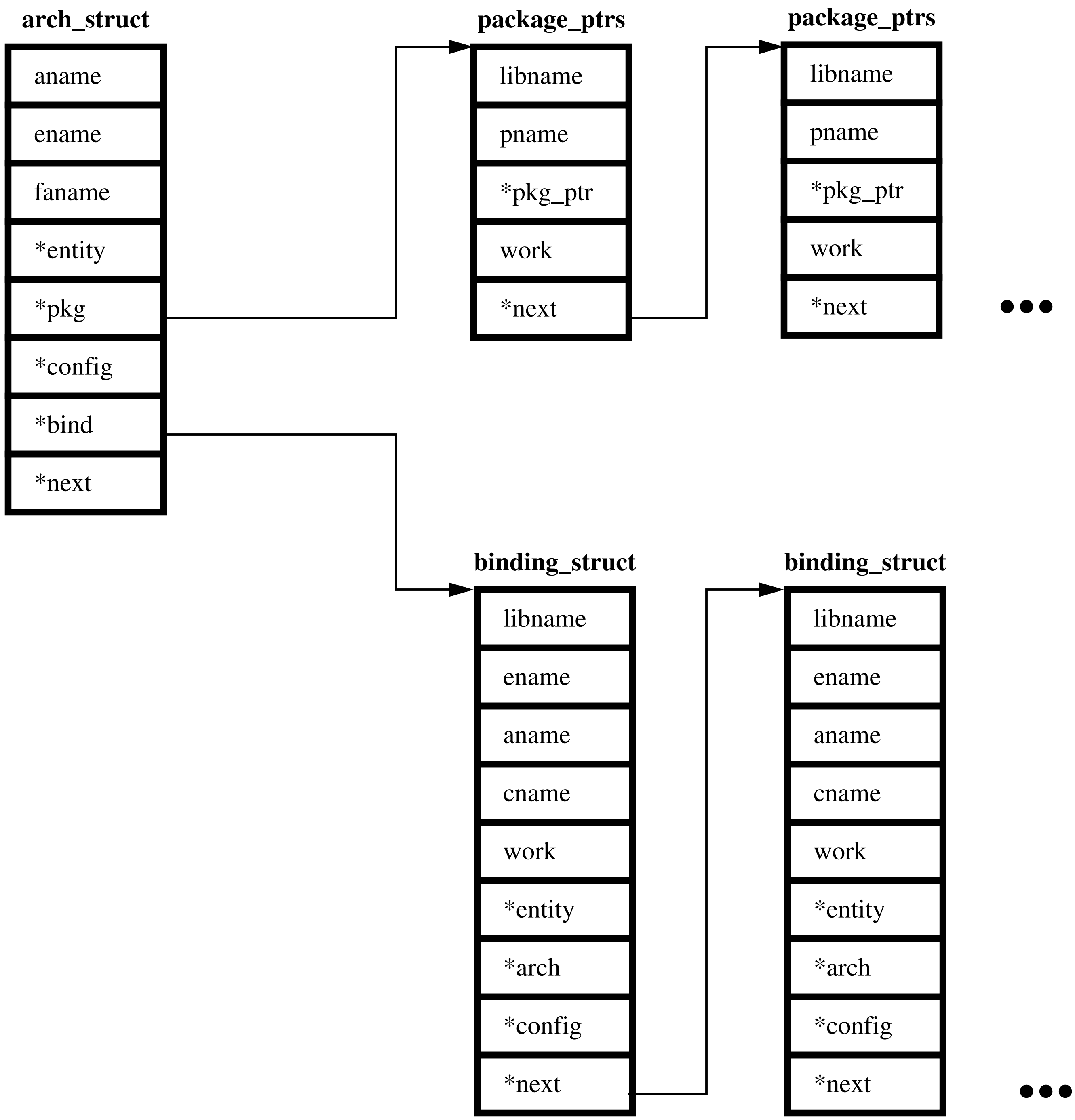
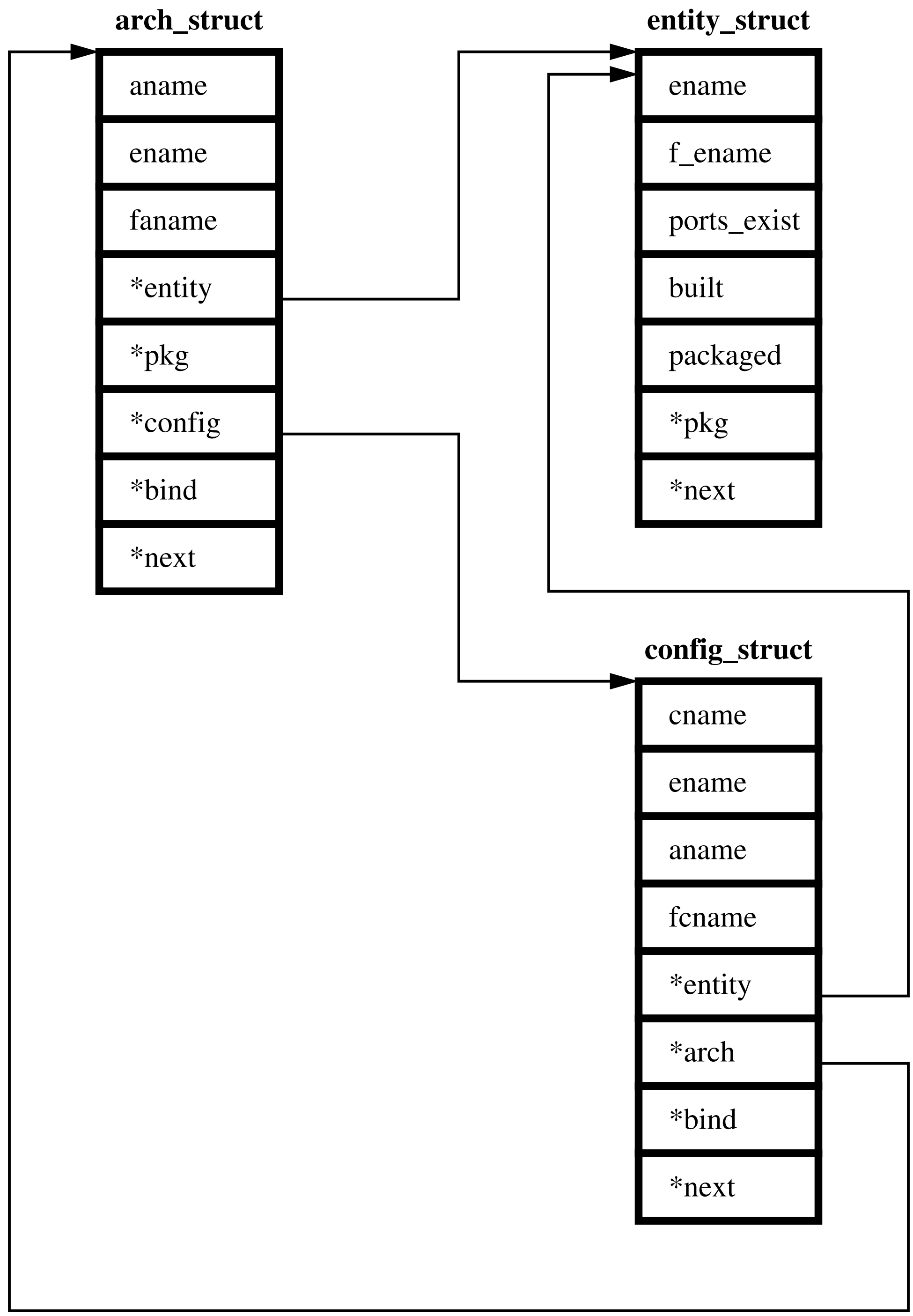

packages
pname
f_pname
fpname
printed
*next
packages
pname
f_pname
fpname
printed
*next
packages
pname
f_pname
fpname
printed
*next
arch_struct
aname
ename
faname
*entity
*pkg
*config
*bind
*next
arch_struct
aname
ename
faname
*entity
*pkg
*config
*bind
*next
arch_struct
aname
ename
faname
*entity
*pkg
*config
*bind
*next
header_package
header_entity
header_arch
header_config
entity_struct
ename
f_ename
ports_exist
built
packaged
*pkg
*next
entity_struct
ename
f_ename
ports_exist
built
packaged
*pkg
*next
entity_struct
ename
f_ename
ports_exist
built
packaged
*pkg
*next
config_struct
cname
ename
aname
fcname
*entity
*arch
*bind
*next
config_struct
cname
ename
aname
fcname
*entity
*arch
*bind
*next
config_struct
cname
ename
aname
fcname
*entity
*arch
*bind
*next

Figure 13



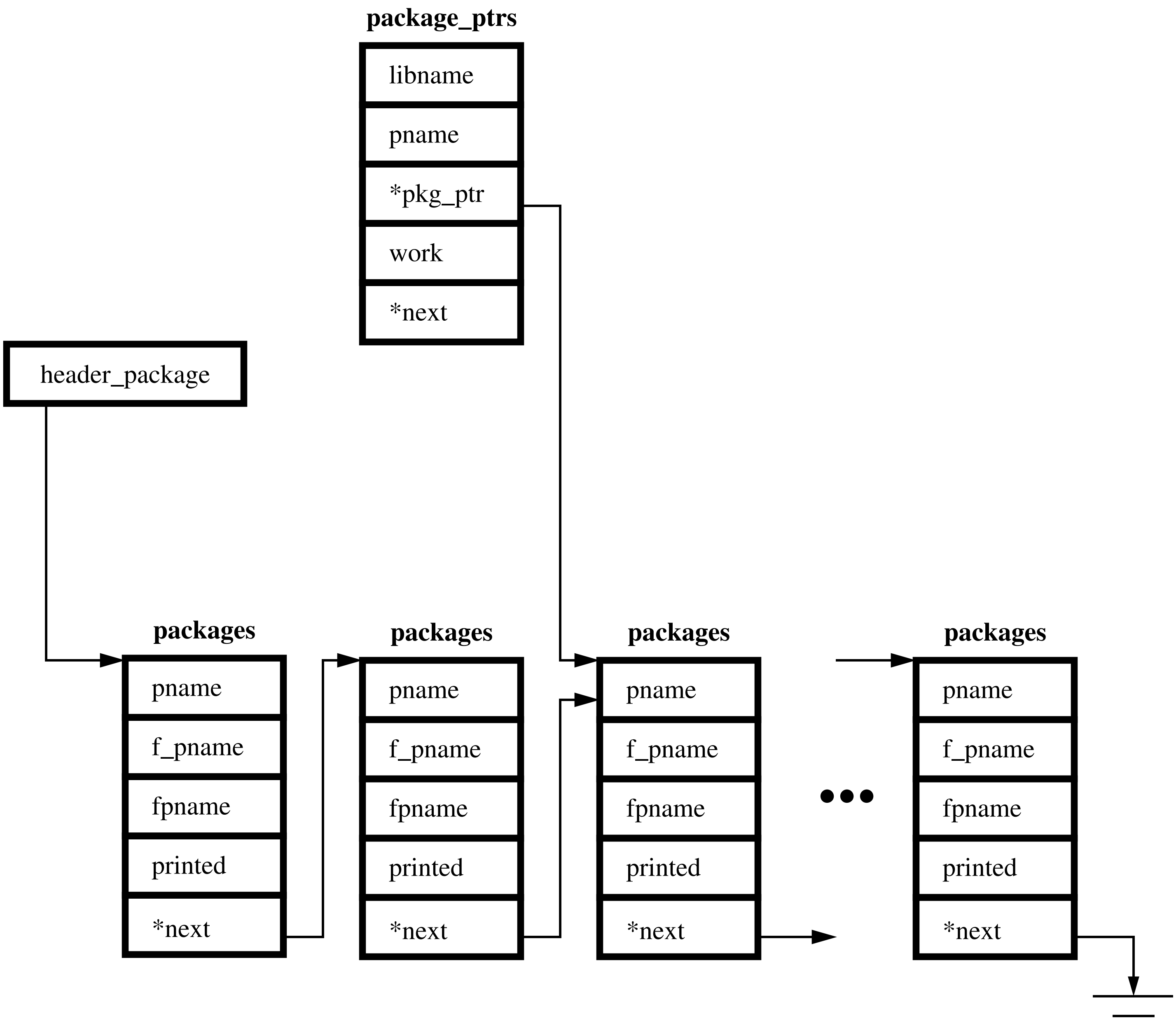
Linking up packages and binding structures

Figure 14



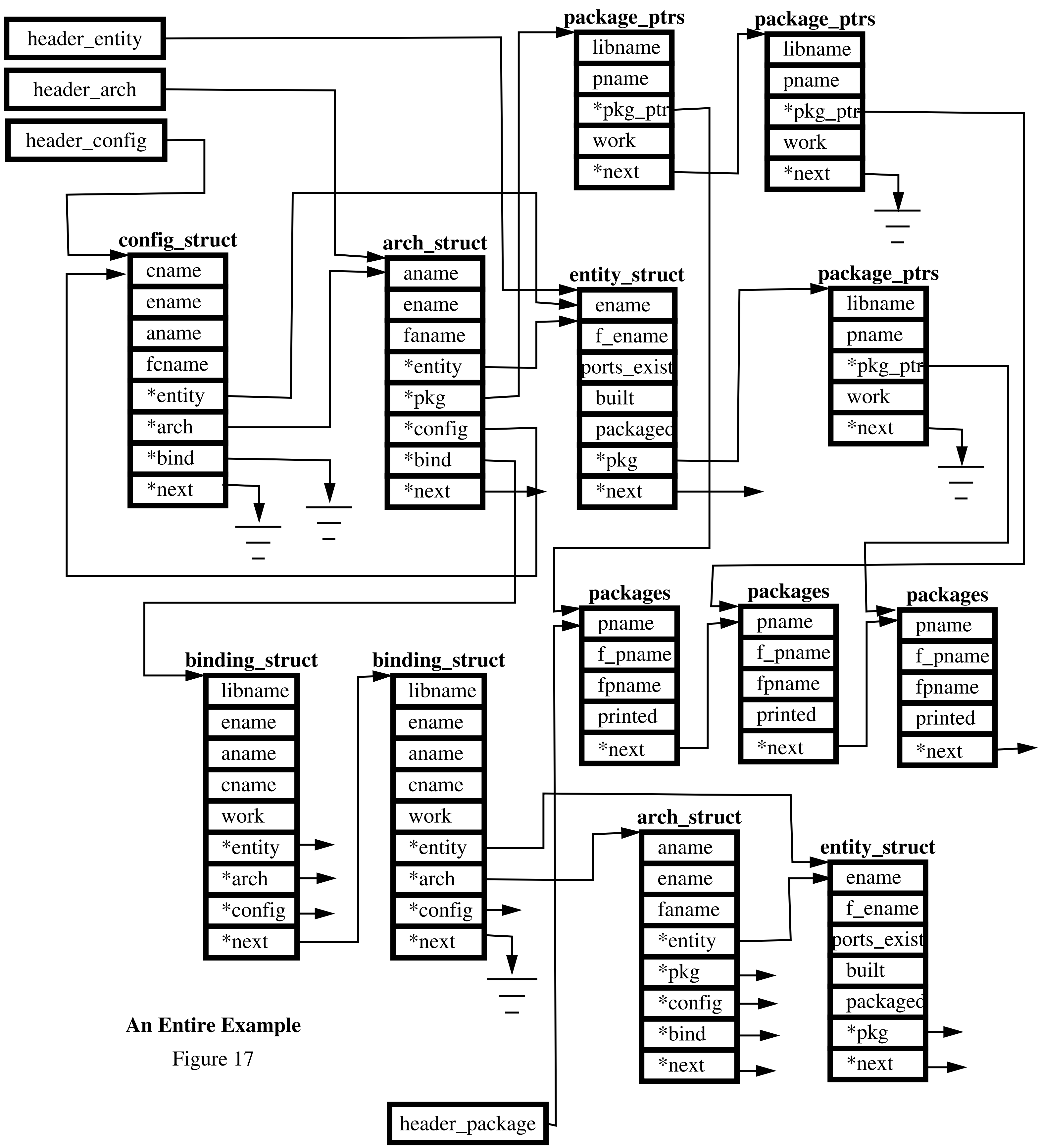
Back annotating entities, architectures, and configurations

Figure 15



Back annotating packages and package_ptrs

Figure 16



An Entire Example

Figure 17