opentext™

Dimensions CM

Dimensions CM Make Guide

Copyright © 2023 Open Text.

The only warranties for products and services of Open Text and its affiliates and licensors (“Open Text”) are as may be set forth in the express warranty statements accompanying such products and services. Nothing herein should be construed as constituting an additional warranty. Open Text shall not be liable for technical or editorial errors or omissions contained herein. The information contained herein is subject to change without notice.

Publication date: December 8, 2023

Table of Contents

Chapter 1

Guide to OpenText™ Dimensions CM Make

Overview of Dimensions CM Make

IMPORTANT! The current version of Dimensions CM Make does not support Dimensions CM streams. All references to Dimensions CM projects (for example, in the section "Integrating with Dimensions CM Versioning" on page 10) mean precisely the term "project" and do not encompass Dimensions CM streams.

A Dimensions CM product can have a physical representation in the form of a directory structure. Dimensions CM Make provides a mechanism for building configurations against the directory structure by way of Makefiles and offers a close integration of the Dimensions CM product's version management capabilities with the Make capability. This mechanism facilitates full version control and audit trails for the target files created by the build process and also embeds the necessary intelligence to determine whether a target file is out of date and therefore if it must be rebuilt.

The Dimensions CM Make approach is mainly intended for use in development environments where the use of Makefiles is standard practice. The migration time for these environments to Dimensions CM is minimized by virtue of the Dimensions CM Make facility.

This manual describes the additional functionality provided by Dimensions CM Make compared to GNU Make. For information on GNU Make, see the *GNU Make User Guide*.

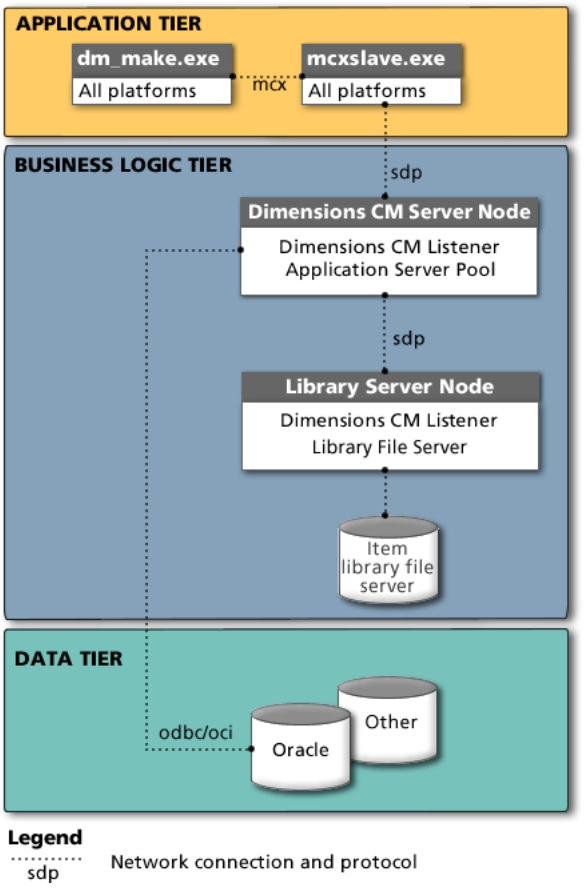
Dimensions CM Make Processes

Dimensions CM Make comprises two cooperating processes:

- A 'make client' executable, which performs the make processing. There are two 'make clients': `dm_make` (UNIX and Windows) and `dm_nmake` (Windows only). `dm_nmake` provides NMAKE compatibility.
- An agent process, which performs the Dimensions CM-related processing on behalf of the `dm_nmake`.

Dimensions CM Make Architecture

The following diagram illustrates the architecture of Dimensions CM Make:



The 'make client' executables are derivatives of GNU Make and are therefore upward compatible with the `make` described in Section 6.2 of "IEEE Standard 1003.2-1992" (POSIX.2).

To use Dimensions CM Make, you need to write a `makefile`. This is a file that describes the relationships between files in your product and the commands needed to update each file.

Using some features of Dimensions CM Make requires an additional file called a `pcmsfile`. This file contains rules for searching the Dimensions CM database and rules for preserving files made using Dimensions CM Make in the Dimensions CM database. A separate file is used for two reasons.

- Existing `makefiles` can be used unmodified. This simplifies integration of subcontracted software.
- The same `makefile` can be used for any configuration with the same physical structure.

Dimensions CM Make references both the files currently on disk and their corresponding versions in the Dimensions CM database when deciding what files need to be remade. This enables:

- Automatic get of outdated source files.
- Automatic get of files made by Dimensions CM Make and preserved in the Dimensions CM database.

Online Examples

If your Dimensions CM administrator chose the "Intermediate" (previously also called "Advanced" or "Payroll Sample") process model/workflow option when installing Dimensions CM, you have access to a demonstration Dimensions CM product called `payroll`. If you access this product, your default project is `payroll`.

Change to the project `PAYROLL:PRJ_BUILD`, which contains example files for `dm_make`.

NOTE `payroll` is a demonstration Dimensions CM product intended to show the Dimensions CM features and functionality with respect to a high degree of control and process in a regulatory environment. In addition to the Dimensions CM Make features, the `payroll` product includes a design part breakdown structure, various requests, and items for which baselines have been created.

Setting Up MCX_LISTEN for Multi-Homed Servers

See the *Installation Guide for Windows* or *Installation Guide for UNIX* for details.

Dimensions CM Make Terminology

The following expressions are commonly used throughout the rest of this guide.

target	A file which is to be made by Dimensions CM Make, or one which was previously made by Dimensions CM Make. This is the file substituted for the automatic `make` variable `$@`.
dependency	A file upon which a *target* depends. A dependency may itself be a *target*.
primary dependency	The first *dependency* listed in the `makefile`. This is the dependency substituted for the automatic variable `$<`.
controlled	Describes a file that is under Dimensions CM control. Dimensions CM Make considers a file to be *controlled* if it can map the path of the file to the project path of an item.
source	Describes a *controlled* file that has not been made by Dimensions CM Make.
stable	Describes a *controlled* file when Dimensions CM Make can match the file on disk to an item revision in Dimensions CM.
unstable	Describes a *controlled* file when Dimensions CM Make cannot match the file on disk to an item revision in Dimensions CM. A file can be considered *unstable* for several reasons: ■ The file on disk maps to a checked out item revision ■ The file on disk is writable ■ The file on disk is not writable, but no Item Revision can be found with the same file length and revised date or checksum.

uncontrolled	Describes a file that is not under Dimensions CM control. Dimensions CM Make considers a file to be *uncontrolled* if it cannot map the path of the file to the project path of an item.
preserved	Used to describe a target that has been put into the Dimensions CM database, creating a new Dimensions CM Item Revision.
Configuration Search Path	A sequence of one or more projects, release baselines or releases that are to be searched for files.
CSP	An abbreviation for Configuration Search Path.
current project	A project determined automatically by Dimensions CM Make when it is invoked. The *current project* is prepended to each Configuration Search Path and is the project in which all preserved targets are placed.

Invoking Dimensions CM Make

Dimensions CM Make is invoked by entering the command:

```
dm_make [ options ]
```

The options can include all those accepted by GNU Make as well as Dimensions CM Make specific options. All Dimensions CM Make specific options are multilateral. Their use is described where relevant and summarized on "Options Summary" on page 24.

Dimensions CM Make comprises the following cooperating processes:

- The `make` client executable, which performs the `make` processing.
- An agent process, which performs the Dimensions CM-related processing on behalf of the `dm_make` executable.

This division does not affect the way Dimensions CM Make is used, but does affect diagnostic output. Messages from the `make client` executable are prefixed with the executable name (for example, `pcmsmake`), while those from the agent process are prefixed with the string:

```
Dimensions CM Make Agent:
```

When Dimensions CM Make is invoked, it attempts to start an agent process unless the `--no-configuration-management` option is specified. This option is useful when you are debugging `makefiles`.

The agent process then attempts to connect to a Dimensions CM database, which can be specified in one of the following ways:

- Using the `--database=connection_description` command line option.

  The form of the `connection_description` depends on the underlying database being used by Dimensions CM.

- Using the base database symbol.

  The form of this symbol's value depends on the underlying database being used by Dimensions CM.

The `--database` option takes precedence if it is specified. If more than one `--database` option is specified, the last one on the command line is used.

If the base database symbol is not set, and a `--database` option is not found, Dimensions CM Make runs without reference to any Dimensions CM database, basing its actions solely on the files present on disk.

If the attempt to connect fails, Dimensions CM Make terminates, unless the `--keep-going` option has been specified.

NOTE Dimensions CM 10.1.2 and later: dm_make uses metadata to determine the stability or otherwise of inputs to a build. To revert to pre-10.1.2 functionality, you need to explicitly add "`DM_MKAGENT_NO_METADATA <any-value>`" to the Dimensions CM server **dm.cfg** configuration file.

Integrating with Dimensions CM Versioning

Dimensions CM Make uses a mechanism called a Configuration Search Path (CSP) to determine how to search the Dimensions CM database for files and where to put the files it has made. A CSP is a sequence of one or more projects. Dimensions CM Make obtains CSPs from the following sources:

- The current project.
- The `--configuration-path` option.
- The `pcmsfile`.

Dimensions CM Make uses the project directory and project filename of a file as the keys when searching a CSP. The Global Project (`$GENERIC:$GLOBAL`) allows different items to have the same project directory and project filename. It is the only project in Dimensions CM that allows this. Consequently, the Global Project cannot be used in a Configuration Search Path. Dimensions CM Make issues a warning whenever it encounters the Global Project in a CSP.

The Current Project

In Dimensions CM, every user can specify a default working directory for each project. This is done using the Dimensions CM Set Current Project (`SCWS`) operation without the `/NODEFAULT` option. Dimensions CM uses this directory when getting, checking out, and checking in files within a project.

Dimensions CM Make also makes use of the default working directory associated with each project. When you run Dimensions CM Make, the agent process attempts to determine the current project from its starting directory unless the project `--workset` and/or the `--workset-root` options have been specified. (This is either the current working directory or the directory resulting from any GNU Make `--directory` command-

line options.) It does this by matching the starting directory against the project default directories recorded for the current user in the Dimensions CM database.

CAUTION!

The absolute path of the starting directory is used to perform the match. If a relative path or, on UNIX platforms, one containing symbolic links, is specified when the `SCWS` operation is performed, the corresponding database record is ignored. To ensure that this does not happen, change directory to your intended project default directory and specify the operating-system directory in the `SCWS` command.

On Windows, you need to explicitly specify the operating-system directory to do this, for example:

```
dmcli scws PAYROLL:WS_MAKE /dir="c:\temp\make"
```

On UNIX, you can use the pwd command to set the operating system directory, for example:

```
dmcli scws PAYROLL:WS_MAKE /dir=\"`pwd`\"
```

in Bourne shell or C shell.

Dimensions CM Make matches the project or projects with a default directory that is the longest prefix of the starting directory.

If no project can be matched, Dimensions CM Make does not set the current project, and a warning is issued.

If more than one project can be matched to the starting directory, Dimensions CM Make selects the default project if it is one of those matched. The default project is the project specified in the most recent `SCWS` operation without a `/NODEFAULT` option. Dimensions CM Make issues a warning if it has to do this. If none of the projects matched the default project, Dimensions CM Make does not set the current project and issues a warning.

If the current project can be determined, a confirmation message is given.

Dimensions CM Make uses the relative path from the default directory of the current project to the starting directory to map relative paths used in the `makefile` to project paths. If the current project cannot be determined a one-to-one mapping between relative paths used in the `makefile` and project paths is used.

The current project is always used as the destination project when preserving files made by Dimensions CM Make. If it cannot be determined, no files can be preserved. It is also always the first project searched when looking for files, regardless of the content of the `pcmsfile` or the value specified for the `--configuration-path` option.

You can override this behavior using the project options:

- `--work-set=work-set-spec` and
- `--work-set-root=directory`.

If you do this, the starting directory must be a descendent of the specified root. If you use the `--work-set` option but not the `--work-set-directory` option, the starting directory is mapped to the project root. If you use the option `--work-set-directory` but not the option `--work-set`, the current user's default project is used as the current project.

The --configuration-path Option

The `--work-set` option is used to specify a fallback CSP. This CSP is used for two purposes:

- When Dimensions CM Make is invoked, this CSP is searched for a `pcmsfile` if the current project could not be determined or no `pcmsfile` could be found in the current project.
- After any `pcmsfile` is processed, this CSP is searched for any file that cannot be found in the current project or the CSPs specified in the `pcmsfile`.

The syntax of `configuration_search_path` is a sequence of one or more project specifications, separated by vertical bar (`|`) characters. Normal Dimensions CM quoting rules apply to the specifications (see the related document *Command-Line Reference* for details), with the addition that project specifications containing a vertical bar must also be quoted, for example:

```
--configuration-path='PROD1:FIX1 | "PROD1:SYSTEM SOURCE"'
```

The single-quotation characters (`'`) protect the vertical bar and double-quotation characters (`"`) from interpretation by the shell. No double-quotation characters are required around `PROD1:FIX1` because it does not contain either the space character or the vertical bar.

Problems with --configuration-path Option

Dimensions CM Make uses a one-to-one mapping between relative path and project path if there is no current project.

Consider a `makefile` that employs recursive makes in other directories, for example,

```
some_target:
   cd_directory
   $(MAKE)
```

If the `configuration_search_path` option is used, both the parent and child `makes` map their starting directory to the project default directory of the project(s) specified in the option, if no current project can be determined. This is unlikely to give the desired results.

We recommend not to use the `--configuration-path` option in these circumstances.

The pcmsfile

The `pcmsfile` contains rules that tell Dimensions CM Make where to search for source and target files and what values to use when preserving targets. These rules have a format similar to the rules in a `makefile`:

```
pattern [ pattern ... ] : [ pattern ... ]
   directive
   ...
```

The pattern line *must not* begin with whitespace. Directive lines *must* start with whitespace. This is consistent with `makefile` syntax.

Unlike a `makefile`, there must be at least one directive. The patterns are GNU Make style stem patterns.

Rules are matched by comparing the target file with the left-hand side pattern(s) and the primary dependency with the right hand side pattern(s), if specified. The rule is selected if there is a match. Once a rule has matched, the automatic variable `$@` may be used to refer to the target file in directives and the automatic variable `$<` may be used to refer to the primary dependency. The right-hand pattern match is only done when building the preservation rule base—it is not done when building the CSP.

For example, the following rule tells Dimensions CM Make to search for object files in Project `"PROD1:RELEASED C OBJECTS"` and to preserve object files made from C source files using item type `OBJ` and format `SUNOBJ`:

```
%.o: %.c
   configuration     "PROD1:RELEASED C OBJECTS"
   preserve          $TYPE = OBJ \
                     $FORMAT = "SUNOBJ"
```

Long lines may be broken by making the last character on the line a `'-'` or `'\'`. Line joining is done before all other processing.

The `'\'` character can be used to remove the special meaning of the next character. A `'\'` can be written as `'\\'`, but this is only necessary when the next character is a special character that is to retain its meaning. The special characters are `'!'`, `'#'`, `'-'`, `'\'`, and `'%'` (refer also to the *GNU Make User Guide*).

Either the `'#'` or `'!'` character can be used to start a comment. Unlike a `makefile`, the `'\'` character can be used to quote the comment character. The comment character and the remainder of the line are ignored. Because line joining is done before comment processing, you can use escaped new lines in comment lines too.

Naming pcmsfiles

The name of the `pcmsfile` can be specified through the `--pcms-file=<filename>` option. If this option is not used, Dimensions CM Make looks for a file named `pcmsfile` in its starting directory. If a file is not found, it repeats the search using the name `Pcmsfile`, if the operating system supports mixed case names. If neither file is found, the search is repeated in the directory that corresponds to the project default directory of the current project. Paths in `pcmsfiles` found by the default search mechanism are always interpreted relative to the project root. Paths in `pcmsfiles` specified with the `--pcms-file` option are interpreted relative to the Dimensions CM Make starting directory.

Specifying CSPs in the pcmsfile

CSPs are specified in the `pcmsfile` using the configuration directive:

```
configuration configuration_search_path
```

This directive sets up a configuration path that is used for files that match the target pattern of the rule. If more than one rule containing a configuration directive matches, the CSPs are traversed independently in the order in which they appear in the `pcmsfile`.

The target patterns should include the pattern `%/` if you want the CSP to be searched for directories.

The syntax of `configuration_search_path` is as follows:

```
disable-current-search = '-'
explicit-current-search = '.'
meta-element = disable-current-search | explicit-current-search
enable-global-search = '*'
disable-global-search = '!'
work-set-specifier = 'work-set' | 'ws'
baseline-specifier = 'baseline' | 'bl'
release-specifier = 'release' | 'rl'
object-class-specifier = work-set-specifier |
   baseline-specifier |
   release-specifier
object-specification = product-id ':' object-id
csp-element = [ object-class-specifier ] object-specification
csp = { csp-element | meta-element }
   [ { '|' { csp-element | explicit-current-search } } ... ]
   [ { enable-global-search | disable-global-search } ]
```

If the optional `object-class-specifier` is omitted, Dimensions CM Make behaves as if a `work-set-specifier` had been entered. This is essential to maintain upward compatibility with existing CSPs.

Dimensions CM Make searches for an object of the class and with the specification entered in the repository. If a matching object is found, the CSP element maps to that object.

Normal Dimensions CM quoting rules apply to the specifications (see *Command-Line Reference* for details), with the addition that project specifications containing a vertical bar must also be quoted, for example:

```
../src/%.c :
   configuration "prod:work set one" |\
      "prod:contains|bar"
```

This directive is very useful. For example, you can use different CSPs for source and object files, allowing a single portable source code configuration to be used with any number of architecture dependent object code configurations, for example:

```
%.o :
   configuration "fs:rudder fix 1" |\
   "fs:release 4 mc88000 fs objects"
%.cxx :
   configuration "fs:rudder fix 1" |\
   "fs:release 4 source"
```

The meta-elements are used to control when the current project is searched. `Disable-current-search` defers searching the current project until after all subsequent matching CSPs, including the fallback CSP, are processed. It can only be specified as the initial element of a CSP. The current project is still searched to ensure the project file namespace is complete. `Explicit-current-search` specifies where the current project is searched in a matching CSP. Specifying `explicit-current-search` after `disable-current-search` in a CSP overrides the `disable-current-search` behavior.

Until a matching CSP containing a meta-element is encountered, an implicit search of the current project is performed before the other elements in matching CSPs are processed. This maintains upward compatibility with existing CSPs. Consequently, `disable-current-search` is best used as either the first element in the fallback CSP or as the first element in a match-all configuration directive specified as the first configuration directive in the `pcmsfile`.

The `enable-` and `disable-global-search` elements, when enabled Dimensions CM Make perform a global project search after performing the fallback CSP and any deferred current project searches.

Searching Baselines and Releases

If an element in a CSP is a baseline, Dimensions CM Make agent exhibits the following behavior:

- It attempts to match the path requested by the Dimensions CM Make client against the path structure stored in the baseline.
- If no match is found, Dimensions CM Make exhibits the same behavior as it does when no match is found in a project.
- If a match is found, Dimensions CM Make updates the file as necessary in the same fashion as is done for updates with respect to a project. This means that built files recorded in the baseline are subject to the same reuse checks as they would be if they had been located in a project.

If an element in a CSP is a release, Dimensions CM Make maps the element to the baseline used to create the release and thereafter behaves as if that baseline had been specified as the CSP element.

Preserving Targets as Items

You can tell Dimensions CM Make to preserve targets it has made in the Dimensions CM database by using preserve directives in the `pcmsfile` to set up preservation data. Preserving the target either creates or updates a Dimensions CM item.

The `--no preserve` option can be used to direct Dimensions CM Make to omit any target preservation it would otherwise have performed. The `--preserve` option can be used to specify that some files are to be preserved as well as built files.

Dimensions CM Make permits preservation of a target only if the following apply:

- The `pcmsfile` being used is a stable, controlled file. If you want to preserve an unstable or uncontrolled `pcmsfile` as part of a build, ensure that it contains rules to preserve itself and use the `--preserve` option with the appropriate preservation class(es). This preserves the `pcmsfile` before any other targets are preserved.
- The `makefiles` used are reproducible. The `makefile` for the `make` instance you state from the command line must be a controlled and stable file. Any other `makefile` that is statically defined must be a controlled stable file. Dynamically generated `makefiles` (such as ADG output files) are considered reproducible if the `makefile` containing the rule used to build the file is a controlled, stable file. These criteria are applied recursively, so any `makefiles` built from such a dynamically built `makefile` are also considered reproducible.
- All other dependencies that are controlled files are also stable.

It is possible that some or all of the dependencies may be uncontrolled files. Dimensions CM Make still preserves the target in this case as long as the conditions above hold true.

The `preserve` directive has the following syntax:

```
preserve assignment [ assignment ... ]
```

Each `assignment` is of the form:

```
variable = value
```

The `variable` is either the variable name of a user defined attribute or one of the following item-related system-defined attributes:

```
$PRODUCT_ID              $ITEM_ID      $VARIANT   $TYPE
$FORMAT                  $STATUS       $OWNING_PART_ID
$OWNING_PART_VARIANT     $OWNING_PART_PCS
```

The attributes (part-id, variant and PCS) of the design part by which the item is owned are referred to here as `$OWNING_PART_ID`, `$OWNING_PART_VARIANT`, and `$OWNING_PART_PCS` respectively.

The value is a string specifying the value that should be assigned. When assigning values to user-defined attributes, the same syntax as the Dimensions CM command line is used to handle special characters and multi-valued attributes (see *Command-Line Reference* for details).

Existing attribute values can be substituted into these assignments using the following syntax:

```
$(variable file)
```

The `file` must be `$@` or `$<`. The `variable` is the variable name of a user-defined attribute or one of the system-defined attribute variables given above. If the `file` refers to an uncontrolled file or the attribute value is undefined, an empty string is substituted.

If `$@` is specified, the value is taken from the most recent revision of the item corresponding to the target in the configuration.

NOTE Unlike the configuration directive (which searches the various CSPs in order and independently) assignments in all matching preserve directives are merged. The merge order is given by the order of the rules in the `pcmsfile`. This makes it easy to write general preservation directives and then override specific values for certain groups of files.

The minimum set of attributes that must be provided in order to preserve a target by creating an item are shown in the following table. Dimensions CM Make calculates appropriate default values for all of these attributes except `$ITEM_TYPE`, which must always be specified.

Variable	Create/ Update/ Check In	Notes
`$PRODUCT_ID`	Create	If there is a primary dependency and it is an item, then `$($PRODUCT_ID $<)` is used. If this is not the case, the product that owns the current project is used.

`$ITEM_ID`	Create	If automatic ID generation is enabled for the item type being used to preserve the target, this variable is silently ignored.
		
		The default value is based on the target filename as follows:
		
		All non-alphanumeric characters are replaced by spaces. The resulting string is truncated to 25 characters if necessary. Trailing spaces are removed.
		
		Earlier releases of Dimensions CM Make used `$($ITEM_ID $<)` as the default. To retain this functionality, add the following rule to your `pcmsfile`. This rule should precede all other preserve directives that specify `$ITEM_ID`:
		
		`%:`
		`preserve $ITEM_ID = $($ITEM_ID $<)`

`$VARIANT`	Create	If there is a primary dependency and it is an Item, then `$($VARIANT $<)` is used. If this is not the case, the variant of the owning part is used.
`$TYPE`	Create	This must be specified. There is no default value.
`$FORMAT`	Create	If unspecified, the suffix of the target filename is used. If the target filename does not have a suffix, an error occurs.
`$OWNING_ PART_VARIANT`	Create	If there is a primary dependency and it is an Item, then `$($OWNING_PART_ID $<)` is used. If this is not the case, the root part of the owning product selected above is used.
`$OWNING_ PART_VARIANT`	Create	If there is a primary dependency and it is an Item, then `$($OWNING_PART_VARIANT $<)` is used. If this is not the case, the variant is that of the owning part selected above is used. If that part has more than one variant, an error occurs. If a variant is specified, but the owning part does not have such a variant, an error occurs.
`$LIB_FILENAME`	Create	Dimensions CM Make automatically generates a value for this variable, based on `$WSPATH` if it is not specified.

Variable	Create/ Update/ Check In	Notes
$STATUS	Create Update Check In	The initial lifecycle state is used if no value is specified.
$COMMENT	Create Update Check in	A comment stating that the item revision was created, updated or checked in by Dimensions CM Make is used if no comment is specified.
$DESCRIPTION	Check In	A description stating that the item revision was created by Dimensions CM Make is used if no description is specified.

Other item related inherent attributes may be used as values in assignments. Their values may not be changed through preserve directives. These are listed in the following table.

Variable	Create/ Update/ Check in	Notes
$REVISION		The item revision.
$OWNING_ PART_VARIANT		The PCS of the owning design part.
$WSPATH		This is the path relative to the project root. The value can only be used in specifying other values. It cannot be changed via a preserve directive.
$DIRPATH		This is the directory part of $WSPATH.
$FILENAME		This is the filename part of $WSPATH.

For example, consider a situation where all object files should be preserved with a common item type and format and be owned by the same design part as their primary dependency, except for object files in the directory lib/. These object files should be owned by the design part "CUSTOM LIBRARY.SUN" in the same product as the primary dependency.

The following rules accomplish this:

```
%.o :
   preserve          $TYPE = OBJ    $FORMAT = SUNOBJ \

   lib/%.o :
   preserve          $OWNING_PART_ID = "CUSTOM LIBRARY" \
                     $OWNING_PART_VARIANT = SUN
```

Because the assignments in all matching rules are merged, the actual values used for lib/%.o would be:

```
$TYPE = OBJ
$FORMAT = SUNOBJ
$OWNING_PART_ID = "CUSTOM LIBRARY"
$OWNING_PART_VARIANT = SUN
```

All other object files would use the values:

```
$TYPE = OBJ
$FORMAT = SUNOBJ
$OWNING_PART_ID = $($OWNING_PART_ID $<)
$OWNING_PART_VARIANT = $($OWNING_PART_VARIANT $<)
```

Obsolete Gotten Files May Appear Unstable

If an item is gotten at its initial lifecycle state and subsequently overwritten in Dimensions CM because the item type permits overwrite at the initial lifecycle state, Dimensions CM Make cannot recognize the file. Any such file is treated as an unstable (locally modified) file. If the `--no-clobber` option has been used, the file is not updated and no target built using the file as an input can be preserved as an item.

A warning message is generated when such a file is encountered, so that the condition may be detected.

This restriction does not apply to items that are inputs to targets preserved as items by Dimensions CM Make, including `makefiles` and `pcmsfiles`. The Dimensions CM core system prevents such items from being overwritten.

Recommendation

Item types used by items that are subject to this restriction should not allow over-writing at the initial lifecycle state. This is accomplished by setting the appropriate flag option for item types in the Administration Console.

Item Header Substitutions Can Cause Gotten Files To Appear Unstable

If `--no-clobber` is not used and an item revision contains substitution variables that can take different values for the same library file version, Dimensions CM Make may fail to recognize any gotten file corresponding to the item revision. Similarly, changes to these substitutions after the file has been gotten may not cause an update on rebuild. This second problem is independent of the `--no-clobber` option.

The following substitutions can be used safely:

%PP%	Product ID
%PI%	Item ID
%PV%	Item variant
%PT%	Item type
%PR%	Item revision
%PID%	Item specification
%PF%	Item revision format
%PIV%	Item revision file version
%PM%	Project filename

The following substitutions can cause Dimensions CM Make to consider a file unstable:

%PL%	Item history
%PLA%	Item history including action history
%PRT%	Date of last change
%PO%	Owner

%PS%	Status
%PIRC%	Related requests
%PIRP%	Related design Parts
%PIRB%	Related baselines
%PIRW%	Related projects
%PIRIT%	Items related to item as parents
%PIRIF%	Items related to item as children
%PIATF%	Expand the failure entries in the audit trail for an item revision.
%PIATS%	Expand the success entries in the audit trail for an item revision.
%VARIABLE%	Any user-defined attributes

Recommendation Do not use these substitutions in item revisions that are to be used with Dimensions CM Make.

Workaround Use the `--no-expand` option during product development to avoid files appearing unstable due to changes in item header substitution values.

Toggling Item Header Substitution Can Cause Gotten Files To Appear Unstable

Changing the item header substitution flag for an item type in the Administration Console may prevent Dimensions CM Make from recognizing files corresponding to items of that type that were gotten before the change.

Recommendation Once item header substitution has been enabled for an item type, it should never be disabled. If the unexpanded version of an item revision is required, a Get Item operation with the `/NOEXPAND` qualifier should be used.

Possible Problems When Preserving Files

Some compilation tools require write-access to the inputs. For example, BSCMAKE (a Windows development tool) requires write-access to the `.SBR` files it reads in order to truncate them after processing. If Dimensions CM Make has preserved these inputs, they are not writable.

Workaround Perform one of the following.

- Do not preserve these files.
- Modify the rules to make the input files writable before the command is invoked. In the case of `.SBR` files, it is recommended that they are not preserved as they are transient files used by BSCMAKE when building a source browser database.

Specifying Target Reuse

The network administration cluster in the Administration Console can be used to define a Resident Software definition (RSD) and associate it with nodes across that it is intended to reuse targets built by Dimensions CM Make.

If existing targets built by Dimensions CM Make are *not* to be reused after the introduction of RSDs, they must be marked with a *native* RSD for the node on which they were built before associating the real RSD with the node. The procedure to accomplish this is given below.

If Dimensions CM Make cannot find an RSD for the node on which it is invoked, it issues the diagnostic:

`Warning: No Resident Software Definition found for node (<node-name>)`

where (`<node-name>`) is the name of the node on which Dimensions CM Make is being invoked. In this case, any files preserved in the Dimensions CM database are considered to be native to the node on which Dimensions CM Make was invoked. They are reusable only on that node.

If Dimensions CM Make can find an RSD for the node on which it is invoked, it searches the Dimensions CM database for any existing native targets for the node and marks them as compatible with the RSD.

If the introduction of the RSD coincides with an incompatible change in the resident software, this behavior causes potentially incompatible files to be reused. Use the following procedure to avoid the problem:

1 Create a native RSD for the node and associate it with the node.

2 If the node is a UNIX system, log in to that node and enter the command:

```
dm_make -n -f /dev/null
```

3 If the node is a PC, log in to that node and enter the command:

```
dm_make -n -f nul
```

Existing native files are now marked with the native RSD. Once this has been done, associate the real RSD with the node. Subsequent invocations of Dimensions CM Make do not reuse the files marked with the *native* RSD.

There is a limitation on the reuse of uncontrolled files. These must have the same absolute path on all nodes where it is intended to reuse them.

Automatic Dependency Generation

Dimensions CM Make supports dynamic dependency generation through its capability to rebuild its own input files coupled with a dependency generation utility called `adg`. To employ dynamic dependency generation, you first add include statements to your `makefile` for dependency files related to the targets you want to make. For example:

```
OBJS = object1.o object2.o
                    # We want the dependencies for these to be
                    # generated dynamically
prog:       $(OBJS)
            linkprog $(OBJS)
include $(OBJS:.o=.dep)     # This line includes the dependencies
```

This causes Dimensions CM Make to try and include the files `object1.dep` and `object2.dep`. You can then add a rule to rebuild these files, say:

```
%.dep:     %.c
adg --language=C -output=$@ -- $(CFLAGS) -- $<
```

This causes the `adg` utility to run for the C language, interpreting the options supplied between the two `--` options as C compiler options. The advantage of using `adg` rather than a compiler option is that `adg` is not compiler-dependent. The `adg` utility is described in "Guide to ADG" on page 29.

Building Object Libraries and Other Containers

Dimensions CM Make must be informed if the target to be built is some type of container. This is necessary because different update and preservation strategies are required for these targets. You inform Dimensions CM Make that a target is a container in two ways as follows.

- Use library notation in your `makefile`. The container must be an object library in this case, for example:

  ```
  libtarget.a: libtarget.a(blue.o red.o gold.o)
  ranlib $@
  ```

NOTE A command is required for the target to be preserved as an item, even on systems where object library indices are maintained on each update. If you are using such a system, the simplest solution is to use "true" as a dummy command.

- Use the "container" directive in your `pcmsfile`, for example:

  ```
  %.a:
        container
  ```

  In this case, the container can be any type of file that contains elements that can be individually updated, for example:

  ```
  container:   element1 element2 element3
        update_container $@ $?
  ```

  This method supports the traditional approach to maintaining object libraries, for example:

  ```
  libtarget.a   blue.o red.o gold.o
        ar rv $@ $?
        ranlib $@
  ```

Each approach has its advantages. If you are using library notation, there is no need to explicitly create the object files as targets. They can instead be inferred as intermediate dependencies, for example:

```
# No need for explicit library indexing on this system
RANLIB = true

libtarget.a:   libtarget.a(blue.o red.o gold.o)
  $(RANLIB) $@

# Do not put in dependencies between the object files and their
# sources. This causes them to be inferred as intermediate
```

```
# targets required to build the library used. The object files
# are automatically deleted when Dimensions Make terminates.
# They are rebuilt only when the corresponding library
# member needs updating.

blue.o:   blue.h target.h
red.o:    red.h target.h
gold.o:   gold.h target.h

# You can still use your own object file suffix (or stem pattern)
# rule if you like. It is selected by implicit rule search
# when the object files are # inferred.

.c.o:
 $(CC) $(CDEBUG_FLAGS) $(CPP_FLAGS) $(CARCH_FLAGS) -c $<

# You can also use the standard POSIX approach of having a .c.a
# suffix rule. Note that you are responsible for deleting the object
# file in this case.

.c.a:
 $(CC) $(CDEBUG_FLAGS) $(CPP_FLAGS) $(CARCH_FLAGS) -c $<
 ar rv $@ $*.o
 rm -f $*.o
```

The advantage of this approach is that the object files are not retained, which can result in a considerable reduction in disk space requirements. The POSIX style approach is most efficient in terms of disk space requirements, as the object files are deleted as they are added to the library. The intermediate target approach does not remove the object files until Dimensions CM Make terminates, so the transient disk space requirement may be high.

The disadvantage of library notation is that the library must be rewritten each time a member is updated. This can be time consuming, especially with large libraries.

The converse is true with the traditional approach to library maintenance. The object files must be explicit targets in this case, so the disk space requirements are high. The advantage is that the library is updated in a single operation, which saves considerable time, especially with large libraries.

If you use the traditional approach and preserve the library in Dimensions CM, you should also preserve the object files. This is not essential, but is advised to avoid unnecessary rebuilds should the object files be deleted or another build area be used.

Building Double Colon Targets

Dimensions CM Make does not preserve double colon targets with no dependencies. These targets must be rebuilt whenever Dimensions CM Make is invoked, so preserving them would be inappropriate.

"Double colon" targets with dependency lists are preserved. Dimensions CM Make combines all the dependency lists into a single dependency list and concatenates the rules in the order they appear in the makefile to form a single process. This approach is used on the assumption that evaluating the rules with some set of dependencies and no target file

present has the same result as performing some subset of the rules with any previously built target file present. If you use a double colon target that does not conform to this assumption, it should not be preserved in Dimensions CM.

Options Summary

Dimensions CM Make extends the options understood by GNU Make with the following.

NOTE Type `"dm_make -v"` if you wish to check which version of Dimensions CM Make you are running and the version of GNU Make it was derived from; and type `dm_make -h` and/or refer to the *GNU Make User Guide* if you require information on the standard GNU Make options that are accepted by Dimensions CM Make.

`--no-configuration-management`
`--no-cm`
`--no-pcms`

Do not attempt to start the agent process and connect to Dimensions CM. This option is useful for debugging makefiles and enables Dimensions CM Make to be used as an ordinary `make`. The --xml-bom option also implies the --no-cm option.

`--database=connection_description`
`--db=connection_description`
`--pcmsdb=connection_description`

Connect to the Dimensions CM database specified by `connection_description`. If more than one of these options is specified, the last one on the command line is used. The form of `connection_description` depends on the underlying database being used by Dimensions CM.

`--configuration-path=configuration_search_path`
`--csp=configuration_search_path`

Use `configuration_search_path` as the fallback CSP. If more than one of these options is specified, they are concatenated in the order in which they appear on the command line.

The syntax of `configuration_search_path` is a sequence of one or more project specifications, separated by vertical bar (`|`) characters. Normal Dimensions CM quoting rules apply to the specifications (see the *Command-Line Reference* for details), with the addition that project specifications containing a vertical bar must also be quoted.

`--no-preserve`
`--no-save`

Do not preserve any targets in the Dimensions CM database.

`--work-set=<work-set-specification>`
`--ws=<work-set-specification>`

Use the project specified as the current project. If this option is used and `--work-set-root=<directory>` is not used, the starting directory is mapped to the project root.

`--work-set-root=<directory>`

`--ws-root=<directory>`

Map the directory specified to the project root. The directory must be an ancestor of the starting directory, or the starting directory itself. A relative path may be used and is interpreted relative to the starting directory. If this option is used and `--work-set=<project-specification>` is not used, the current user's default project is used as the current project.

`--pcms-file=<file-path>`
`--pf=<file-path>`

Use the specified file as the configuration rule file. If more than one such option is used, the files are read in sequence.

`--preserve[=<class-list>]`
`--save[=<class-list>]`

Preserve the specified classes of target in Dimensions CM. The `<class-list>` is a comma-separated list comprising one or more of the following values:

built	Targets built by Dimensions CM Make
extracted	Checked out files
unstable	Unstable source files
uncontrolled	Uncontrolled source files

If no `<class-list>` list is specified, the default is 'built'. If a `<class-list>` is specified, it must include all classes to be preserved.

If this option is specified, it overrides any `--no-preserve` option.

`--no-expand`

Do not expand items when they are gotten.

`--verbose`

Provides verbose output from the Dimensions CM Make agent.

`--no-clobber`

Do not overwrite read-only unstable files. By default, Dimensions CM Make overwrites such files.

`--no-dualsum`

Disables matching on both the text and binary lengthsum data when using Dimensions CM Make on Windows.

`--xml-bom[=<bom-file>]`

Causes a Dimensions Build compatible XML bill-of-materials (BOM) to be created. Specifying this option for `dm_make`, `dmnmake`, or `adg` implies the use of the --no-cm option, that is, only files on disk are considered and *no communication* with the Dimensions CM server occurs. In this mode, the Dimensions CM database is not accessed directly by Dimensions CM Make, instead Dimensions Build is responsible for populating the build area before the build and for collecting build outputs once the build is complete.

The default value for `<bom-file>` is `bom.xml`.

"Sub-makes" update their parent's BOM, that is, there is a single BOM for each "top-level" dm_make invocation.

Improving Dimensions CM Make Efficiency

When `dm_make` searches for a file, it passes a request to the *agent process* that may cause a Dimensions CM database search to be performed. The efficiency of Dimensions CM Make can therefore be improved by:

- Reducing the number of Dimensions CM database searches.
- Reducing the length of each Dimensions CM database search.

The number of database searches can be reduced by reducing the number of implicit rule searches done by dm_make. This can be done in the makefile as follows.

- Using the .PHONY special target

  The `.PHONY` special target should be used to explicitly state which targets do not correspond to real files. This prevents an implicit rule search for their dependencies.

- Limiting the number of extensions

  The `.SUFFIXES` special target can be used to reduce the number of implicit rules that is searched. This is done as follows:

  ```
  .SUFFIXES:                    # Clear the suffix list
  .SUFFIXES:   .o .c .h         # Set the suffix list
  ```

  After this has been done, only implicit rules involving the suffixes in the list are searched.

- Always explicitly stating dependencies

  For example, if you are making an executable and your object files are built from C sources, put in an explicit dependency between each object file and the C source file, rather than leaving Dimensions CM Make to infer it.

- Disabling implicit rule searching

  Implicit rule searching can be disabled by using the `--no-builtin-rules` option. If this is done, replacements for any implicit rules that the `makefile` uses must be written. Two things can make this task easier:

  - Use the `--print-data-base` option to obtain the text of the built in rules that would normally be searched
  - Use the include directive so that the rules can be maintained in a separate makefile.

The length of each Dimensions CM database search can be reduced by making sure that Configuration Search Paths are kept as short as possible. This is most easily accomplished by minimizing the number of projects from which files are to be obtained. In practice, most tasks can be accomplished with no more than four projects in any CSP as listed below.

- A target project, where all targets are to be preserved. This must be the current project because that is the only place preservation can take place.
- A source project for any changes being worked on.

- A project containing preserved targets for the system against which any changes are being made.
- A project containing the source for the system against which any changes are being made.

Behavior of Dimensions CM Make Under Windows

- All diagnostics are written to standard output

  Because of the limited capabilities for output redirection in Windows, all information, error and warning messages from Dimensions CM Make are written to the standard output stream. This differs from the behavior on UNIX, where errors are written to the standard error stream.

- Pathnames in makefiles and projects

  Within a `makefile`, either `/` (forward slash) or `\` (backslash) can be used as the separator in a path. Comparisons with project pathnames are case insensitive.

- Paths in configuration files

  Rule matching in the configuration file is case insensitive.

- Command interpreter selection

  The UNIX version of Dimensions CM Make always sets `SHELL` to `/bin/sh` unless this is explicitly overridden in the makefile. The Windows version of Dimensions CM Make does not set `SHELL`. Instead, `COMSPEC` is inherited from the environment unless explicitly set in the `makefile`.

- Use of command interpreter

  Dimensions CM Make uses one command interpreter per set of rule commands, not one command interpreter for each command. This may cause problems in `makefiles` that rely on target evaluation order to:

  - operate in the correct directory, or
  - give environment variables different values.

- Default rule base differs from UNIX

  The default rule base used by Dimensions CM Make for Microsoft Windows is written to support Microsoft, rather than UNIX, tools. You can query this rule base online by entering the command:

  ```
  dm_make --no-cm -p -f nul
  ```

  This writes the rule base to the standard output.

Comparing Behavior of Dimensions CM Make to GNU Make

- Recognition of dependency and command lines

Some editors on PC compatibles automatically expand tabs to spaces. This can result in a syntactically incorrect `makefile`. To avoid the issue, Dimensions CM Make follows the behavior of other make processors for Windows by interpreting any line starting with whitespace as a command line, rather than just those starting with a tab character. Specifying the special `.POSIX` target as the first non-comment line of a makefile disables this behavior.

- Prerequisite Inheritance

  Dimensions CM Make tracks modified prerequisites recursively. For example, you can specify the prerequisites for an object file as:

  ```
  example.obj : example.c
    example.c : example.h
  ```

  rather than

  ```
  example.obj : example.c
    example.obj : example.h
  ```

Limitations in NMAKE Compatibility

Dimensions CM Make doesn't implement some features of NMAKE, and there are a few other minor differences:

- The `A`, `B`, `C`, `NOLOGO`, and `X` options are accepted but ignored

  Dimensions CM Make always writes to standard output, so the X option is effectively fixed as `/X-`

- Functioning of the `!CMDSWITCHES` pre-processor directive

  This directive sets in `MAKEFLAGS` only those flags that are meaningful to Dimensions CM Make. This should not affect operation.

- Use of command interpreter

  Dimensions CM Make uses one command interpreter per set of rule commands, not one command interpreter for the whole session. This may cause problems in `makefiles` that rely on target evaluation order to:

  - operate in the correct directory, or
  - give environment variables different values.

- Return code differences

  Dimensions CM Make uses the following return codes which are different from those used by NMAKE.

0	Success	2	An error has occurred
1	Target requires update (/Q)	3	An interrupt was received

Chapter 2

Guide to ADG

Overview of ADG

ADG is an Automatic Dependency Generator. It employs internal rules to obtain source file dependencies. It produces output in a form suitable for use with the `make` utility.

Full C and C++ preprocessor support is provided. This allows `include` statements to be detected and added to the dependency list.

ADG is integrated with Dimensions CM Make. If Dimensions CM Make is available, the Dimensions database is searched for input files and `include` files in addition to the system filestore.

Invoking ADG

The command line syntax of ADG is:

```
adg [ adg-options] -- [ language-options] \
  -- input-file ...
```

The `adg --options` are described below. The `language-options` are those options you would usually pass to a translator for the language being processed. These are essential in languages where a macro preprocessor can change the dependencies in a file. C and C++ languages are common examples of such languages.

Specifying the Inputs

Specifying the Language being Processed

You use the `--language=language_name` option to tell ADG which language you are processing. If you do not specify this option then ADG attempts to infer it from the first input-file name. All input files must therefore be in the same language.

ADG assumes the C language for files ending in `.c` and the C++ language for files ending in `.cpp`, `.cxx`, and `.c++`. If the operating system differentiates filenames by case, files ending in ',' are also treated as C++.

Specifying the Initialization File

Because ADG is intended to run in many environments, it needs a source for system definitions. This is provided by the initialization file. ADG looks for a file called `adg_language_name.ini` by default. You can specify a different name by using the `--init-file=path` option.

Specifying the Outputs

ADG produces one or more dependency files as its output. How these files are produced depends on the target or targets specified the output control options and how the input files are specified.

Specifying the Target

In order to produce a dependency file, ADG needs to know the name of the target the dependencies are being generated for. This can be specified using the `--target=pattern` option. The pattern can be a Dimensions CM Make style stem pattern or a single path. If a single path is specified, the target is that file, and a single dependency file is produced. If a pattern is specified, the `--primary=pattern` option must be used to map the `%` in the target pattern onto the input filename. A dependency file is produced for each input file that matches the pattern, using the corresponding target.

If you do not specify a target, ADG assumes the target has the same path as the first input file name, less any suffix.

Specifying the Dependency Filenames

If generating a single file, the default behavior of ADG is to produce a file in the same directory as the first input file, using the name of that input file but with any suffix replaced by `.md`. This can be overridden in the same fashion as the target and primary names by using the `--output=pattern` option.

ADG always adds the dependency file to the targets in the dependencies it generates. This ensures that the dependency file is regenerated if any of the dependencies change.

Ignoring Errors

The ADG client supports a `--ignore-errors` option. If specified, ADG generates a warning when it fails to parse an input file, rather than an error.

Ignoring System Include Files

The ADG client supports a `--ignore-system-includes` option. If specified, ADG ignores the inclusion of system header files (`#include <file.h>`) when parsing the code. This option can greatly reduce processing and improve performance.

Examples

- Process a single C source file called `example.c`, producing the dependency file `example.md` for the target `example.o`:

  ```
  adg --target=example.o -- -- example.c
  ```

- Process several C++ source files in directory `src`, creating a dependency file for each source file in the directory *`deps`* with the corresponding target in directory `obj`:

  ```
  adg --target=obj/%.o --primary=src/%.cpp -- \
       output=deps/%.md -- \
       src/file1.cpp src/file2.cpp src/file3.cpp
  ```

  Considering `src/file1.cpp`, this creates the dependency file `deps/file1.md` for the target `obj/file1.o`.

Typical ADG Usage

Writing ADG Initialization Files

The initialization file depends on the C/C++ compiler you are using. This file needs to contain the following information:

- The Default `include` Path for the Compiler

  For UNIX systems, this is typically `/usr/include`. However, many compilers use their own specific 'include' directories. On Windows, the default `include` path usually depends on where the compiler was installed.

- Predefined Preprocessor Macros

  These macros depend entirely on the compiler used and may depend on the flags passed to the compiler. For example, some UNIX C compilers that parse both "classic" and ANSI standard C define the macro "unix" in 'classic' mode, but not in ANSI mode.

  ADG cannot handle option-dependent predefined macros. You need different initialization files. See "Rule Writing" on page 35 for information on handling these in your `makefile`.

There are three ways to obtain this information:

1. Work it out from the compiler documentation.
2. Use the compiler's dry-run facility if it exists.
3. If [1] and [2] fail, contact the compiler vendor's support team and ask them.

For example, if you are compiling C++ on IBM AIX you might use:

```
xlC -# somefile.c
```

Note that with this particular compiler `somefile.c` does not have to exist for the example to work. This produces several lines of output, including the list of predefined symbols:

```
-D_AIX
-D_AIX32
-D_AIX41
-D_IBMR2
-D_POWER
```

This particular compiler does not provide `include` path information, but the '`include`' paths are mentioned in the compiler's user documentation:

```
-I/usr/lpp/xlC/include
-I/usr/include
```

Once you have the information, you can write the initialization file which has the format:

```
{option-lines}
{preprocessor-lines}
```

The `{option-lines}` section can contain `-D`, `-U`, and `-I` options. More than one option can be placed on a line. Comments are not allowed.

The `{preprocessor-lines}` section can contain any preprocessor directive and comments may be used.

The information for `xlC` on IBM AIX could therefore be stated as:

```
-I/usr/lpp/xlC/include
-I/usr/include
/* Predefined symbols */
#define   _AIX
#define   _AIX32
#define   _AIX41
#define   _IBMR2
#define   _POWER
```

By default, ADG looks for `adg_c.ini` when processing C and `adg_c++.ini` when processing C++, but you can call the initialization file anything you like. For this example, the name `aix41_xlC.ini` is appropriate.

Updating Your *makefile* to Use ADG for Dependency Maintenance

This updating operation comprises the following steps.

- Removing the existing dependency lines for your C and C++ source to object dependencies. These are redundant after the conversion is complete.
- Adding 'include' directives for the dependency files you intend to create. This can be done by using one `include` per file or by writing one or more macros to represent the C and C++ derived object files and writing `include` statements based on these macros.
- Adding one or more rules to generate the dependency files. You also need to add the suffix you have chosen for the dependency files to the `.SUFFIXES` special target.

As an example, consider the following `makefile` for a simple program built from C and C++ sources:

```
.SUFFIXES: .o .c .C

# Redefine the compiler names for AIX.

CC = xlc
CXX = xlC
prog:         c_object1.o c_object2.o cxx_object1.o \
              cxx_object1.o cxx_object2.o cxx_object3.o
   $(CXX) -o $@ $^
c_object1.o: c_object1.c
c_object2.o: c_object2.c
cxx_object1.o: cxx_object1.C
cxx_object1.o: cxx_object1.C
cxx_object2.o: cxx_object2.C
cxx_object3.o: cxx_object3.C
```

Proceed as follows.

1 Remove the object file dependency lines:

```
.SUFFIXES: .o .c .C

# Redefine the compiler names for AIX.

CC = xlc
CXX = xlC
prog:      c_object1.o c_object2.o cxx_object1.o \
           cxx_object1.o\ cxx_object2.o cxx_object3.o
   $(CXX) -o $@ $^
```

2 Put the object files into macros and add the `include` statements. Use separate macros for the C and C++ objects, as you need different rules to build the dependency files:

```
.SUFFIXES: .o .c .C

# Redefine the compiler names for AIX.

CC = xlc
CXX = xlC

# Object file lists, per compiler.

C_OBJ =          c_object1.o c_object2.o
CXX_OBJ = cxx_object1.o cxx_object2.o cxx_object3.o

prog:              $(C_OBJ) $(CXX_OBJ)
   $(CXX) -o $@ $^

# You can use any suffix, but ADG uses .md by default.

include $(C_OBJ:.o=.md)
include $(CXX_OBJ:.o=.md)
```

3 Add the rules and update the `.SUFFIXES` special target:

```
.SUFFIXES: .o .c .C .md

# Redefine the compiler names for AIX.

CC = xlc
CXX = xlC

# Object file lists, per compiler.
C_OBJ =          c_object1.o c_object2.o
CXX_OBJ =        cxx_object1.o cxx_object2.o cxx_object3.o

prog:            $(C_OBJ) $(CXX_OBJ)
                 $(CXX) -o $@ $^

# You can use any suffix, but ADG generates .md
# files by default.

include $(C_OBJ:.o=.md)
include $(CXX_OBJ:.o=.md)
# Dependency file rules, again per compiler. This allows
# different initialization files to be used. Note that
# the compiler options are also passed to ADG. This
```

```
# allows -I, -D and -U options in the compiler command to
# be taken into account when producing the dependencies.

%.md: %.C
   rm -f $@
   adg          --target=$*.o \
                --init-file=aix41_xlC.ini \
                --output=$*.md \
                -- $(CXXFLAGS) -- $<
# It is assumed that a C compiler initialization
# file has also been produced.

%.md: %.c
   rm -f $@
   adg          --target=$*.o \
                --init-file=aix41_xlc.ini \
                --output=$*.md \
                -- $(CFLAGS) -- $<
```

Your `makefile` is now converted to use automatic dependency maintenance.

Rule Writing

The example above puts the dependency files in the same directory as the sources and also assumes that the object files are in the same directory. This is often not the case. For example, you may want to put the dependency files in their own sub-directory. This can be handled by adjusting the include statements and rules:

```
include $(addprefix md/, $(C_OBJ:.o=.md))

md/%.md: %.c
   rm -f $@
   adg          --target=$(<F).o \
                --init-file=aix41_xlc.ini \
                --output=$*.md \
                -- $(CFLAGS) -- $<
```

The file `obj/dynamic.mak` in project PRJ_BLD of the `PAYROLL` demonstration product includes rules which cope with separate source, object and dependency file directories should you need to use the whole repertoire.

Another common problem is compiler flags which change the list of predefined symbols, requiring either a different initialization file or adjustment of the preprocessor related flags passed to ADG. A typical use of this is to undefine the `__STDC__` macro which ADG defines as `1` by default when processing C language files. There are two approaches to this described below, both based on the "findstring" function of Dimensions CM Make.

Approach 1

- Maintain macros containing the relevant sets of `-D`, `-U`, and `-I` options.
- Use conditional directives based on the compiler flags to select the correct macro value.
- Pass the macro as an additional language-dependent parameter to ADG.

Assuming the IBM AIX we are using as an example, a suitable `makefile` fragment to determine whether __STDC__ is defined as listed below assuming the compiler is invoked as `xlc`, rather than one of the variant specific names:

```
ifeq "$(findstring -qlanglvl=, $(CFLAGS))" ""
  # Default: __STDC__ is defined as 1
  ADG_STDC =
else
  # There is a -qlanglvl option. Check it does
  # not reinforce the default
  ifeq "$(findstring -qlanglvl=ansi,
  $(CFLAGS))" ""
     ADG_STDC = -U__STDC__
  endif
endif
```

The rule for generating dependency files from C source is changed to:

```
%.md: %.c
  rm -f $@
  adg        --target=$*.o \
             --init-file=aix41_xlc.ini \
             --output=$*.md \
             -- $(CFLAGS) $(ADG_STDC) -- $<
```

This means the required `-U` option gets passed when needed.

Approach 2

Base the name of the initialization file on the compiler options. This uses a similar `makefile` extract to modify the filename for the initialization file, for example:

```
ifeq "$(findstring -qlanglvl=, $(CFLAGS))" ""
  # Default: __STDC__ is defined as 1
  ADG_STDC = ansi
else
  # There is a -qlanglvl option. Check it does
  # not reinforce the default
  ifeq "$(findstring -qlanglvl=ansi, $(CFLAGS))" ""
     ADG_STDC = classic
  else
     ADG_STDC = ansi
  endif
endif

%.md: %.c
  rm -f $@
  adg   --target=$*.o \

        --init-file=aix41_xlc_$(ADG_STDC).ini \
        --output=$*.md \
        -- $(CFLAGS) -- $<
```

Use whichever approach is simplest in your configuration.

ADG Options

Controlling the Set of Generated Dependencies

`--target=<pattern>`

Specifies a pattern for the target.

`--primary=<pattern>`

Specifies a pattern for the primary dependency.

`--output=<pattern>`

Specifies a pattern for the generated dependency file.

`--controlled-only`

Specifies that only dependencies that are under Dimensions control are to be included in the generated dependency list. This option is ignored if `--no-configuration-management` is specified.

`--link-to-target`

Specifies that an additional dependency making the target dependent on the generated dependency file. This is the default if `--no-preserve` or `--no-configuration-management` is specified. This additional dependency prevents missed rebuilds when controlled files with modification times older than previously built uncontrolled targets are gotten during dependency analysis. It is inadvisable to use this option when preserving files, as a spurious dependency on the generated dependency file is recorded in the Dimensions database.

`--no-link-to-target`

Specifies that the additional dependency described under `--link-to-target` above is not to be added to the generated dependency file. This is the default if neither `--no-preserve` nor `--no-configuration-management` is specified.

Specifying Additional Inputs

`--include-path=<path-string>`

Allows an additional include path to be specified. The syntax of `<path-string>` is a list of directories separated by a colon (`:`) on UNIX or a semicolon (`;`) under Windows.

`--include-symbol=<symbol-name>`

Allows an additional include path to be specified. The value of the environment variable `<symbol-name>` must have the same syntax as the `<path-string>` for the `--include-path` option.

NOTE Include paths specified with either the `--include-path` or `--include-symbol`. A command is required for the target to be preserved as an item, even on systems where object library indices are maintained on each update. If you are using such a system, the simplest solution is to use "true" as a dummy command.

Specifying Language and Compiler Details

`--init-file=""`

Specifies the initialization file. The default is `adg_<language-name>.ini`, where language-name is either `"c"` or `"c++"`.

`--language=<language-name>`

Specifies the language. Valid values for `<language-name>` are `"c"` and `"c++"`.

Controlling Dimensions Access

`--no-configuration-management, --no-cm, --no-pcms`

Specifies that ADG should not connect to a Dimensions database.

`--no-expand`

Specifies that any dependency files gotten from Dimensions should not be expanded.

`--no-preserve, --no-save`

Specifies that the target referred to by the generated dependency file is not preserved in Dimensions by a subsequent Dimensions CM Make invocation. This option affects the default set of dependencies generated (refer to `--link-to-target` on page 37). ADG does not update the Dimensions database itself.

`--preserve=<preserve-classes>, -save=<preserve-classes>`

Specifies that the target referred to by the generated dependency file is preserved in Dimensions by a subsequent Dimensions CM Make invocation. This option affects the default set of dependencies generated (see `--link-to-target` on page 37). ADG does not update the Dimensions database itself. The `<preserve-classes>` value can be any string, so values valid for Dimensions CM Make are acceptable. ADG only uses the presence of the option to determine its behavior. The value of the option is irrelevant.

`--work-set-root=<path>, --ws-root=<path>`

Specifies which directory is to be mapped to the project root. The default is calculated in the same manner as Dimensions CM Make.

`--work-set=<work-set-spec>, --ws=<work-set-spec>`

Specifies which project is to be mapped to the current project. The default is calculated in the same manner as Dimensions CM Make.

Miscellaneous Options

`--version`

If specified, ADG prints its version and exit without performing any other action.

`--configuration-path=<path>, --csp=<path>`

Specifies the fallback configuration search path. The syntax and semantics for this option are the same as the Dimensions CM Make `--configuration-path` option.

`--ignore-errors`

If specified, ADG generates a warning when it fails to parse an input file, rather than an error.

`--xml-bom[=<bom-file>]`

See "--xml-bom[=<bom-file>]" on page 25.