A Technical Overview of the Alcatel-Lucent nmake Product Builder

ADVANCED SCALABLE SOFTWARE BUILD TOOL

Alcatel-Lucent Bell Labs

nmake@alcatel-lucent.com

http://www.bell-labs.com/project/nmake

Alcatel-Lucent nmake Product Builder

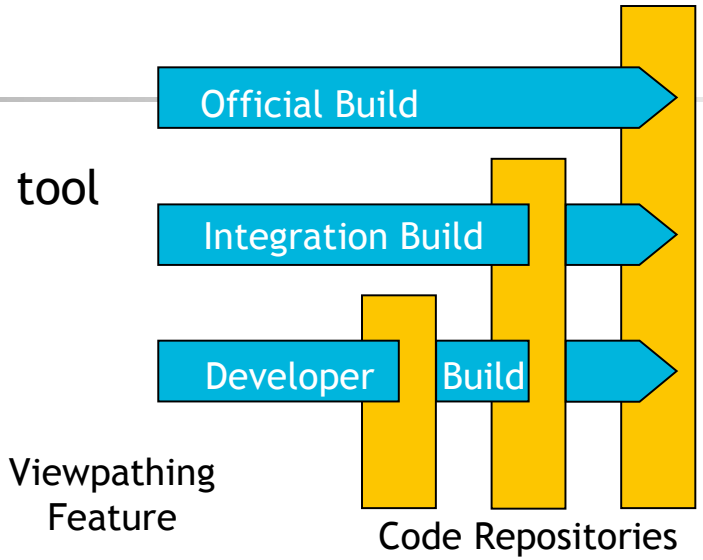
Scope of talk

- High-level technical overview
- Emphasizes unique features and benefits
- Agenda
  - Description of product
  - Benefits/impacts
  - Core features
  - Recent features

Audience

- Build engineers, developers, technical managers
  - Anyone with an overall interest in nmake's capabilities
- Assumes no previous background in configuration management, or in specific build tools

What is Alcatel-Lucent nmake?



- An advanced, proven, scalable software build tool
  - Simplifies and streamlines software builds
  - Integrated set of features significant enhancing and extending the conventional make model
- Developed by Alcatel-Lucent Bell Labs
- Widely deployed
  - Used by hundreds of projects within Alcatel-Lucent, AT&T, and elsewhere
- Built in support for widely used platforms and development tool chains
- Easy to tailor to new environments and needs
  - Customize using *rules* and *assertions*
- Also: a programming language, a virtual directory structure tool, a distributed/concurrent compilation tool, a scan language, a persistent state database, a C language preprocessor, ...

See http://www.bell-labs.com/project/nmake for more information.

Supported Platforms

- New releases are certified on the platforms shown below
- Releases generally upward compatible with newer OS versions
- Set of supported platforms follows market demand
- Ports to additional platforms may be performed upon request
- See download page on web site for up-to-date information

Platform	Versions	Platform Notes
AIX	5.1, 4.3	IBM
HP-UX	11.00, 10.20	HP (compatible with Itanium)
Red Hat Linux	RHELv3, 8.0, 7.2	X86; other Linux distributions may work
Solaris	2.10, 2.9, 2.8, 2.7, 2.6, 2.5.1	Sun Sparc
Windows/SFU	3.5	Windows Server 2003/Windows 2000

Alcatel-Lucent nmake Benefits

Enhanced build accuracy

- Automatic dependency generation
- Redundancy elimination (high level assertions, viewpathing)
- Persistent state retained across builds

Reduced effort

- Built-in tools knowledge and run-time tool probes
- Shared rule repositories (baserules, project rules)
- Reduced Makefile complexity (high level assertions)
- Enhanced flexibility (high level assertions, expressive rule language)
- Source in multiple directories (.SOURCE.x)

Improved performance

- Concurrent/distributed jobs
- Optimized shell interface
- Compiled Makefiles
- Scale to many users (viewpathing)

Alcatel-Lucent nmake Features/Benefits/Impacts

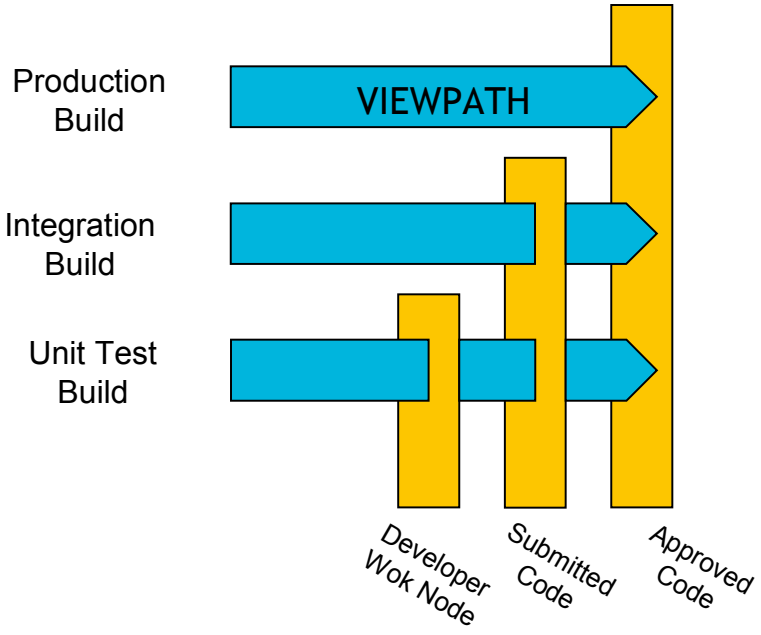
Feature	Benefit	Impact		
Scalable viewpathing	Easily reference multiple build trees.	P	E	A
State-based incremental builds	Ensures that inconsistent source files are rebuilt.	P		A
Dynamic dependency generation	Simplifies makefile maintenance. Fully automatic. Extensible.		E	A
Efficient shell interface	Allows concurrency, simplifies action blocks.	P	E	
Distributed/concurrent builds	Distributes jobs to idle hosts, or multiple CPUs.	P		
Makefile compilation	Speeds up execution times for large projects.	P		
Powerful rule language	Leads to extensible, concise, reusable makefiles.		E	
Assertion operators	Enables portable, high level Makefile specification.		E	A
Common actions	Automatically implements generic Makefile targets.		E	A
Built-in tool knowledge	Built-in support for widely used development tools.		E	A
Automatic tool probe	Automatically probes build platform/tools and configures build rules.		E	A
Portable Makefiles	Run Makefiles unmodified across platforms/compilers.		E	
Hierarchical Rule Management	Makefile ➠ Project rules ➠ baserules.		E	A
Metarules; metarule closure	Infer prerequisites and action by pattern.		E	
Variable edit operators	Flexible variable expansion; provides access to build dependency graph and build state.		E	A

Key: P: Increases **P**erformance E: Reduces build management **E**ffort A: Enhances build **A**ccuracy

Viewpathing

Viewpath defines ordered list of build nodes

- Each node is root directory of complete build tree
- Picks up leftmost occurrence of file in $VPATH
- Both source and derived objects subject to viewpathing
- Example spec: `VPATH=/home/me/proj1:/proj1/submitted:/proj1/approved`



- Implemented in user space using platform native filesystem and unmodified compilation tool chains
- No kernel modifications or special drivers needed
- Implemented without file copying or linking
- Writes go to top node only

Applications of Viewpathing

Individual developer workspaces

- Viewpath through integration, approved nodes
- Project files shared, not copied
  - Setup of developer node is very fast
- Supports arbitrary number of developers (each with own VPATH)

Build avoidance

- Share derived objects from nightly integration build/baseline
- Enormous savings in developer build time

Separate build products from project source

- Separate platform specific builds
- Builds with variant compilers, compiler options

Minimize file copying and disk space use

- Top node need only contain changed files

Eliminate use of inconsistent/out of date source

- Files down VPATH are automatically picked up
- Local copies/links not needed

State-Based Incremental Builds

What is a state-based incremental build?

- Key build data stored in a statefile: *makefile*.ms
  - State rules, state variables
  - File timestamps, explicit/implicit dependencies, action blocks, target attributes
- Target triggered if inconsistent with stored state

Benefits

- Able to detect inconsistency and trigger target re-build if
  - Rule action changed
  - Compiler or compiler flags changed
  - CPP #define variable value changed
    - On per-source file basis
  - File restored to earlier version
- Greatly improved incremental build accuracy
- Enormous savings in time and resource allocation

Dynamic Dependency Generation

Automatically derives implicit dependencies

- Generates header file and state variable dependencies
  - Rebuilds if CPPFLAGS changes!
- Uses integral programmable scanner
- Built-in rules available for widely used languages
  - C, C++, Fortran, ESQL/C, IDL, and others
- User can define new scan strategies
  - Scanner is customizable, even for default scan rules
- Scanner recursively follows `#include` chain

Provides significant benefits

- Dependencies maintained complete and up-to-date at all times
  - Generator run during target update procedure
  - Process completely transparent to user
- Eliminates a major source of development errors
- Handles generated prerequisites

Dynamic Dependencies Example

a.c

```
#include "b.h"
int main()
{
  b();
}
```

b.c

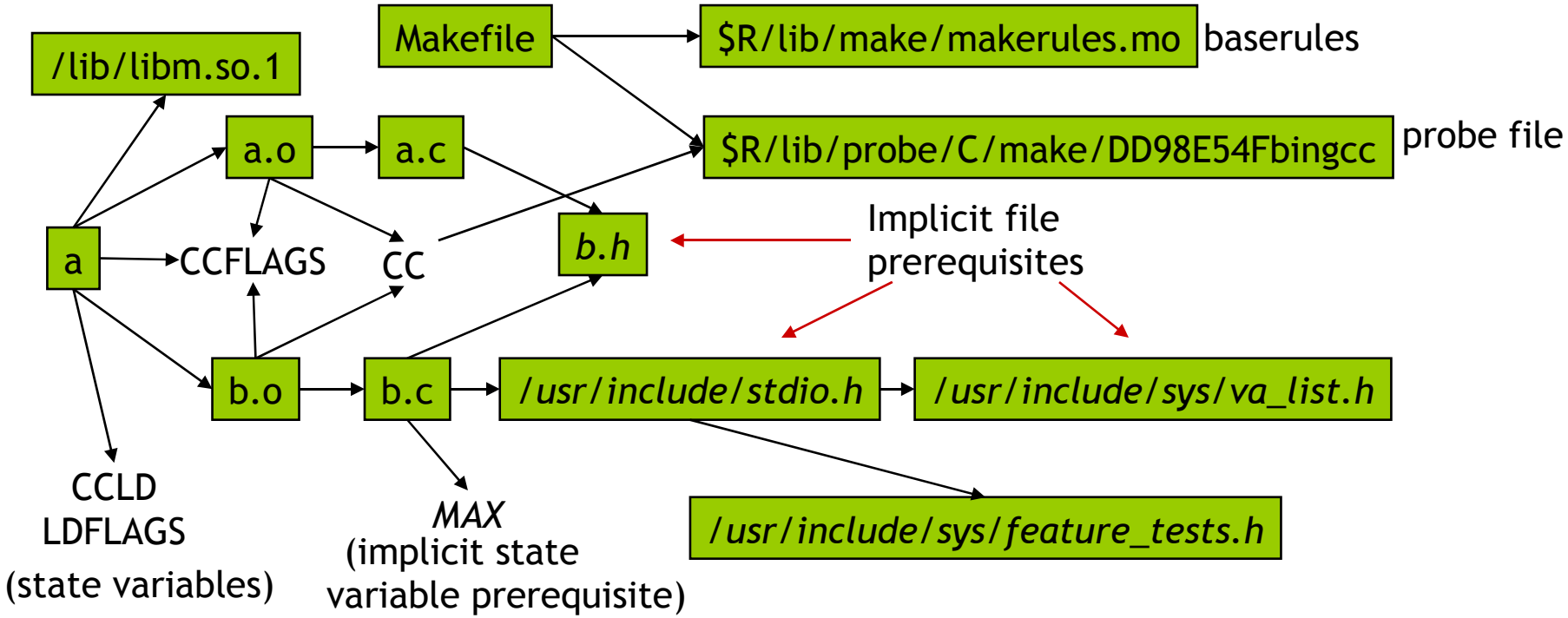
```
#include <stdio.h>
#include "b.h"
int b()
{
    printf("max=%d\n", MAX);
}
```

Makefile

```
MAX == 2
a :: a.c b.c -lm
```

run nmake

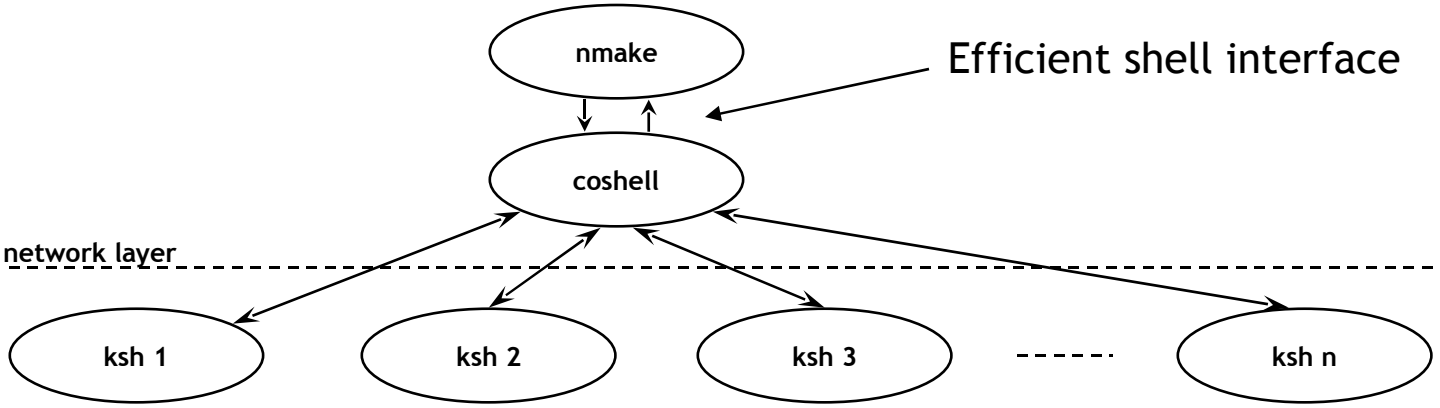
```
$ nmake
+ gcc -O -I. -I- -c a.c
+ gcc -O -I. -I- -DMAX=2 -c b.c
+ gcc -O -o a a.o b.o -lm
```



Concurrent and Distributed Processing

Supports multiple concurrent jobs on a single host, and can optionally run jobs concurrently on multiple hosts in a local area network.

- Essentially no Makefile changes required
  - nmake Makefiles are by default parallelizable
  - Implicit dependencies detect implicit ordering requirements
  - Mutual exclusion provides flexible per-rule concurrency control
  - Enable using COSHELL and NPROC variables
- Distributed jobs managed by coshell server process
  - coshell initiates persistent remote shells using rsh
  - Jobs distributed based on system load and idle time
- Observe 4-6x speedup



Powerful Rule Language for Extensibility

Assertions

- Central element of Makefiles (discussed later)

Variables

- Types: regular, state, automatic (work with viewpathing)
- Allow traversal of build dependency graph
- May be scoped within assertions

Edit Operators

- Perform many useful tasks
  - pattern matching, path manipulation, other
- Can be pipelined
- ~58 edit operators

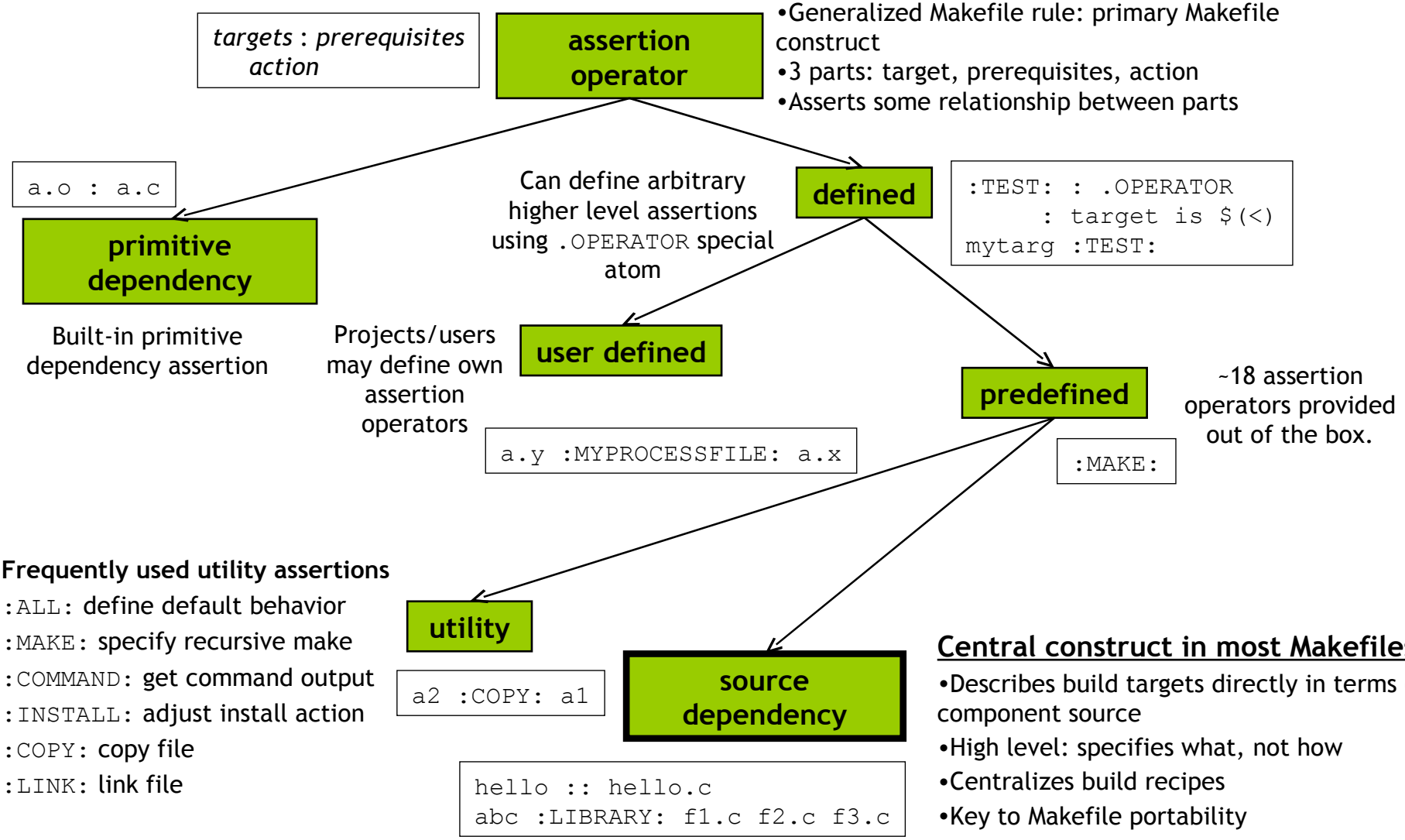
Special atoms

- Customize behavior, typically of an assertion
- ~95 special atoms

Programming language features

- Control flow, functions, arithmetic and string operations, etc.

Types of Assertion Operators



High-Level Assertions

- Minimal build specification
  - Generally specify only desired targets and sources
- Greatly simplify Makefile development
- Enhance flexibility
  - Implementation details specified elsewhere (baserules, global rules)
  - Enable Makefile portability
- Examples:

`a :: a.c b.c c.c`	Complete makefile to generate executable `a`. Install, clean, clobber automatically set up.
`libx.a :: x.c y.c z.c`	Complete makefile to generate archive `libx.a`
`x 1.0 :LIBRARY: x.c y.c -lz`	Generate library `x` with more control.
`:ALL:`	Build all source dependency targets by default.
`:MAKE: lib - cmd`	Recursively build `lib`, then build `cmd`
`$(INCLUDEDIR):INSTALLDIR: a.h`	Extend install common action by installing *rhs* in *lhs*.
`include $(VROOT)/java/jglob.mk` `:JAVA: com/lucent/stc`	Build java files specified on *rhs*.
`j1.jar :JAR: class/*.class`	Create jar from files specified on *rhs*.

Common Actions

Assertion operators work with predefined rules (baserules) to provide *common actions*

- Command line pseudo-targets providing useful functions
- Used on command line: `nmake install`
- Provided completely automatically
  - Conventional make tools require explicit definition
  - Ensures consistent operation among all Makefiles in project
- Over 16 common actions provided

Frequently used common actions

`all` - make all :: targets

`install` - make and install target files

`clean` - remove generated intermediate files

`clobber` - remove most generated files

`clobber.install` - remove installed files

`cc-`, `cca-` - build variant executables

`recurse` *action* - force recursive action

Other nmake Features

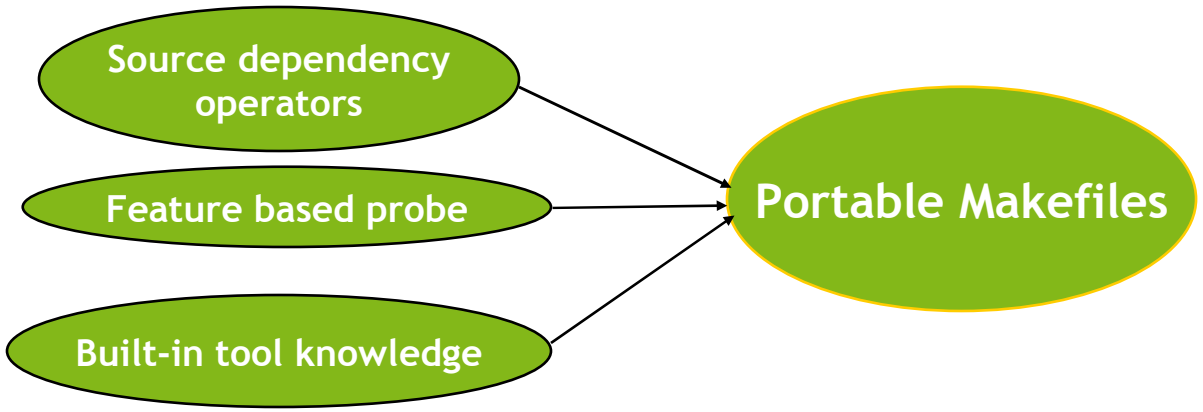
- C++ support
  - headers, templates
  - Tracking support for Sun C++ compilers
- IDL scan rule
- Instrumentation support
  - purecov, quantify, sentinel, insight/insure, codewizard
  - `nmake instrument=codewizard`
- Dependency reporting tools
- Serialized build log for concurrent/distributed builds
- Include makefile expansion
- Probe hints
- Option to disable probe
- Performance
  - engine lookahead thread, directory caching

Makefile Portability

High level build specification lists what to do, not how to do it
ex: `abc :LIBRARY: abc.c`

Automatically configures nmake and cpp for compiler/platform environment
ex: *current PIC flag is* `-KPIC`

Knowledge of build procedures for popular compilers and tools
ex: jar manifest file update



For example, given assertions to build a shared library, nmake will automatically adapt generated commands for the current compiler/platform.

Sample Areas of Variation	Typical Variants
shared object extension	.so, .sl
PIC generation option	-KPIC, -D_PIC_, -fpic, +z
option to make shared object	-G, -b
viewpathing support	built-in, nmake cpp
cpp override mechanism	cc option flag, env variable, sh wrapper
archive command/options	ar, CC -xar
command to use for loading	ld, $(CC)

Library generation assertion specified in Makefile:

```
CCFLAGS += $$(CC.PIC)
abc :LIBRARY: abc.c
```

Typical generated command sequence (Sun WS6 cc/Solaris 2.8):

```
+ cc -O -KPIC -I- -c abc.c
+ ar r libabc.a abc.o
+ rm -f abc.o
+ nm -p libabc.a …
+ cc -G -o libabc.so.1.0 -u
abc libabc.a
```

There are currently 34 “make” probe variables, some with multiple sub-options.

Dependency-based Java support

Dependency-based Java build feature supports:

- Viewpathing
- Simplified user interface
- Automatic dependency generation
- Incremental dependency update
- Dependency cycles and implicit targets
- Batched compilations
- Concurrent and distributed build
- Incremental global build
- Incremental safe inside package local build
- Jar file support
- #empty file filtering

Example Java Makefile:

```
JAVAPACKAGEROOT=$(VROOT)/java
:JAVA: com
```

- Recursively collects source files in specified sub-directories.
- Single :JAVA: can build entire source tree spanning multiple packages.

Example jar Makefile:

```
:ALL:
a.jar :JAR: class/*.class
```

- Single Makefile can build multiple jars.

Java Build Example

Makefile:

```
JAVAPACKAGEROOT=$(VROOT)/java
JAVACLASSDEST=$(VROOT)/class
:JAVA: com/alu/stc/pkg1 com/alu/stc/pkg2
```

Run nmake:

```
+ /tools/nmake/javadeps/jdeps … -o localjavadeps -m globaljavadeps -d
../class --classpath=../class com/alu/stc/pkg1/A.java …
+ /opt/exp/java/j2sdk1.4.0_01/bin/javac -d ../class -classpath ../class:.
com/alu/stc/pkg2/D.java com/alu/stc/pkg1/B.java com/alu/stc/pkg2/C.java
com/alu/stc/pkg2/E.java com/alu/stc/pkg1/A.java
```

- `JAVAPACKAGEROOT` is java package source root (required)
- `JAVACLASSDEST` is destination class tree root (optional)
- `:JAVA:` rhs may be *files* or *directories*; files may contain shell patterns
- `:JAVA:` picks up all java source in trees rooted at specified rhs directories
- `jdeps` initially generates global (including cross-package) dependencies
- Subsequent dependency updates and builds may be incremental for speed
- Dependencies ensure correct compilation sequence even in multi-package builds

nmake on Windows/SFU

nmake now available on Windows under the SFU subsystem

- Certified on Windows 2000 and Windows Server 2003
- Available since lu3.7 release
- Portably written Makefiles should work without change
- Several build tools supported
  - Interix native gcc compiler
  - MS Visual C/C++ compiler using /bin/cc wrapper
  - Native Windows Java SDK
- Handles DOS format Makefiles
- Supports POSIX style paths
- coshell not yet supported

See http://www.bell-labs.com/project/nmake/release/win.html for more information.

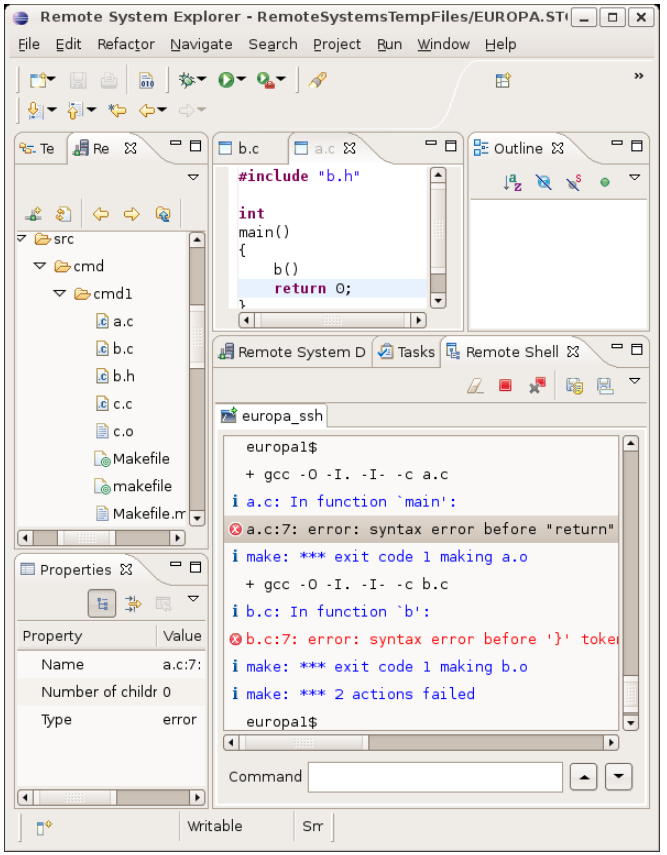
Use with Eclipse

Eclipse is “an open source community whose projects are focused on building an extensible development platform, runtimes and application frameworks for building, deploying and managing software across the entire software lifecycle.” (www.eclipse.org)

Eclipse:

- Is open source under the Eclipse Public License (EPL)
- Runs on multiple platforms include Linux, Windows, Solaris, AIX
- Is extensible using plugins and “extension points”
- Provides C/C++ and Java IDEs, also supports scripting languages such as perl and python

nmake may be used to build C/C++ and Java projects from within Eclipse. nmake may also run remotely using the TM/RSE (Remote System Explorer) plugin



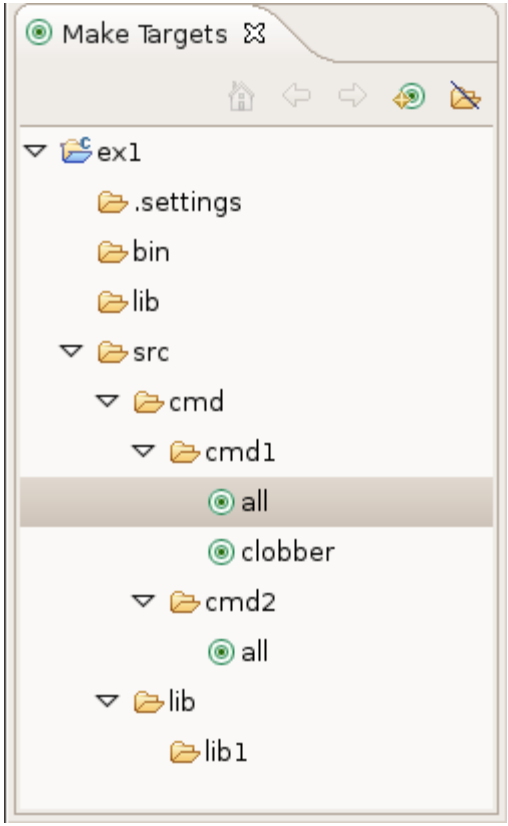
nmake build within RSE (Remote System Explorer) remote shell.

See http://www.bell-labs.com/project/nmake/manual/eclipsesupport.html for more information, including example multi-level build projects illustrating integration with Eclipse CDT, JDT, and TM.

Use with Eclipse CDT

nmake may be used to build C/C++ projects from within the popular Eclipse CDT IDE.

- Leverages Eclipse CDT Standard Make C/C++ mode
- Builds launched from within the Eclipse CDT environment
- Full range of build types possible
  - Full and incremental builds
  - Multiple level and directory local builds
  - Build types are specified and selected through the Eclipse CDT IDE
- Eclipse correctly populates “problems view” using alternate GNU compatible directory recurse message format
  - Set `recurse_begin_message = gnu` to enable (introduced in release alu3.9)
  - Rules file retrofitting release lu3.8 available
  - Enables correct mapping from build compilation errors back to source locations even in multi-level builds



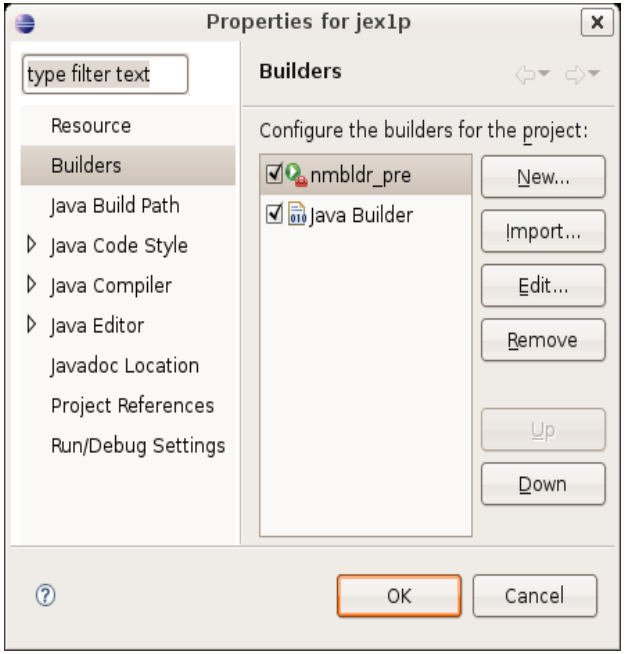
Eclipse CDT make target view for multi-level build.

See http://www.bell-labs.com/project/nmake/manual/eclipsecdt.html for more information, including a sample project illustrating a basic multi-level build integration scenerio.

Use with Eclipse JDT

nmake may be used as a builder within JDT, the Eclipse Java IDE.

- nmake may be configured as an Eclipse integrated external builder that participates in project builds
- The external builder runs before or after, but does not replace, the internal Java compilation step
  - Example: Java code generator implemented as a preprocessing step prior to compilation
- nmake may also be configured as an Eclipse "external tool" that is run on demand rather than during project builds
  - Example: Jar file generation using the nmake :JAR: asssertion
- This approach has been tested on Linux as well as Windows XP SP2 platforms
- External builders/external tools approach are general Eclipse features so may be used in non-JDT projects as well



Eclipse builder configuration panel showing an nmake-based build step.

See http://www.bell-labs.com/project/nmake/manual/eclipsejdt.html for more information, including an example multi-level build illustrating integration with Eclipse JDT.

make v. Alcatel-Lucent nmake: a Quick Example

Conventional Makefile:

```
TABS = -DTABS=8
USG= -DUSG=1
CFLAGS = -c ${TABS} ${USG}
SOURCE = main.c process.c hash.c
OBJECT = main.o process.o hash.o
INSTALLDIR = ${HOME}/bin
program : ${OBJECT}
     ${CC} ${OBJECT} -lm -o program
install :
     -mv ${INSTALLDIR}/program \
              ${INSTALLDIR}/program.old
     cp program ${INSTALLDIR}
main.o : main.c main.h
process.o : process.c process.h main.h hash.h
hash.o : hash.c hash.h
lint :
     lint ${CFLAGS} ${SOURCE}
clean :
     - rm -f core ${OBJECT}
clobber : clean
     - rm -f program
clobber.install:
     - rm ${INSTALLDIR}/program
```

Alcatel-Lucent nmake Makefile:

```
TABS == 8
USG == 1
program :: main.c process.c hash.c -lm
```

Despite small size, this nmake version does much more:

- Full viewpathing support
- Automatically maintained header dependencies
- Dynamic state variable dependencies minimize rebuilds
- Dependencies on rule action, compiler, and library
- Remembered state time to trigger updates
- Portable across platforms
- Support for additional common actions such as cc-
- Automatic common actions consistent across Makefiles
- Concurrent and distributed build

Conclusions, Q&A

Alcatel-Lucent nmake is an advanced, flexible build tool

- Has a long, successful track record
- Available on UNIX/Linux/Windows®
- Actively supported by Bell Labs
- Leads to significant savings in build times, resource usage, makefile maintenance, and portability
- New product releases regularly scheduled

Come visit our website

http://www.bell-labs.com/project/nmake

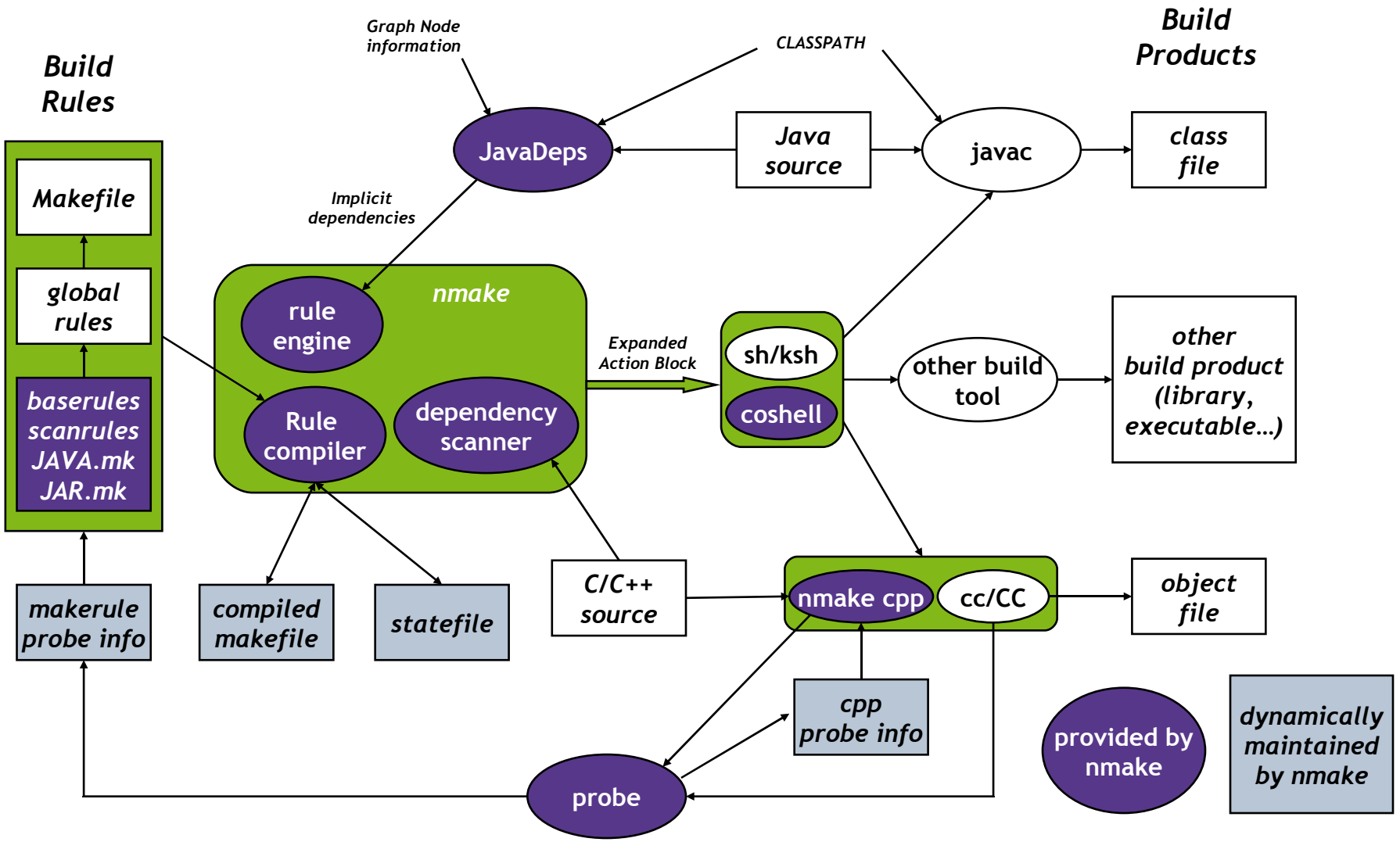
Contains availability and ordering information, latest documentation, FAQ, newsletters, and training information

BACKUP

Alcatel-Lucent nmake Technical Support

- Product fully supported by knowledgeable experts
- Support continuously staffed during normal business hours
  - Available via email during normal business hours (US, Eastern time)
  - Access via nmake@alcatel-lucent.com
- Response within 1 business day
  - Typically within 1 hour
- Support includes
  - Problem resolution
  - Assistance using product
- Patches provided if necessary for last 2 point releases
  - Will provide baserule fixes for earlier releases as possible

Core Components of Alcatel-Lucent nmake



GNU make v. Alcatel-Lucent nmake, feature comparisons

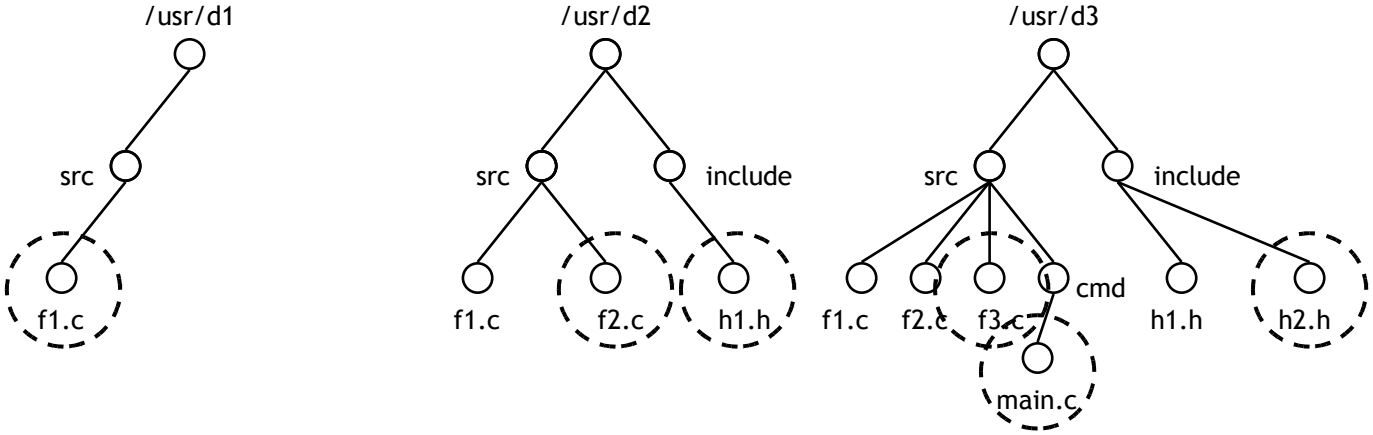
• Interprets makefile on each execution	MAKEFILE PROCESSING	• Compiles makefile, recompiles only if original is altered
• Has no knowledge of previous executions, must infer out-of-date targets by checking timestamps	REMAKING TARGETS	• Creates a statefile, using the data in it to determine if target out-of-date
• No preprocessor	PREPROCESSING	• C preprocessor included
• Each command line is executed in a new subshell	COMMAND EXECUTION	• Shell command lines are processed as an *action block*
• No mechanism to scan a file and detect implicit dependencies. Header files must be explicitly specified in makefile	AUTOMATIC DETECTION OF DEPENDENCIES	• Relationships between source and header files are dynamically determined
• Non-portable tool commands specified directly in Makefile.	PORTABILITY	• Minimal specification and built-in tool knowledge enable portable Makefiles.

GNU make v. nmake, feature comparisons

• VPATH variable -- a colon or blank separated list of directories to be searched (as with nmake's .SOURCE Special Atom). No notion of directory nodes. • "vpath" directive -- colon or blank separated list of directories which allow searching for files that match a particular pattern	SEARCHING DIRECTORIES FOR DEPENDENCIES	• VPATH variable -- a colon separated list of directory nodes, which are viewed as a single virtual node comprising the product's directory structure • .SOURCE Special Atom -- a blank separated list of directories to search for files. Used in conjunction with VPATH
• Not supported	VARIABLE PREREQUISITES	• State variables can be used as prerequisites, similar to files. Variable values and modification times are saved in the makefile's statefile
• Not supported	DISTRIBUTED BUILDS	• Uses network shell coprocess server (coshell) to distribute jobs amongst homogeneous machines within a LAN
• All targets must be explicitly defined in the makefile, including common targets such as install, clean, all	COMMON ACTIONS	• Commonly used targets are implicitly defined in the product's base rules, and can be expanded upon

Scalable Viewpathing Scheme

- VPATH=/usr/d1:/usr/d2:/usr/d3
  - Suppose the program consists of main.c, along with functions and header definitions in the remaining files. With viewpathing, a build for this example would use the following versions of these files:
    - /usr/d1/src/f1.c
    - /usr/d2/src/f2.c
    - /usr/d3/src/f3.c
    - /usr/d2/include/h1.h
    - /usr/d3/include/h2.h
    - /usr/d3/src/cmd/main.c



Scalable Viewpathing Scheme

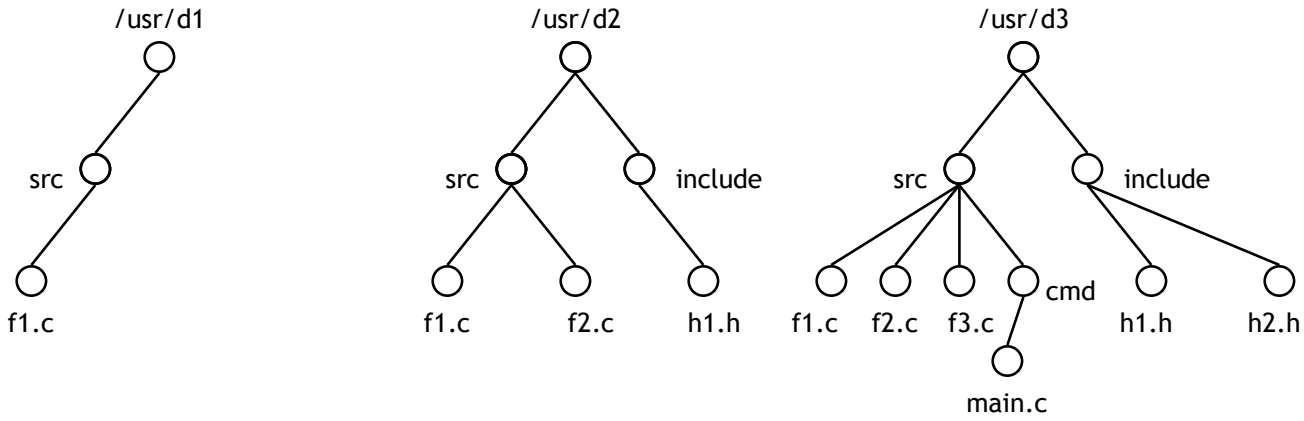
- VPATH=/usr/d1:/usr/d2:/usr/d3

- A Makefile for this example:

The first two lines tell nmake where to find *.c and *.h file

```
.SOURCE.c : src src/cmd
.SOURCE.h : include
main :: main.c f1.c f2.c f3.c
```

The third line uses a built-in “assertion”, and is all that is required to build the program; nmake determines the proper settings from the development environment.



Architecture of Interix Port – High Level