



A Technical Overview of Alcatel-Lucent nmake Product Builder Advanced Scalable Software Build Tool

nmake@alcatel-lucent.com

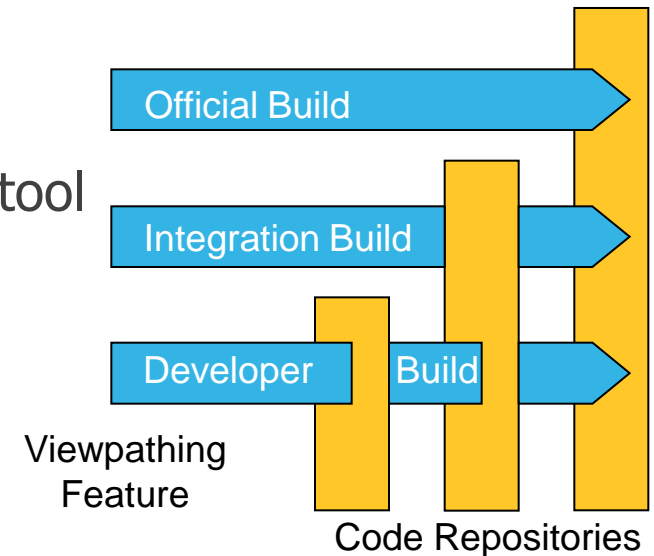
<http://www.bell-labs.com/project/nmake>

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

- nmake is supported 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. Tier 3
HP-UX	11.00, 10.20	HP (compatible with Itanium)
Red Hat Linux	RHELv4, 8.0, 7.2	X86: Most major Linux distributions known to work. Port to Linux on IBM System Z available
Solaris	2.10, 2.9, 2.8, 2.7, 2.6, 2.5.1	Oracle Sparc
Windows/SFU	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)
 - Web/XML based build logs
- Improved performance
 - Concurrent/distributed jobs
 - Optimized shell interface
 - Compiled Makefiles
 - Scale to many users (viewpathing)

See <http://www.bell-labs.com/project/nmake/impact> for more benefits/impacts

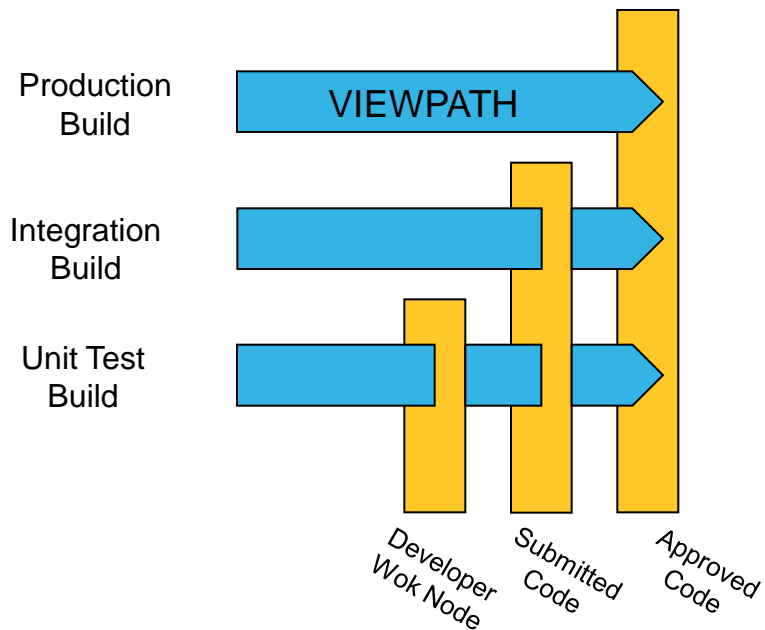
Alcatel-Lucent nmake Features/Benefits/Impacts

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
Web/XML based build logs	Facilitates faster build log analysis		E	

Key: **P: Increases Performance** **E: Reduces build management Effort** **A: Enhances build Accuracy**

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
- File accesses run at full native file system speeds
- 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

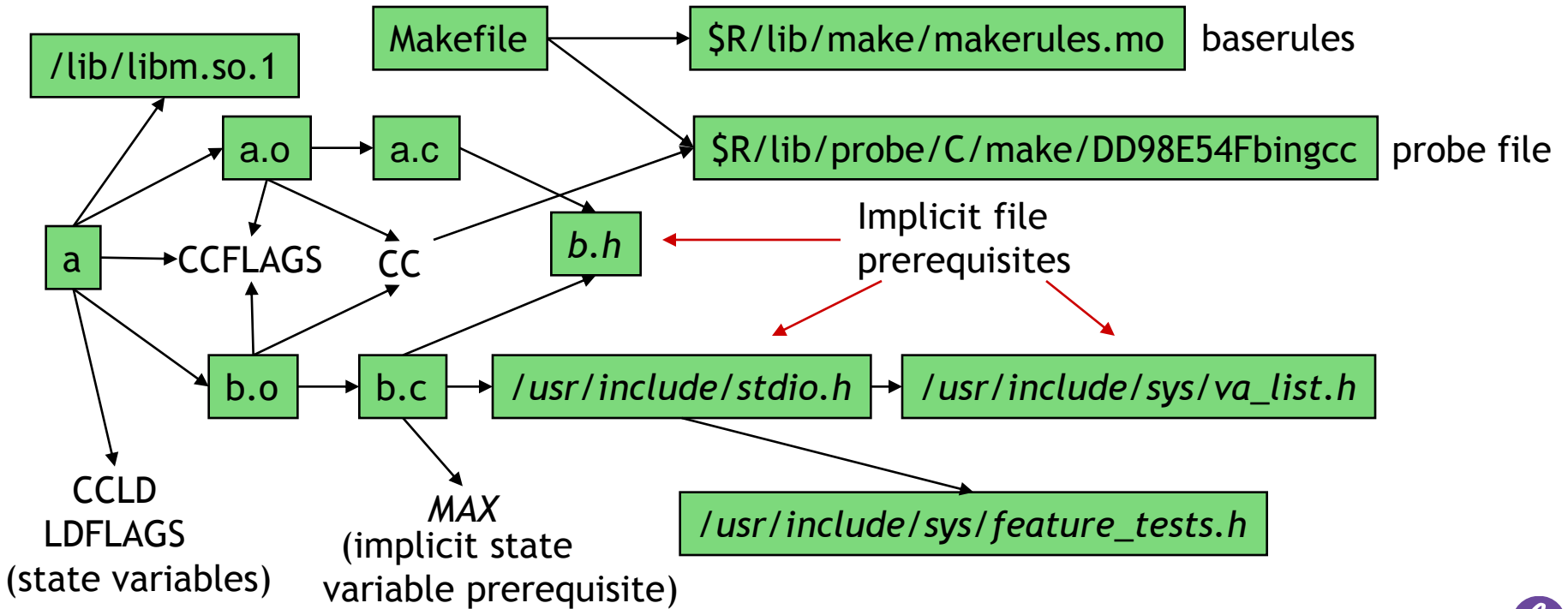
```
#include <stdio.h>
#include "b.h"
int b()
{
    printf("max=%d\n", MAX);
}
```

Makefile

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

run nmake

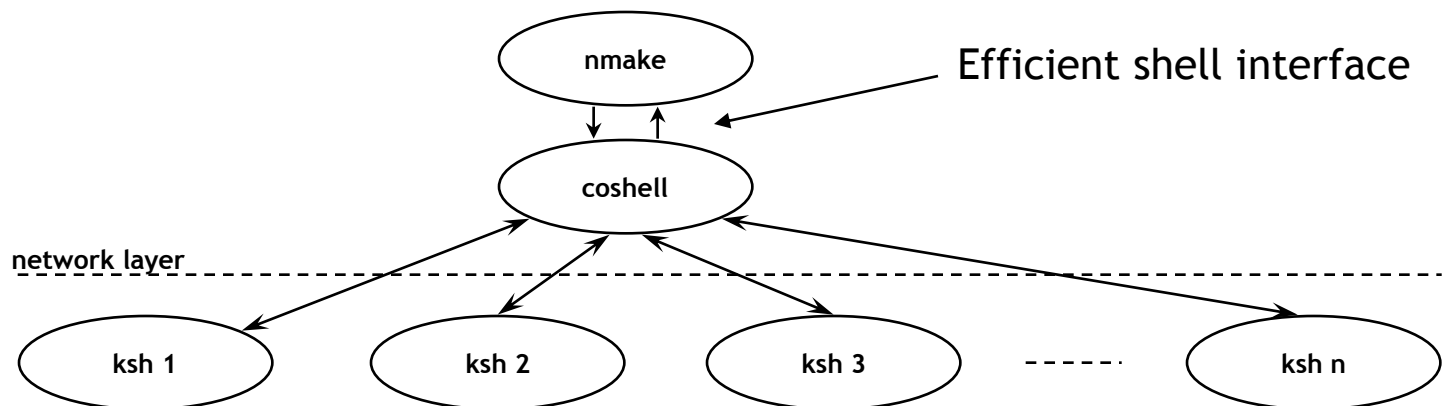
```
$ nmake
+ gcc -O -l. -l- -c a.c
+ gcc -O -l. -l- -DMAX=2 -c b.c
+ gcc -O -o a a.o b.o -lm
```



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.

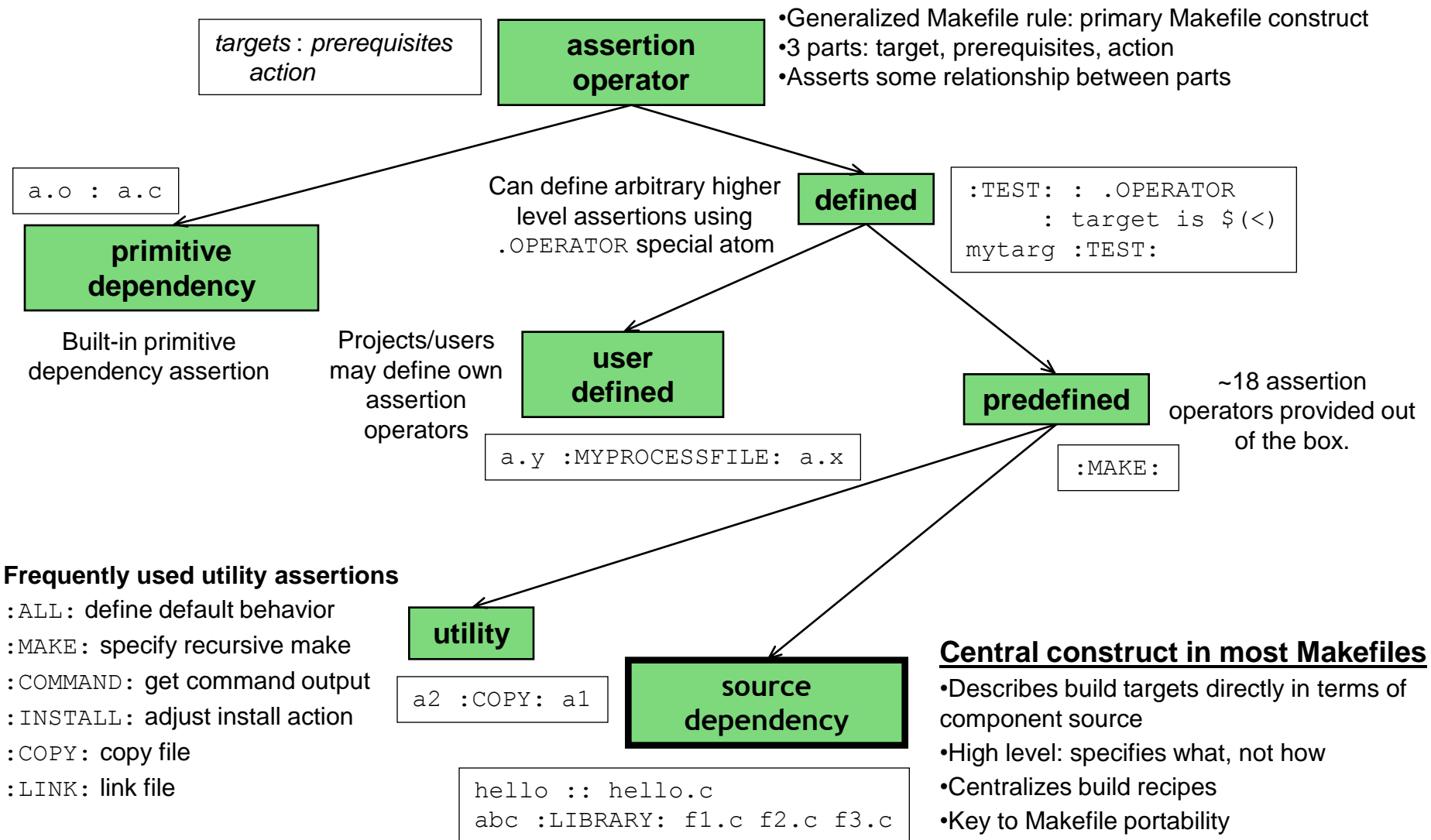
- Essentially no Makefile changes required
 - nmake Makefiles are by default parallelizable
 - Implicit dependencies detect implicit ordering requirements
 - Mutual exclusion provides flexible per-rule concurrency control
 - Enable using COSHELL and NPROC variables
- Distributed jobs managed by coshell 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
 - ~60 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:

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



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 Oracle Solaris 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
- Structured build log/web-based build log visualization
- Interoperability with Eclipse

Makefile Portability

High level build specification lists what to do, not how to do it

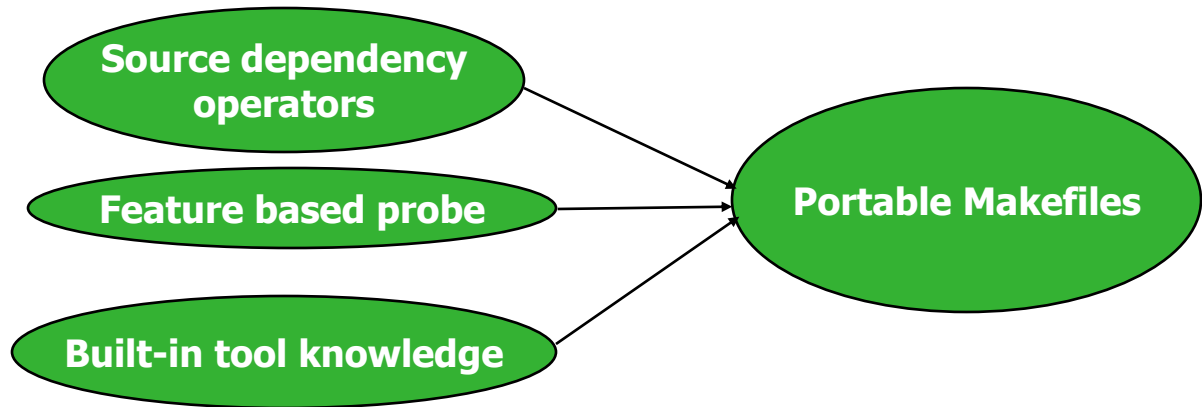
ex: `abc :LIBRARY: abc.c`

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, -shared
viewpathing support	compiler built-in, nmake cpp
cpp override mechanism	cc option flag, env variable, sh wrapper
archive command/options	ar, CC-xar
command to use for linking	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/javadepts/jdeps ... -o localjavadepts -m globaljavadepts -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

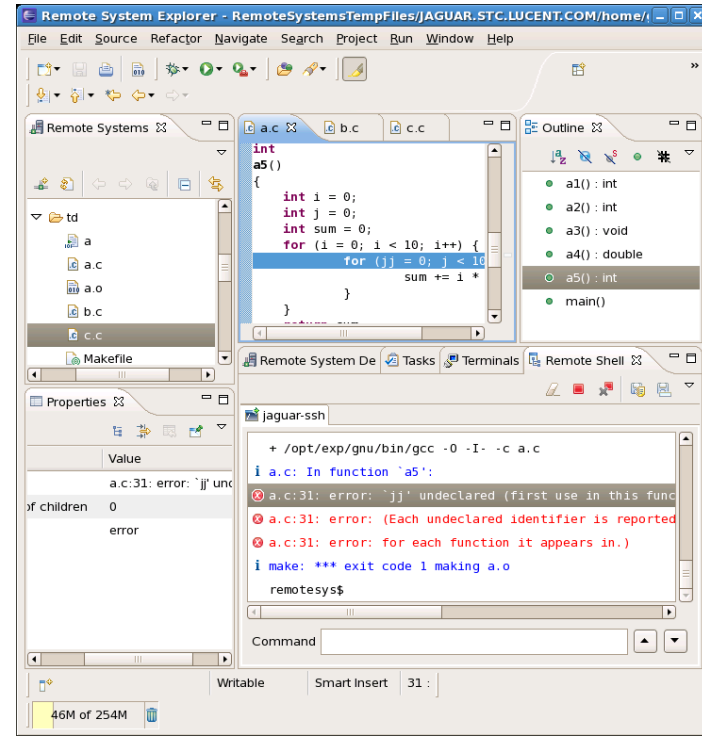
nmake on Windows/SFU

- 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, HP-UX, Mac OS X
 - Is extensible using plugins and “extension points”
 - Provides C/C++ and Java IDEs, also supports dynamic languages such as perl, python, ruby
- 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) and RDT (Remote Development Tools) plugins

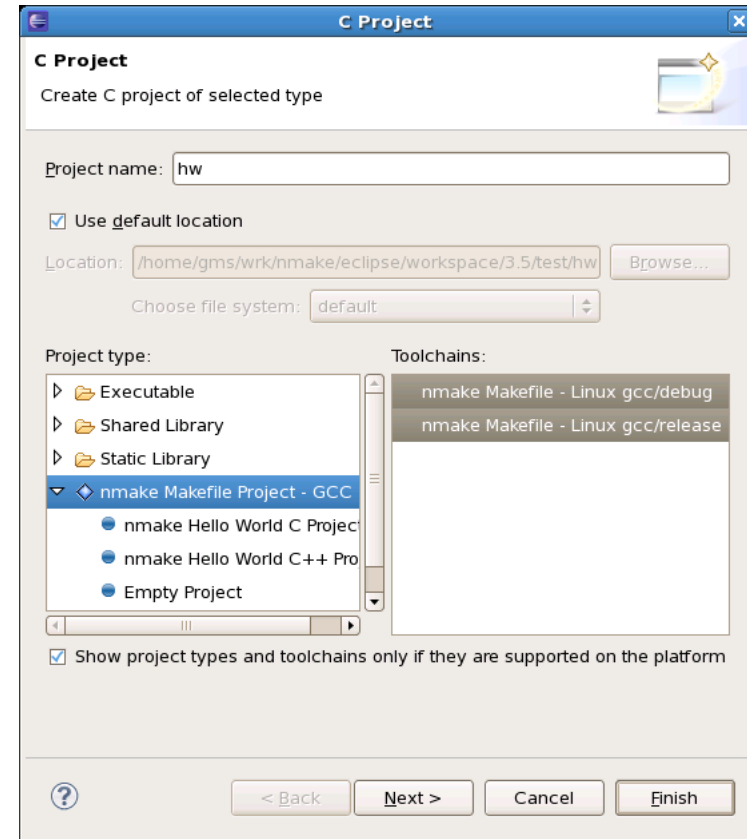


nmake build within TM/RSE
(Remote System Explorer)
remote shell.

See <http://www.bell-labs.com/project/nmake/manual/eclipsesupport.html> for more information

nmake Eclipse CDT Plugin

- The nmake Eclipse plugin supports use of nmake in Eclipse CDT
 - nmake/CDT tool chain definitions
 - Configure nmake as the C/C++ project build tool
 - Support gcc/g++ on Solaris/Linux
- Fully functional multi-level C and C++ project templates
- “Make Targets View” allows build from lower levels in the hierarchy
- Eclipse correctly populates “problems view” using alternate GNU compatible directory recurse message format
- Plugin contains an integrated helpbook and is distributed using the nmake update site



Eclipse CDT project creation wizard showing nmake project types and tool chains.

See http://www.bell-labs.com/project/nmake/manual/eclipse/help_cdt_plugin/gettingstarted.html for more information

Structured Build Log

- Captures dynamic structure of a build
 - Hierarchical Makefile/target nesting
 - XML format allows processing and analysis by scripting languages such as XSLT and Python
- Captures additional information about the build
 - Associates job output with triggering rule
 - Target attributes: name, times, error codes, host
 - Reason why target triggered
 - Build metadata such as build arguments, VPATH, UID, nmake version
 - User defined metadata
- Possible uses
 - Log structure visualization, Makefile hierarchy navigation, drill-down
 - Web-based build log
 - Python/Javascript based visualization
 - Automated log analysis
 - Build performance optimization
 - Text based report of build errors

Structured Build Log

- Captures additional build data in an XML formatted build log. Captured data includes:

Per target data

Job start and stop times
Exit code
Name of triggering rule
Reason rule triggered

Structural data

Delimited output for each triggered target
Hierarchical relationships of jobs and Makefiles

Per Makefile data

Makefile start and stop times
VPATH, PATH
Makefile Name
Execution host
nmake version

Build metadata (information describing the build)

Build start and stop times
Build arguments
User defined (such as build-id)

- Log is easily extended by user to capture any data accessible in rule and job shell execution contexts.

See <http://www.bell-labs.com/project/nmake/manual/buildlog.html> for more information

Web Based Build Logs

- Command `buildlog2html` transforms a structured build log into an interlinked collection of web pages providing overview and detailed views of a build.
 - High level views show overall hierarchical structure of build
 - Links provide drill-down to details for each triggered target
 - Color coding is used to highlight failed targets
 - Indices provide fast navigation to failed targets
 - Multi-build indices are automatically generated

See example <http://www.bell-labs.com/project/nmake/manual/buildlogs/xyz/nightly-20100722-linux>

Build log visualization examples

<http://www.bell-labs.com/project/nmake/manual/buildlogs/xyz/>

Project XYZ



builds for week 07/19/2010—4 Builds Failed

n	Outcome	Project	Build	Host	Start	Duration	Triggered Leaf Targets		Triggered Makefiles	
							Failed	Total	Failed	Total
1	failed	XYZ	nightly-20100722-linux	phoebe	2010-07-23 12:56:21.09	5m39.03s	3	458	1	15
2	failed	XYZ	nightly-20100722-solaris	jaguar	2010-07-23 12:45:04.19	3m58.30s	3	458	1	15
3	succeeded	XYZ	nightly-20100723-linux	phoebe	2010-07-23 13:10:23.78	2m9.82s	0	24	0	15
4	succeeded	XYZ	nightly-20100723-solaris	jaguar	2010-07-23 13:10:13.26	20.46s	0	24	0	15
5	failed	XYZ	release-201007231327-linux	phoebe	2010-07-23 13:27:40.48	6m14.24s	1	465	0	15
6	failed	XYZ	release-201007231327-solaris	jaguar	2010-07-23 13:27:22.84	4m5.87s	1	465	0	15
7	succeeded	XYZ	release-201007231339-linux	phoebe	2010-07-23 13:39:31.38	1m31.75s	0	15	0	15
8	succeeded	XYZ	release-201007231339-solaris	jaguar	2010-07-23 13:39:22.08	17.46s	0	15	0	15

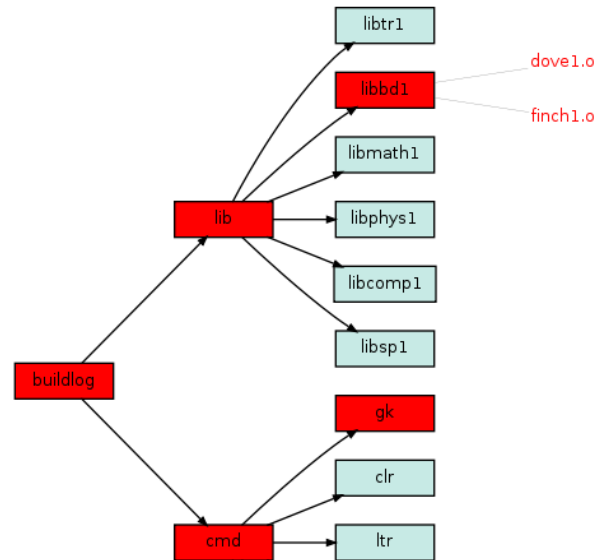
Alcatel-Lucent nmake

2010-07-23 17:57:53Z

Build log views: [list](#) [table](#) [graph1](#) [graph2](#) [summary](#) [text](#) [xml](#) Expanded views: [list](#) [table](#) [graph1](#) [graph2](#)
 Failed targets: [dove1.o](#) [finch1.o](#) [boysenberry1.o](#) Failed Makefiles: [gk](#)

Project XYZ Build Log—Graph1 View

Build [nightly20100722linux](#) Host phoebe—3 Targets Failed, 1 Makefile Failed



Build log views: [list](#) [table](#) [graph1](#) [graph2](#) [summary](#) [text](#) [xml](#) Expanded views: [list](#) [table](#) [graph1](#) [graph2](#)
 Failed targets: [dove1.o](#) [finch1.o](#) [boysenberry1.o](#) Failed Makefiles: [gk](#)

Project XYZ Build Log—List View



Build [nightly20100722linux](#) Host phoebe—3 Targets Failed, 1 Makefile Failed

- buildlog 5m39.03s
 - o lib 4m11.33s
 - libtr1 56.19s
 - libbd1 20.59s
 - dove1.o 0.13s
 - finch1.o 0.13s
 - libmath1 68.13s
 - libphys1 38.10s
 - libcomp1 34.56s
 - libsp1 28.44s
 - o cmd 1m21.78s
 - gk 13.48s
 - clr 13.71s
 - ltr 21.79s
 - gsl 13.30s
 - ml1 9.62s
 - br1 4.95s
 - boysenberry1.o 0.13s

Alcatel-Lucent nmake

2010-07-23 17:53:38Z

10 Target dove1.o

Attribute	Value
Duration	0.13s
Start Time	2010-07-23 12:57:34.17
End Time	2010-07-23 12:57:34.30
Host	phoebe
Reason	state variable (CCFLAGS) initialized to '-O'
Exit Code	1

Output:

```
+ cc -O -l -c /home/ricb/projects/nightly/source/src/lib/libbd1/dove1.c
cc1: note: obsolete option -l- used, please use -iquote instead
/home/ricb/projects/nightly/source/src/lib/libbd1/dove1.c: In function 'dove1':
/home/ricb/projects/nightly/source/src/lib/libbd1/dove1.c:5: error: expected ';' before ')' token
```

11 Makefile Output

```
make: *** exit code 1 making dove1.o
```

12 Target duck1.o

Attribute	Value
Duration	0.31s
Start Time	2010-07-23 12:57:34.37
End Time	2010-07-23 12:57:34.68
Host	phoebe
Reason	state variable (CCFLAGS) initialized to '-O'

Script-Based Build Reporting / Analysis Tools

- Set of small stand-alone scripts analyzing structured build logs
- Currently available reports include: plain text error report, graphical job/Makefiles durations reports, hierarchical build reports including sunburst
- Results are output in plain text, HTML, and graphical formats
- High-level scripting allows quick prototyping
 - Useful data extraction possible in 4-6 line scripts
- Any modern scripting language can be used
 - Example scripts in several scripting languages are available
 - Good support for XML, JSON, date/time manipulation desirable
- Currently using Python for prototyping build processing scripts

See <http://www.bell-labs.com/project/nmake/tools/reports.html> for more information

Tool Repository

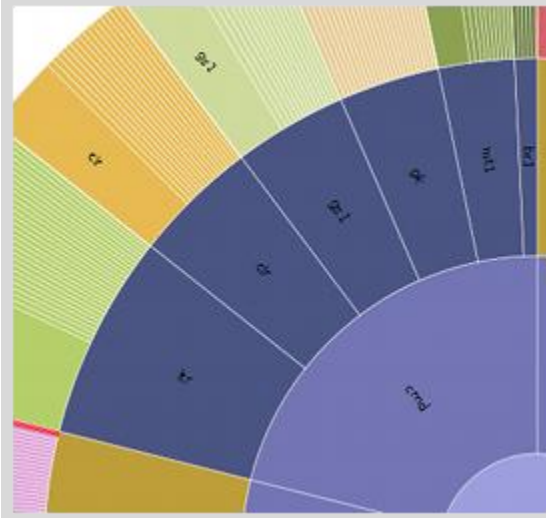
- Web based collection of small standalone tools and scripts
- Allows to publicize tool availability and get tools out to users fast
- Facilitates quick experimentation and prototyping
- Encourages user customization and tool extension
- Encourages tool sharing
- Current contents include scripts for processing structured build logs
 - Demonstrate log processing techniques
 - Provide simple common infrastructure simplifying log processing
 - Includes script producing informative plain text build error report

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

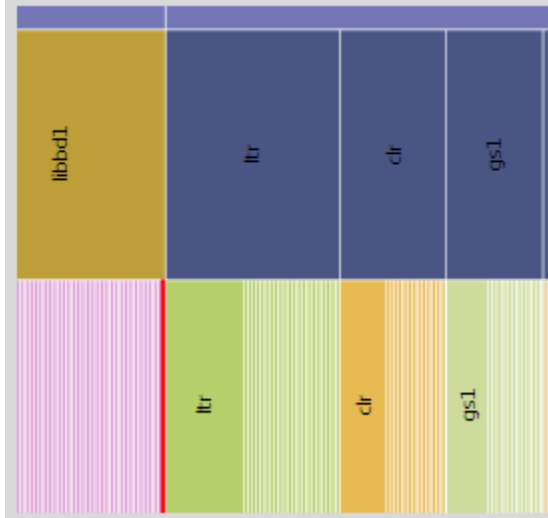
Build log scripted report examples

<http://www.bell-labs.com/project/nmake/tools/>

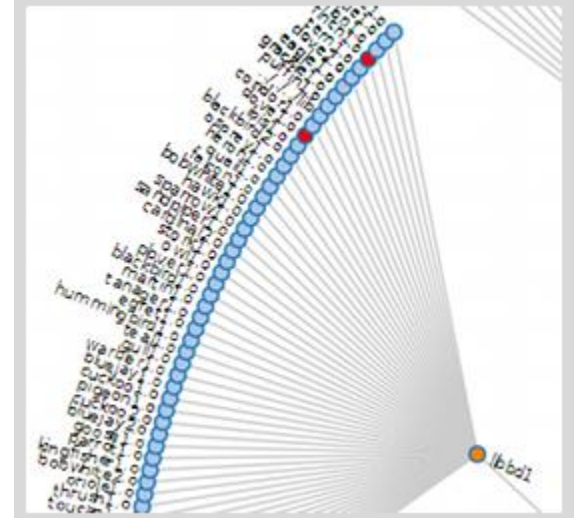
Sunburst



Icicle



Radialtree



Target Durations

omath1.a	
l libtr1.a	
bphys1.a	10.60
ocompl.a	10.29
l libsp1.a	7.37
cmd/ltr ltr	6.60
md/clr clr	3.90
d/gsl gs1	3.46
RCLEAN	2.96
RCLEAN	2.82
mt1 mt1	2.28
RCLEAN	1.95
RCLEAN	1.73
RCLEAN	1.34
apple1.o	0.8
aplace1.o	0.6
imann1.o	0.1

Web Build Index URL: <http://my.example.com>

Errors

```
Failed Target 1
=====

Error code: 1
Makefile target: a.o
Execution directory: /home/richb/newsletter/
Offset from vpath: src/cmd/t1
Make recursion level (MAKELEVEL): 2
Start time: 2010-12-17 15:06:53.805588 -05:00
End time: 2010-12-17 15:06:54.325555 -05:00
Job duration: 0:00:00.519967
Why triggered: state variable (CCFLAGS) init
Weblog URL: http://my.example.com/mybuild/m
Rule action output:
+ gcc -O -I- -c a.c
a.c: In function 'a':
a.c:5: error: syntax error before '3' token
```

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®
 - Responsive technical support provided
 - 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

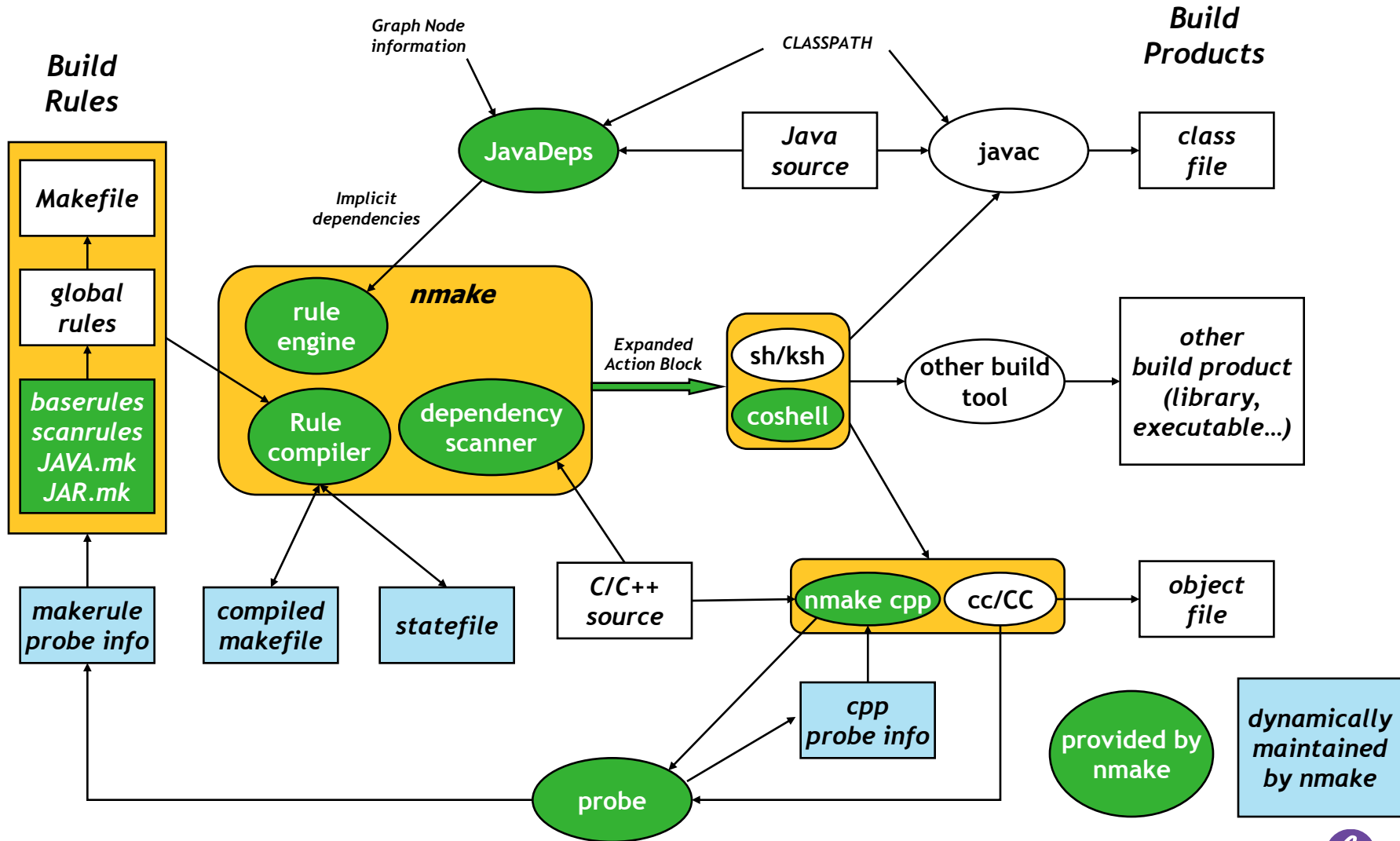


BACKUP

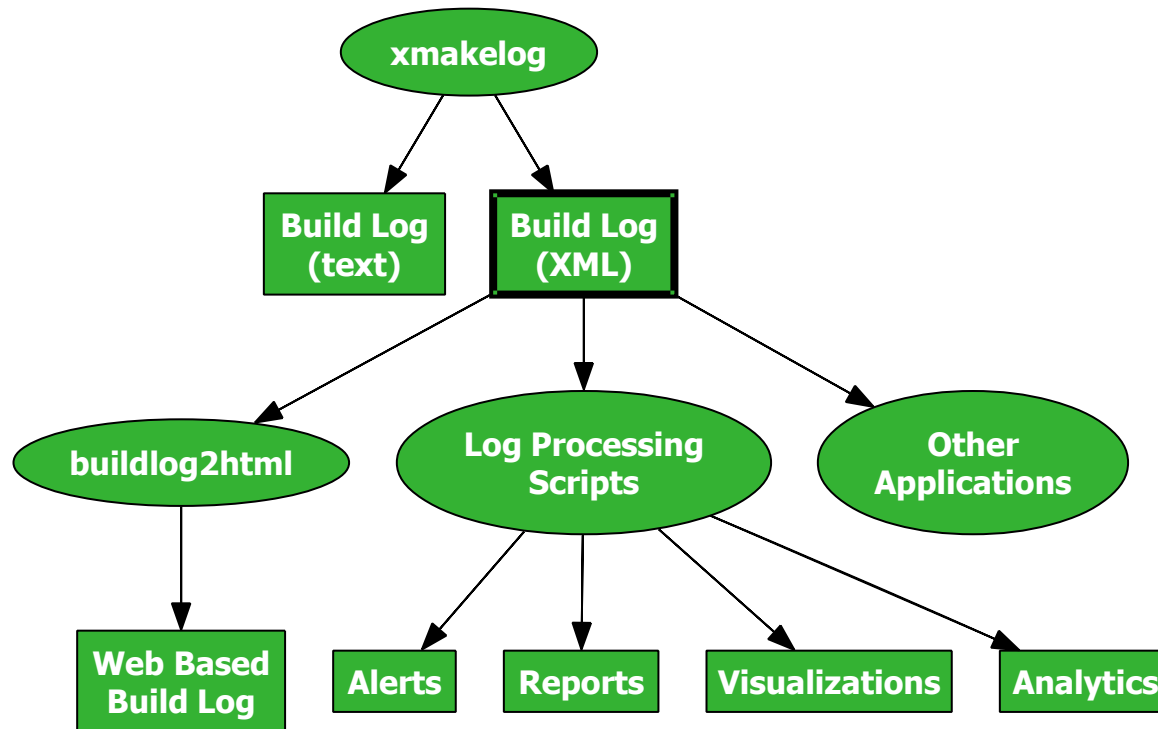
Alcatel-Lucent nmake Technical Support

- 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



Structured Build Log Processing Overview



- Use xmake log command instead of nmake command
- Additional XML build log produced in addition to traditional flat text log
- XML log may be processed by analysis applications to provide insight into the build
- Buildlog2html analysis application generates HTML and is provided with nmake
- User-extensible scripts provide additional analysis/visualization

GNU make v. Alcatel-Lucent nmake, feature comparisons

<ul style="list-style-type: none">• Interprets makefile on each execution	MAKEFILE PROCESSING	<ul style="list-style-type: none">• Compiles makefile, recompiles only if original is altered
<ul style="list-style-type: none">• Has no knowledge of previous executions, must infer out-of-date targets by checking timestamps	REMAKING TARGETS	<ul style="list-style-type: none">• Creates a statefile, using the data in it to determine if target out-of-date
<ul style="list-style-type: none">• No preprocessor	PREPROCESSING	<ul style="list-style-type: none">• C preprocessor included
<ul style="list-style-type: none">• Each command line is executed in a new subshell	COMMAND EXECUTION	<ul style="list-style-type: none">• Shell command lines are processed as an <i>action block</i>
<ul style="list-style-type: none">• No mechanism to scan a file and detect implicit dependencies. Header files must be explicitly specified in makefile	AUTOMATIC DETECTION OF DEPENDENCIES	<ul style="list-style-type: none">• Relationships between source and header files are dynamically determined
<ul style="list-style-type: none">• Non-portable tool commands specified directly in Makefile.	PORTABILITY	<ul style="list-style-type: none">• 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
- Not supported

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
- All targets must be explicitly defined in the makefile, including common targets such as install, clean, all

VARIABLE PREREQUISITES

- State variables can be used as prerequisites, similar to files. Variable values and modification times are saved in the makefile's statefile

DISTRIBUTED BUILDS

- Uses network shell coprocess server (coshell) to distribute jobs amongst homogeneous machines within a LAN

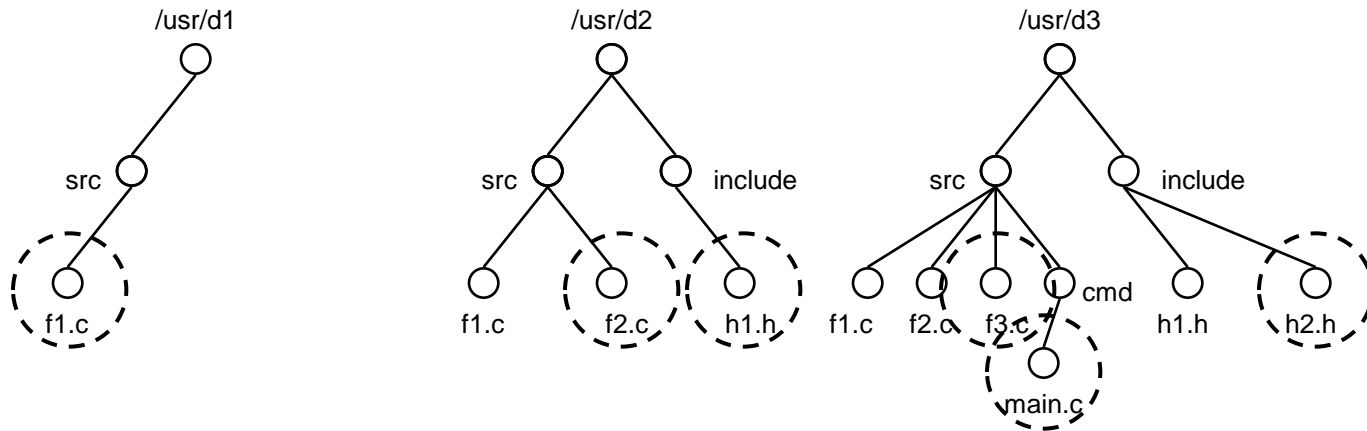
COMMON ACTIONS

- 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

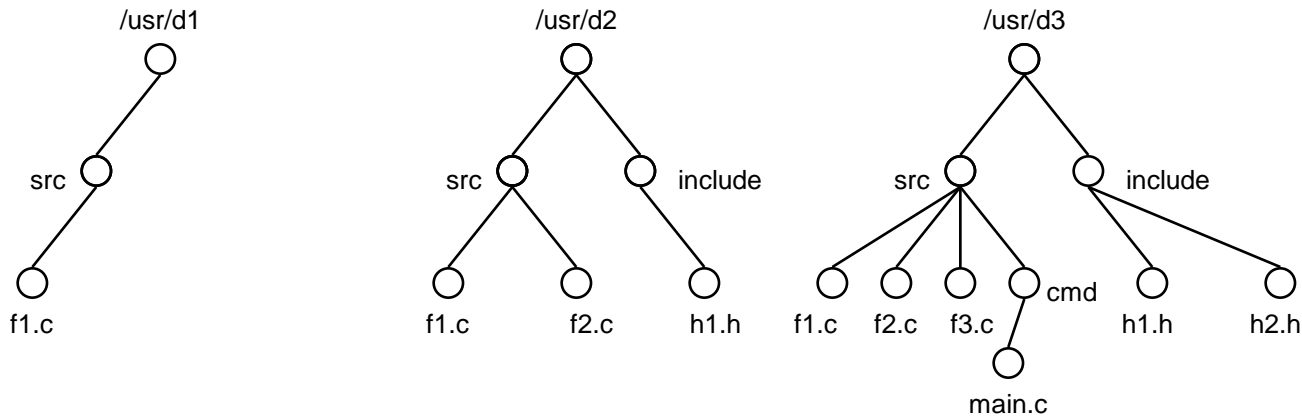
- VPATH=/usr/d1:/usr/d2:/usr/d3

- A Makefile for this example:

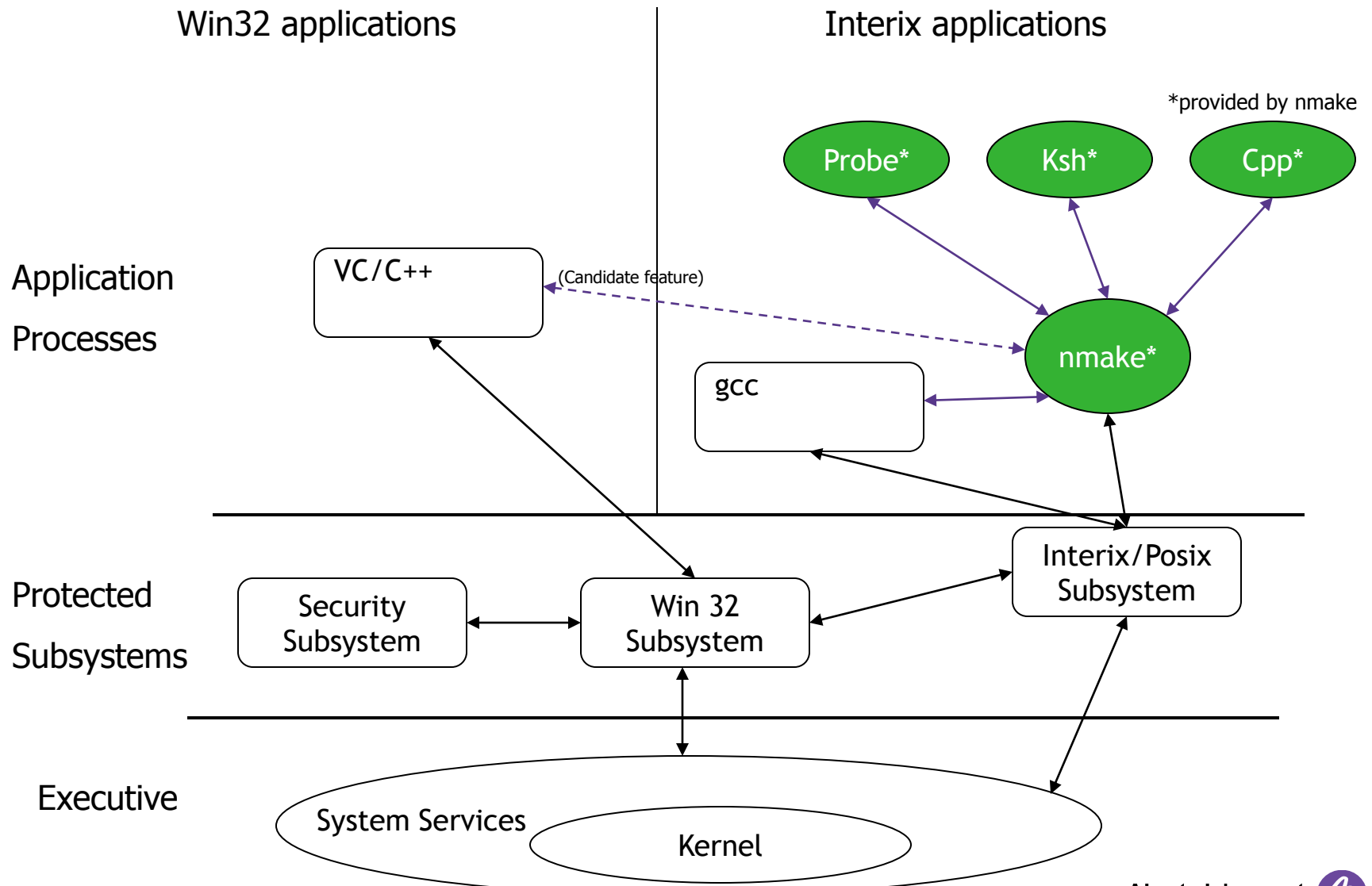
The first two lines tell nmake where to find *.c and *.h file

```
.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



AT
THE
SPEED
OF
IDEAS™

www.alcatel-lucent.com