Alcatel-Lucent nmake Structured Build Logs

April 2014

Abstract

Alcatel-Lucent nmake release 11 and above can output build logs in an XML structured format. In contrast to traditional textual build logs, the structured format clearly tags the elements of the build log, making the content and structure of the log explicit. This facilitates automated build log processing and analysis. Applications include build log visualization such as generation of high level graphical views of a build and build performance analysis. The standard XML format allows use of widely available tools to speed implementation of such applications. In addition to the build command output found in traditional build logs, the structured format contains additional data such as build target attributes and timing information which can provide additional insight into a build. The set of additional data is user customizable. This document provides a description of the structured build log feature and a guide to its use.

Table of Contents

1. Overview

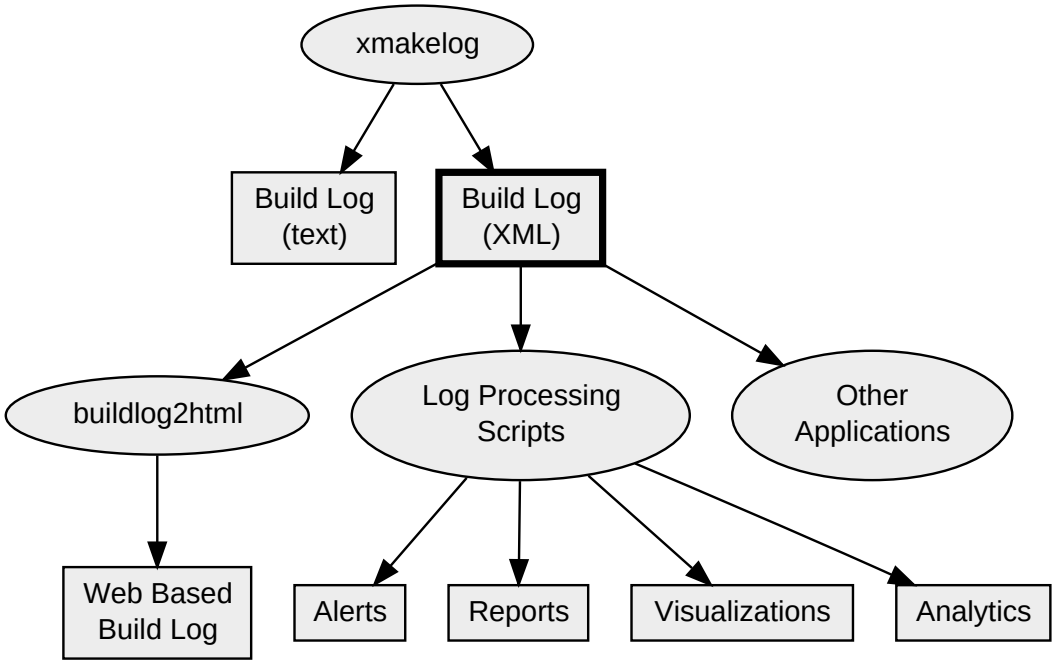
Alcatel-Lucent nmake release 11 and above can output build logs in an XML structured format. Build log structure is indicated through insertion of tags at appropriate locations in the build log, such as at beginning and end of build output for each triggered action. Structured build logs contain all the information available in traditional build logs including build command output; this basic information is augmented with additional data including nmake target names and attributes. This allows, for example, correlation of build output with the triggered target generating the output, an association not possible with traditional textual build logs. A core set of XML element types and tag attributes is available out of the box,

in addition the set of elements, attributes, and captured data is user configurable and extensible; any data accessible during a build at nmake or shell levels may be captured in the build log for use by downstream tools. This document describes features and use of the structured build feature.

nmake release 12 added support for target trigger-time capture of “explain” data, providing explanations for why certain actions were taken during a build. The sections of this document pertaining to explain data, including discussions of the `es` and `f` elements introduced to support this data, apply only to nmake 12 and up.

Downstream processing tools may operate on structured build logs to provide value-added build information and analysis. An overview of the process is shown in the following figure:

Figure 1. Process Overview



The traditional process is shown by the two nodes labeled "xmakelog" and "Build Log (text)". In the traditional approach, some variant of the nmake tool is run producing a simple textual build log. The new **xmakelog** variant adds a second output file containing the same build log but formatted as XML (highlighted in the diagram). The XML build log facilitates automated log processing and may feed into downstream applications as shown in the diagram. **xmakelog** is a front end for the **nmake** command and is an almost 100% upward compatible drop-in replacement for that command.

Build log post-processing may occur at a different time and/or on a different machine than the main build. A representative build log processing tool has been implemented that demonstrates the approach. The tool, called **buildlog2html**, provides HTML and graphical based build log visualization and is included in the nmake distribution starting with the nmake 11 release. See Section 5.2, “Visualization” for a description of **buildlog2html** and its outputs. Custom processing scripts may also process the data into useful formats. High level scripting languages ease development of such scripts; see Section 5.1, “Build Log Processing Using Scripts” for examples. The structured data might also feed into other applications such as XML viewers, outliners, and rich client applications providing advanced interactive navigation, display and build diagnostics.

This document describes the structured build log feature and provides a guide to its use.

2. Getting Started

Getting started is easy. Simply substitute the **xmakelog** command for the **nmake** command when running your build. The remainder of the **nmake** command line is essentially unchanged (several additional options are available which must precede other options on the command line). For example, instead of running

```
nmake install
```

substitute

```
xmakelog install
```

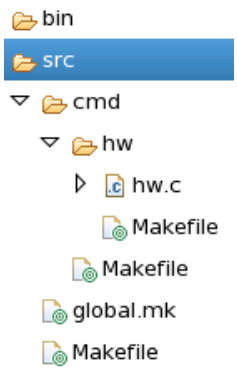
Regular build output will appear on standard output as usual. An additional file `makelog.xml` is created in the build directory containing the structured form of the build log. See **xmakelog**(1) [http://nmake.alcatel-lucent.com/manual/current/xmakelog.html] for more information about **xmakelog**.

3. Extended Example

3.1. Structure of Example

Here is a small but complete example illustrating a simple project build along with build log generation. The example uses a typical project build structure--a hierarchical layout with 3 levels of makefiles: directory `src` contains all project source, directory `cmd` under `src` contains all command source, and lowest level directory `hw` contains source for the **hw** command. Once built, commands are installed into the `bin` directory which is a sibling of `src`. The following tree diagram depicts the structure with the `src` directory highlighted:

Figure 2. Example Build Structure



There is a makefile in each level of source directory, and there is a `global.mk` in the top level directory containing definitions for the entire build. The top level `src/Makefile` contains:

```
include $(VROOT)/src/global.mk

:MAKE: cmd
```

`src/cmd/Makefile` contains

```
include $(VROOT)/src/global.mk

:MAKE: hw
```

and `src/cmd/hw/Makefile` contains:

```
include $(VROOT)/src/global.mk

hw :: hw.c
```

`src/global.mk` is included in all makefiles and defines global options for the build:

```
INSTALLROOT = $(VROOT)
recurse_begin_message=gnu
```

Although quite short, these makefiles define a complete working build due to the declarative nature of nmake and automatic provisioning of common actions such as `clobber` and `install`. To run the entire build and install the build products, **cd** to `src` and run

```
nmake install
```

The regular build log is displayed on the output:

```
make[1]: Entering directory `/a1/gms/nmb/nmake12/r12/src/doc/buildlog/ex1/src/cmd'
make[2]: Entering directory `/a1/gms/nmb/nmake12/r12/src/doc/buildlog/ex1/src/cmd/hw'
+ cc -O -I- -c hw.c
+ cc -O -o hw hw.o
+ cp hw ../../../bin/hw
make[2]: Leaving directory `/a1/gms/nmb/nmake12/r12/src/doc/buildlog/ex1/src/cmd/hw'
make[1]: Leaving directory `/a1/gms/nmb/nmake12/r12/src/doc/buildlog/ex1/src/cmd'
```

To clean up intermediate build products, run

```
nmake recurse clobber
```

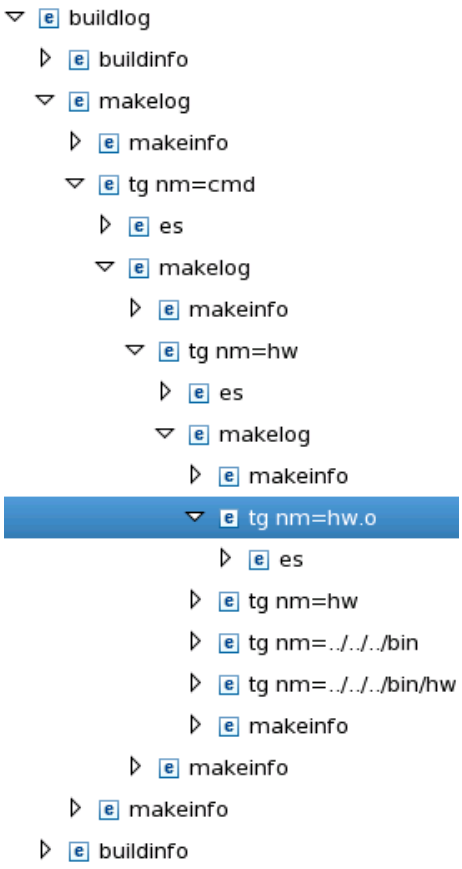
3.2. Example Build Log

To generate a structured build log for this example, simply substitute **xmakelog** for **nmake** when running a build:

```
xmakelog install
```

This produces the same regular build output, but also creates a file `makelog.xml` containing a structured form of the build log. When viewed using the Eclipse XML editor, the outline view shows:

Figure 3. Structured Build Log Outline



Outline tree items marked with an [e] indicate an XML element used to tag a part of the build output. Key elements include `makelog`, marking execution of a single makefile, and `tg`, marking execution of a single target within a makefile. For example, the target with name `hw.o`, corresponding to the build step in which `hw.c` is compiled to `hw.o`, is shown highlighted in the outline view. There are 3 nested `makelog` elements corresponding to the 3 recursion levels in the build. Also visible are `buildinfo` elements providing information about the build, and `makeinfo` elements providing information about each makefile invocation.

Textual excerpts from `makelog.xml` are shown below. Example 1, “Generated `makelog` Element”, shows the `makelog` element generated for the makefile invocation in the `src/cmd` directory. Nested `makeinfo` elements provide information specific to this makefile invocation including start time, end time, and directory.

Example 1. Generated `makelog` Element

```
<makelog>
make[1]: Entering directory `.../src/cmd'
<makeinfo>
<stime>2010-06-08T15:41:26.469285867-04:00</stime>
<pwd>.../src/cmd</pwd>
<makefile>Makefile</makefile>
<makelevel>1</makelevel>
</makeinfo>
...
<makeinfo>
<etime>2010-06-08T15:41:27.441475062-04:00</etime>
</makeinfo>
make[1]: Leaving directory `.../src/cmd'
</makelog>
```

Example 2, “Generated `tg` Element”, shows the `tg` element generated for target `hw.o`, corresponding to the same target highlighted in Figure 3, “Structured Build Log Outline”. Command output is captured as the element text content and additional target information (not present in a traditional textual build log) is captured in target attributes, including target name, start and end times, and exit code. A nested `es` element provides diagnostic information explaining why the target was triggered in a coded, fielded format.

Example 2. Generated `tg` Element

```
<tg nm="hw.o" st="2010-06-14T11:01:14.418577699-04:00" hn="pcgms2.localdomain" ec="0"
    et="2010-06-14T11:01:14.574617181-04:00">
<es><f>17</f><f>(CCFLAGS)</f><f>-O</f></es>
+ cc -O -I- -c hw.c
cc1: note: obsolete option -I- used, please use -iquote instead
</tg>
```

The captured build meta-data allows, for example, correlation of each line of build output with the triggering makefile target which is not possible using traditional build log output. In general there is a rich granular information set that may be useful for enhanced downstream build analysis and diagnostics.

Output log size is minimized since for typical builds most of the output consists of target `tg` elements which add just a few characters overhead over the typical command output. To further conserve space the `tg` element and attribute names are condensed to two characters each. Other elements and their associated meta-data such as `buildlog` and `makelog` occur only once for each build or makefile run, a relatively less frequent occurrence.

4. Description of Generated Format

As was seen in the example in the last section, the structured log file is made up of a set of nested XML elements. Key elements are:

Table 1. XML Element Description

Element	Description
`buildlog`	Top level element representing the entire log.
`buildinfo`	Container for information about the entire build.
`info`	User-defined information about the entire build.
`emapver`	Version of corresponding explain map file.
`stime`	Start time of enclosing build or makefile invocation.

Element	Description
etime	End time of enclosing build or makefile invocation.
args	Container for **nmake** command line argument list.
arg	A single **nmake** command line argument.
ecode	Exit code for entire build.
makelog	Container for a build log for a specific (possibly recursive) makefile invocation.
makeinfo	Container for information about a makefile invocation.
tg	A single triggered target.
es	Container for triggered target explain data fields.
f	A single explain data field.

Several simple informational elements appear only as children of makeinfo. These provide information relating to a specific makefile invocation. Several of these, such as vpath, do not normally change for the duration of a build, so only appear in the first output makeinfo instance (for nmake recursion level 0).

Table 2. makeinfo Sub-Element Description

Element	Description
make	Value of nmake MAKE variable.
makelib	Value of nmake MAKELIB variable.
makeversion	Value of nmake MAKEVERSION variable.
makelog_format	Value of nmake makelog_format variable.
path	Space-separated list of directories in the system PATH.
vpath	Space-separated list of directories in the system VPATH.
host	Output of command **uname -n**.
version_os	Value of nmake CC.VERSION_OS variable.
id	Output of command **id**.
pwd	Value of PWD variable.
makefile	Value of nmake MAKEFILE variable.
makelevel	Value of nmake MAKELEVEL variable.

Following is a table of tg attributes which are collected for each triggered target:

Table 3. tg Attributes Description

Element	Description
nm	Target name.
ec	Exit code from triggered action.
st	Action execution start time.
et	Action execution end time.
hn	Action execution host.

Following is a table of info attributes:

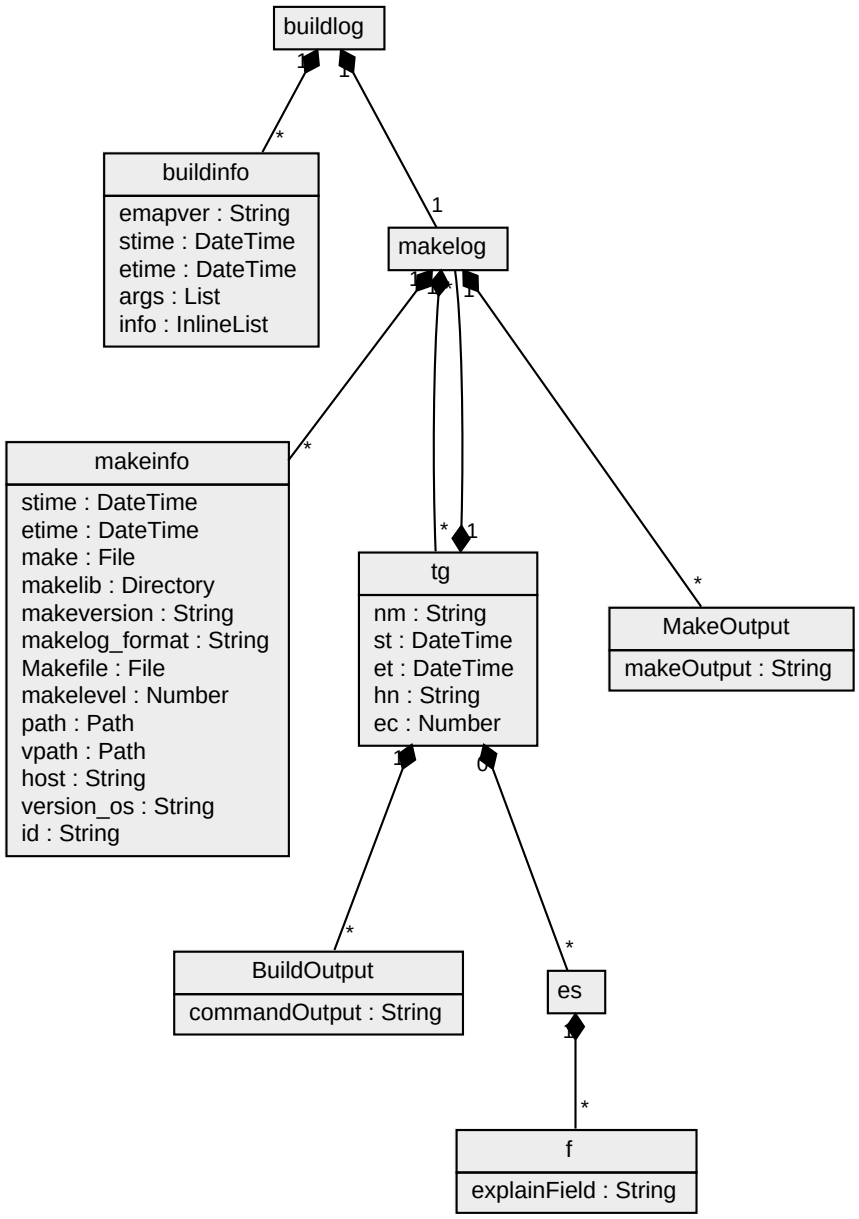
Table 4. `info` Attributes Description

Element	Description
`name`	Name of user defined data item.
`value`	Value of user defined data item.

`info` elements apply to the entire build and provide storage for user-defined build information See Section 7, “Customization and Extension” for a description of how to set these elements.

Following is a UML class diagram describing the interrelationships among the build log elements:

Figure 4. UML Class Diagram Modeling Build Log Structure



`buildlog` models an entire build log. It contains `buildinfo` elements providing information about the entire build, and a single `makelog` element modeling the top level makefile invocation. `makelog` contains `makeinfo` elements providing information about a single makefile invocation, zero or more `tg` elements representing triggered targets, and `MakeOutput` elements containing makefile level output (e.g. messages from nmake itself). A `tg` element provides information about a single triggered target, such as name and exit code, and also contains any action output (labeled in this diagram as `BuildOutput`) which may contain, for example, the output of a compile command.

Each `tg` element may contain 0 or more `es` elements, each providing a reason explaining why this target triggered. Each `es` element contains 1 or more `f` elements encoding the reason, where each `f` element contains a single field. The fielded format allows compact representation where key fields are separated out for easy automated analysis. Each explain message falls into one of a small number of message types. The first field always contains a small integer identifying the message type; the interpretation of the remaining data fields depends on the message type. Examples of field data types include strings and timestamps. Available message types and formats are described in file `lib/make/explain.map` relative to the nmake install root directory. Each line in this file represents an entry for a single message type. `explain.map` may be used by tools which process structured build logs to present explain messages in a friendly human-readable format.

For example, the line in `explain.map` corresponding to the explain message in Example 2, “Generated `tg` Element” (message type 17 as indicated by its first data field) looks like

```
17      state variable %s initialized to '%s'
```

This line provides formatting information for message type 17 as indicated by the initial numeric code. The remainder of the line contains a `printf`-like format string with 2 placeholder fields indicating (in this case) 2 strings identifying the name of a state variable and its new value. In addition to the `%s` placeholder code, there is a `%t` placeholder code indicating a UNIX style timestamp represented as seconds since the epoch (not used in this message type). Given this explain map entry, the explain message in Example 2, “Generated `tg` Element” means that the associated target `hw.o` triggered because the value of state variable `CCFLAGS` was initialized to value `-O`. See Figure 7, “Example Triggered Target Display” for an example of how a reason message might appear when formatted using the information in `explain.map`.

There is also an XML formatted file located at `lib/builddata/explain_map.xml` containing the same information as `lib/make/explain.map` but formatted for easy access by XML based tools.

As of nmake 13, every structured log contains an `emapver` element identifying the version of the explain map file in effect when the log was created. Logs created using versions of nmake prior to 13 do not explicitly provide an explain map version number, all logs without an explicit explain map version number are implicitly using explain map version `v1`. Explain maps follow an upward compatibility convention: higher explain map versions can be used to format messages in logs created under earlier versions. This means that it is sufficient to use the latest explain map for processing any version log file. The convention implies that if the meaning or number of formatting arguments in an explain message changes, the changes would be done under a new message number rather than change the meaning of an existing explain message number.

`makelog` may also contain zero or more `makelog` elements. Each such nested `makelog` element indicates a makefile invocation kicked off during action execution. This models recursive makefile invocations.

See Appendix A, *Structured Build Log DTD* for a listing of the build log DTD.

5. Applications

Explicit representation of build log structure plus capture of additional build meta-data such as target name and exit code facilitate semantic processing of the build information.

5.1. Build Log Processing Using Scripts

Here are some examples illustrating simple build log processing using scripts. Any scripting language may be used; the following sections show examples using scripting languages that provide strong XML processing features. Additional examples, including examples in Python and Perl, are available in the tools area [http://nmake.alcatel-lucent.com/tools] on the nmake web site.

5.1.1. XSLT

The first example lists data about each triggered target. Data about each target appears on a separate output line in a space separated format. Each line displays target name, execution exit code, start time, and end time. An XSLT [http://www.w3.org/TR/xslt] script generating this data is

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
 <xsl:output method="text"/>
 <xsl:variable name="nl" select="'&#xA;'"/>

 <xsl:template match="/">
  <xsl:apply-templates select="//tg"/>
 </xsl:template>

 <xsl:template match="tg">
  <xsl:value-of select='@nm' />
  <xsl:text> </xsl:text>
  <xsl:value-of select='@ec' />
  <xsl:text> </xsl:text>
  <xsl:value-of select='substring(@st, 12, 11)' />
  <xsl:text> </xsl:text>
  <xsl:value-of select='substring(@et, 12, 11)' />
  <xsl:value-of select='$nl'/>
 </xsl:template>

</xsl:stylesheet>
```

The XPath expression `//tg` recursively finds all `tg` elements in the input; then the `apply-templates` instruction causes the `xsl:template match="tg"` template to run once per `tg` tag.

We run the script from the command line using the **xsltproc** XSLT processor. **xsltproc** is a widely used command available on many platforms including Linux, Solaris, Cygwin, Windows, and in exptools. Other XSLT processors should also work. When run on the build log generated in Section 3, “Extended Example” using the command

```
xsltproc targets.xsl makelog.xml
```

the output is

```
cmd 0 17:11:21.16 17:11:21.57
hw 0 17:11:21.22 17:11:21.53
hw.o 0 17:11:21.31 17:11:21.36
hw 0 17:11:21.38 17:11:21.43
../../../bin 0 17:11:21.45 17:11:21.46
../../../bin/hw 0 17:11:21.48 17:11:21.50
```

There happen to be two targets with the same name: `hw`. In this case the two targets are in different makefiles--the action associated with the first `hw` runs the leaf level makefile `src/cmd/hw/Makefile`, and the action associated with the second runs the action that generates `hw` from `hw.o`.

Here is another XSLT example that outputs an HTML document containing a table having a row for each makefile run during the build. The fields displayed in each row are make recursion level, directory

relative to a specified root, and makefile execution start and end times. The basic structure is similar to the previous example: the `apply-templates select="//makelog"` instruction recursively finds all `makelog` elements and causes the `template match="makelog"` to run once per found element. HTML tags in the templates are output verbatim; XPath expressons such as `makeinfo/makelevel` pull out the needed parts of the input tree for output. These expressions are similar to Unix directory paths and are relative to the current XML document node, in this case the `makelog` node of the current template invocation.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
 <xsl:output method="xml" omit-xml-declaration="yes" indent="yes"/>
 <xsl:variable name="root" select="'ex1/'"/>

 <xsl:template match="/">
  <html>
   <head>
    <title>Triggered makefiles</title>
   </head>
   <body>
    <h1>Triggered makefiles</h1>
    <table>
     <tr>
      <th>makelevel</th>
      <th>pwd</th>
      <th>start time</th>
      <th>end time</th>
     </tr>
     <xsl:apply-templates select="//makelog"/>
    </table>
   </body>
  </html>
 </xsl:template>

 <xsl:template match="makelog">
  <tr>
   <td><xsl:value-of select='makeinfo/makelevel' /></td>
   <td><xsl:value-of select='substring-after(makeinfo/pwd, $root)' /></td>
   <td><xsl:value-of select='substring(makeinfo/stime, 12, 11)' /></td>
   <td><xsl:value-of select='substring(makeinfo/etime, 12, 11)' /></td>
  </tr>
 </xsl:template>

</xsl:stylesheet>
```

When run using the command

```
xsltproc makefiles.xsl makelog.xml
```

the following output is produced:

```
<html>
  <head>
    <title>Triggered makefiles</title>
  </head>
  <body>
    <h1>Triggered makefiles</h1>
    <table>
      <tr>
        <th>makelevel</th>
        <th>pwd</th>
        <th>start time</th>
        <th>end time</th>
      </tr>
      <tr>
        <td>0</td>
        <td>src</td>
```

```
        <td>17:11:21.14</td>
        <td>17:11:21.58</td>
      </tr>
      <tr>
        <td>1</td>
        <td>src/cmd</td>
        <td>17:11:21.20</td>
        <td>17:11:21.55</td>
      </tr>
      <tr>
        <td>2</td>
        <td>src/cmd/hw</td>
        <td>17:11:21.25</td>
        <td>17:11:21.51</td>
      </tr>
    </table>
  </body>
</html>
```

5.1.2. Groovy

Here are the same examples implemented using Groovy scripts. Groovy [http://groovy.codehaus.org/] is a general purpose dynamic scripting language for the JVM that provides strong support for processing XML. Groovy is tightly integrated with Java; its syntax is generally a superset of Java syntax. Since Groovy runs on the JVM, scripts may be run on platforms supporting the JVM including Linux, Solaris, and Windows.

As in the previous section, the first example lists data about each triggered target on a separate output line. A Groovy version of this script is

```
def bl = new XmlSlurper().parse(args[0])
bl.'**'.grep{it.name() == 'tg'}.each {
    println "${it.@nm} ${it.@ec} ${it.@st.text()[11..21]} ${it.@et.text()[11..21]}"
}
```

The first line parses the XML document and saves the parsed build log tree in variable `bl`. The second line sets up a pipeline-like processing chain using Groovy GPath. GPath allows navigation of hierarchical object structures much like XPath allows navigation of the structure of an XML document. The `**` step provides recursive depth first traversal of the build log object nodes, and the `grep` step filters for nodes named `tg`. The `each()` method runs the following closure for each `tg` node found. The closure formats the output using a GString expansion and sends it to standard output using `println`. Within the GString, GPath expressions are used to pick out fields for output relative to the current node stored in variable `it`. For example, `it.@nm` accesses the attribute named `nm`. The `[11..21]` subscripts extract a substring from the provided `String` object.

The script may be run from the command line using the **groovy** command:

```
groovy targets.groovy makelog.xml
```

the output is

```
cmd 0 17:11:21.16 17:11:21.57
hw 0 17:11:21.22 17:11:21.53
hw.o 0 17:11:21.31 17:11:21.36
hw 0 17:11:21.38 17:11:21.43
../../../bin 0 17:11:21.45 17:11:21.46
../../../bin/hw 0 17:11:21.48 17:11:21.50
```

Here is a Groovy script that outputs information about each makefile run during the build, similar to the second example in the last section. As in the corresponding XSLT script from the last section, results are generated in an HTML table with each row displaying make recursion level, directory relative to a root, and start and end time of makefile execution.

```
def root = 'ex1/'
def bl = new XmlSlurper().parse(args[0])
def bldr = new groovy.xml.MarkupBuilder()
bldr.html {
 head() { title('Triggered makefiles') }
 body() {
  h1('Triggered makefiles')
  table() {
   tr() {
    th('makelevel')
    th('pwd')
    th('start time')
    th('end time')
   }
   bl.'**'.grep{it.name() == 'makelog'}.each { ml ->
    tr() {
     td(ml.makeinfo.makelevel)
     td(ml.makeinfo.pwd.text().replaceFirst(~/.*$root/, ''))
     td(ml.makeinfo.stime.text()[11..21])
     td(ml.makeinfo.etime.text()[11..21])
    }
   }
  }
 }
}
```

The XML parsing and filtering steps are similar to those in the previous example. The HTML is generated using a Groovy `MarkupBuilder`. The Groovy builder mechanism provides an easy and elegant way to build nested structures without need for complex APIs. For example, the expression `head() { title('Triggered makefiles') }` creates an HTML `head` element containing a `title` element containing the textual content "Triggered makefiles". Since builder expressions are standard Groovy syntax, Groovy filtering and iteration constructs such as `bl.'**'.grep{it.name() == 'makelog'}.each { ... }` may appear intermixed with literal HTML tag generation. This allows looping over input data structures to generate variable parts of the HTML document.

When run using the command

```
groovy makefiles.groovy makelog.xml
```

the following output is produced:

```
<html>
  <head>
    <title>Triggered makefiles</title>
  </head>
  <body>
    <h1>Triggered makefiles</h1>
    <table>
      <tr>
        <th>makelevel</th>
        <th>pwd</th>
        <th>start time</th>
        <th>end time</th>
      </tr>
      <tr>
        <td>0</td>
        <td>src</td>
        <td>17:11:21.14</td>
        <td>17:11:21.58</td>
      </tr>
      <tr>
        <td>1</td>
        <td>src/cmd</td>
        <td>17:11:21.20</td>
        <td>17:11:21.55</td>
```

```
      </tr>
      <tr>
        <td>2</td>
        <td>src/cmd/hw</td>
        <td>17:11:21.25</td>
        <td>17:11:21.51</td>
      </tr>
    </table>
  </body>
```

5.1.3. Scala

Here are the same examples implemented using Scala. Scala [http://www.scala-lang.org/] is a general purpose staticly typed programming language for the JVM that smoothly integrates features of object-oriented and functional programming. Scala provides built-in parsing of inline standard XML syntax and library support for XPath style XML document queries. Programs in Scala appear to be sufficiently compact to consider its use to "script" XML processing. Since Scala runs on the JVM, it runs on platforms supporting the JVM including Linux, Solaris, and Windows.

As in the previous sections, the first example lists data about each triggered target on a separate output line. A Scala version of this script is

```
import scala.xml._
val bl = XML.loadFile(args(0))
(bl \\ "tg") foreach { i => println((i\"@nm")+" "+(i\"@ec")+" "+
 (i\"@st").text.substring(11,22)+" "+(i\"@et").text.substring(11,22))
}
```

The first line parses the XML document and saves the parsed build log tree as the value of `bl`. The second line, `bl \\ "tg"`, is a XPath-like expression that recursively selects all the `tg` elements in the document. The `foreach` iterates over each of the selected elements, printing a line of output for each element. It accesses data from each element using non-recursive XPath-like expressions indicated by the single `\` operator. For example, `i\"@nm"` accesses the value of the attribute named `nm`.

The script may be run from the command line using the **scala** command:

```
scala targets.scala makelog.xml
```

the output is

```
cmd 0 17:11:21.16 17:11:21.57
hw 0 17:11:21.22 17:11:21.53
hw.o 0 17:11:21.31 17:11:21.36
hw 0 17:11:21.38 17:11:21.43
../../../bin 0 17:11:21.45 17:11:21.46
../../../bin/hw 0 17:11:21.48 17:11:21.50
```

Here is a Scala program that outputs information about each makefile run during the build, similar to the second example in the last sections. Results are generated in an HTML table with each row displaying make recursion level, directory relative to a root, and start and end time of makefile execution.

```
import scala.xml.XML
val bl = XML.loadFile(args(0))
val root = "ex1/"
val xhtml =
<html>
 <head>
  <title>Triggered makefiles</title>
 </head>
 <body>
  <h1>Triggered makefiles</h1>
```

```
  <table>
   <tr>
    <th>makelevel</th>
    <th>pwd</th>
    <th>start time</th>
    <th>end time</th>
   </tr>
   {for (ml <- (bl \\ "makelog")) yield
   <tr>
    <td>{(ml \ "makeinfo" \ "makelevel").text}</td>
    <td>{(ml \ "makeinfo" \ "pwd").text.replaceFirst(".*"+root, "")}</td>
    <td>{(ml \ "makeinfo" \ "stime").text.substring(11,22)}</td>
    <td>{(ml \ "makeinfo" \ "etime").text.substring(11,22)}</td>
   </tr>
   }
  </table>
 </body>
</html>
XML.saveFull("out.html", xhtml, "UTF-8", true, null)
```

The XML parsing and filtering steps are similar to those in the previous example. The HTML is specified in standard XML syntax using a Scala XML literal. Scala constructs such as `for (ml <- (bl \\ "makelog")) yield ...` may be interspersed with literal HTML tags by including them between curly braces. This allows iteration over input data structures to generate variable parts of the HTML document.

When run using the command

```
scala makefiles.scala makelog.xml
```

the following output is produced:

```
<?xml version='1.0' encoding='UTF-8'?>
<html>
 <head>
  <title>Triggered makefiles</title>
 </head>
 <body>
  <h1>Triggered makefiles</h1>
  <table>
   <tr>
    <th>makelevel</th>
    <th>pwd</th>
    <th>start time</th>
    <th>end time</th>
   </tr>
   <tr>
    <td>0</td>
    <td>src</td>
    <td>17:11:21.14</td>
    <td>17:11:21.58</td>
   </tr><tr>
    <td>1</td>
    <td>src/cmd</td>
    <td>17:11:21.20</td>
    <td>17:11:21.55</td>
   </tr><tr>
    <td>2</td>
    <td>src/cmd/hw</td>
    <td>17:11:21.25</td>
    <td>17:11:21.51</td>
   </tr>
  </table>
 </body>
</html>
```

Scala can also pattern-match XML structures providing a powerful way to process and transform XML data (not illustrated in above examples).

5.2. Visualization

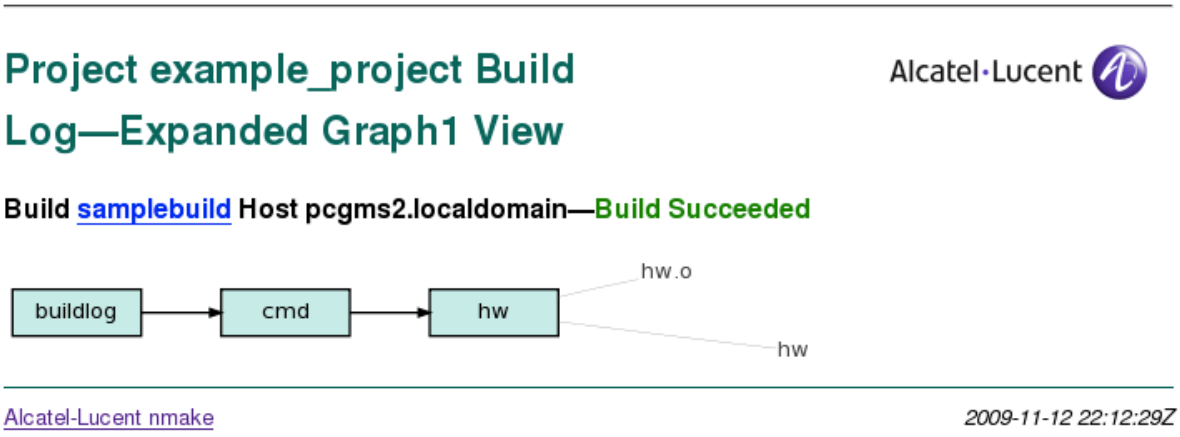
It is possible to provide various views into the build at multiple levels by transforming the structured data into appropriate presentation formats using techniques such as those presented in Section 5.1, “Build Log Processing Using Scripts” As one example, the previously mentioned tool **buildlog2html** transforms a given build log into a set of interlinked HTML pages providing overview and detailed views of the build log. Color coding is used to highlight failed targets in these views.

High level views clearly show the overall hierarchical structure of the build, leaving out detailed data about each of the targets. Following is a high level view of the example build of Section 3, “Extended Example” that uses nested HTML list elements to show the overall shape of the build:

Figure 5. Example HTML Build Log Visualization

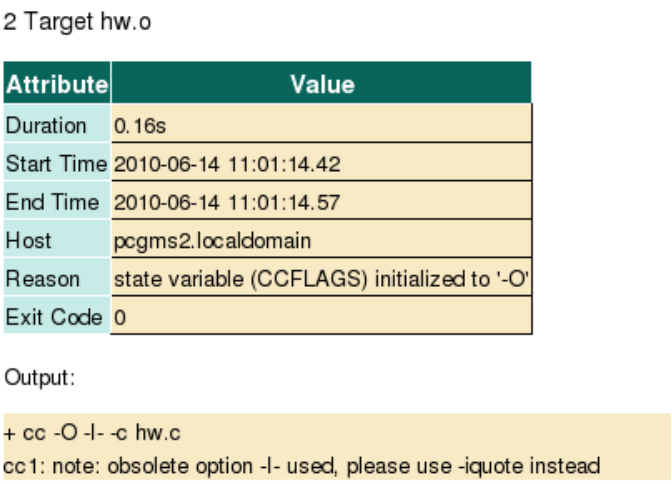
Following is a similar view in a graphical format as generated by **dot** which is part of the AT&T Graphviz package:

Figure 6. Example Graphical Build Log Visualization



Targets in both the HTML list-based view and the graphical views link to per-makefile pages providing details about the makefile execution and the triggered targets. For example, the `hw.o` item in both above examples links to a view providing detailed information about the triggered target. As shown in the following figure, target attributes are presented in a tabular layout followed by a text block displaying rule action block output. The information shown in this example corresponds to the build log structure shown in Example 2, “Generated `tg` Element”. Note that the reason string is presented in a formatted readable form rather than the fielded form in which it appears in the build log.

Figure 7. Example Triggered Target Display


2 Target hw.o

Attribute	Value
Duration	0.16s
Start Time	2010-06-14 11:01:14.42
End Time	2010-06-14 11:01:14.57
Host	pcgms2.localdomain
Reason	state variable (CCFLAGS) initialized to '-O'
Exit Code	0

Output:

```
+ cc -O -I- -c hw.c
cc1: note: obsolete option -I- used, please use -iquote instead
```

A simple multi-build index is also generated providing summary information about each build with a link to the per-build pages. An extended **buildlog2html** example using a realistic sized build is shown in the nmake 11 release notes [http://nmake.alcatel-lucent.com/rnotes-11/s2.html#s2.2]. See the **buildlog2html manual page**(1) [http://nmake.alcatel-lucent.com/manual/current/buildlog2html.html] for more information about using **buildlog2html**. **buildlog2html** is included in nmake starting with the nmake 11 release.

Additional examples including examples using Javascript (Protovis library) are available in the build reports [http://nmake.alcatel-lucent.com/tools/reports.html] area on the nmake web site.

6. Data Collection Performance

An important application of structured build logs is collection of data associated with build targets for later analysis. For certain types of analysis, such as build performance analysis, it may be desirable to optimize data collection performance in order to minimize data collection overhead. This section discusses optimization of the data collection process itself.

The following optimizations have been found to be effective in optimizing build data collection performance:

- Use nmake 16 or greater. nmake 16 introduced a streamlined, optimized **nmakelog**/**xmakelog** implementation that can cut elapsed time and CPU time overheads by almost 50%.

- Minimize data items captured to just those items of interest. It is especially important to minimize those data items requiring a per-job process spawn to obtain the data. 25-30% reductions in both elapsed time and total CPU time overhead have been observed by restricting data items collected to just target name, start, and end time. Selective data collection is easily configurable; to collect just name, start time, and

end time for each triggered target, set `makelog_format='xml:tg/@nm,@st,@et'` in the build environment.

- Use ksh built-ins to collect job start time and end time. This avoid 2 process spawns per target run leading to observed reductions in elapsed time and total CPU time of 20-30%. The command used to acquire time information is easily configurable, for example, to collect time information using the built-in ksh print command, set variables `XSDATE=print` and `XSDATEFLAGS='-f %(%Y-%m-%dT%H:%M:%S.%N%z)T'` in the build environment. The print date/time built-in is available (at least) in ksh93r and newer. The built-in is supported by the system `/bin/ksh` in RHEL 4 and above, and also by `ksh93` in exptools (in directory `/opt/exp/bin`).

  If needed, configure the build to use a ksh providing this feature. For example, on Centos 6.5, set `COSHELL=/bin/ksh` in the build environment. It might also be necessary to run **xmakelog** with a ksh supporting the built-in print feature, for example: `$KSH xmakelog`. If using coshell, the `COSHELL` setting is `COSHELL=coshell`. In this case, a suitable **ksh** should be located first in the `PATH` to ensure that coshell finds and runs it.

Putting it all together, a script configuring **xmakelog** for optimized data collection performance might look something like:

```
# run xmakelog minimizing time collection overhead
KSH=/bin/ksh
export COSHELL=${COSHELL:-$KSH}
export XSDATE=print
export XSDATEFLAGS='-f %(%Y-%m-%dT%H:%M:%S.%N%z)T'
export makelog_format='xml:tg/@nm,@st,@et'
set -x
$KSH xmakelog "$@"
```

With all 3 optimizations in effect, reductions in overall collection overhead of 70-80% have been observed. At this point, data collection with **xmakelog** runs with essentially the same performance as **nmakelog**. With the optimizations in place, both **xmakelog** and **nmakelog** have been observed to run within 3% of stock **nmake** in elapsed time (wall clock), and within 6% of stock **nmake** in total CPU time used. Additional optimizations appear possible, but have not yet been explored. Since **nmakelog** does not collect data and does not use XML, one, perhaps unexpected, conclusion is that, at least for the measured configuration, use of XML format to represent collected data has minimal impact on performance. Experiments indicate that per-job process spawns are the main driver of data collection performance.

On nmake 15 and before, the ksh included in the nmake distribution has issues resulting in larger elapsed run times and low CPU utilization on RHEL 6.5. The problem was not observed on other releases of RHEL, or on other platforms such as Solaris. Use of the system ksh on RHEL 6.5, or use of nmake 16 or above on RHEL 6.5, does not have the issue. Use of nmake 15 or below with the nmake ksh on RHEL 6.5 is not recommended.

7. Customization and Extension

Several customizations and extensions are available that provide control over the content of the generated build log and other aspects of the process.

7.1. `info` Elements

As discussed in Section 4, “Description of Generated Format”, information about the entire build is stored in the `buildinfo` element. `buildinfo` can contain `info` elements that can store arbitrary name/value pairs provided by the user at build time. These can be use to store project specific identifiers such as project name and build id. Specify these at build time with the **`--info`** option to **xmakelog**, for example:

```
xmakelog --info project:myproj --info build:20100301
```

This example specifies 2 `info` elements with names `project` and `build`. These elements go into the build log and are available to downstream processors. **buildlog2html**, in particular, will use the values of several specific `info` elements if present:

Table 5. `info` Elements Recognized by buildlog2html

Name	Description
`project`	Project name for use in page headers etc.
`build`	Build name for use in human readable labels etc.
`project-id`	Unique project id for use in filenames etc.
`build-id`	Unique build id for internal use.

For **buildlog2html**, in general only `project` is needed. **buildlog2html** will generate reasonable values for other items if not provided.

7.2. Environment Variable `makelog_format`

Environment variable `makelog_format` allows customization of the generated build log format. `makelog_format` is currently used to select attributes of triggered targets (`tg` elements) to include in generated output. It has the general form `xml:` ***`elt1/@attrib11,@attrib12,...;elt2/@attrib21,@attrib22,...`*** . Essentially, it is a list of element types separated by semicolons, where each element type may be followed by a slash and a list of selected attributes separated by commas. Currently 2 elements are recognized: `tg` and `es`. The `tg` element is used to specify which target attributes should be output, and the `es` element controls whether to output explain data. Note that semicolons need to be escaped from the shell when providing a specification string on the shell command line.

Some examples are shown in the following table:

Table 6. `makelog_format` Examples

Example	Description
`xml:`	Create XML formatted log. Each target element has a default set of commonly used attributes. Explain data is output.
`xml:tg/@nm,@ec;es`	Create XML formatted log with each target element having an `nm` attribute attribute and an `ec` attribute. Explain data is also output.
`xml:tg/@nm,@ec`	Create XML with each `tg` element populated with the complete set of predefined attributes. Explain data is suppressed.

See Table 3, "`tg` Attributes Description" for a list of the set of available `tg` element attributes. As an example, specifying **`xml:tg/@nm,@ec`** would result in tags of the form `<tg nm="abc" ec="0">`. `makelog_format` defaults to **`xml:tg/@nm,@hn,@st,@et,@ec;es`** if not specified.

7.3. User Defined Build Log Extension

Since build log tags are fully specified in the nmake baserules, just about any aspect of the generated format can be changed by overriding or extending the baserules. Here are examples of user defined format

extensions adding new attributes for the `tg` element. The approach allows capture of arbitrary information from nmake and/or shell variables, functions, and commands.

The first example captures the relative time of the current nmake target into a new attribute. This can be done by appending a definition of the new attribute to the `targ_attr_defs_begin` nmake variable. `targ_attr_defs_begin` contains attribute definitions to be expanded at the beginning of a triggered action; `targ_attr_defs_end` contains attribute definitions to be expanded at the end of a triggered action. Here is a complete makefile example with the added lines highlighted:

```
.INIT : .add_attrs

.add_attrs : .VIRTUAL .MAKE .FORCE
 targ_attr_defs_begin += rtime="$$(<:T=ZR)"

a :: a.c
```

Assuming this makefile is in the file `nm.mk`, when run using the command:

```
xmakelog -f nm.mk makelog_format=xml:tg/@nm,@hn,@st,@et,@ec,@rtime
```

we get additional attributes on the `tg` elements in `makelog.xml` of the form rtime="1266350680". Note that we need to add the name of the new attribute to the `makelog_format` variable to select the new attribute for inclusion.

Since the attribute definition is also expanded by the shell, we can capture the values of shell variables and command output. Here is an example that captures the output of the **uptime** command:

```
.INIT : .add_attrs

.add_attrs : .VIRTUAL .MAKE .FORCE
 targ_attr_defs_begin += uptime='\""$$$(uptime)"\"'

a :: a.c
```

Assuming this makefile is in the file `sh.mk`, when run using the command:

```
xmakelog -f sh.mk makelog_format=xml:tg/@nm,@hn,@st,@et,@ec,@uptime
```

we get new attributes on the `tg` elements of the form:

```
uptime=" 15:28:03 up 15:54,  1 user,  load average: 0.60, 0.37, 0.27"
```

This captures system load average data at the start time of each triggered target.

As a final example we show how to execute an arbitrary shell command, passing in values from the environment of the triggered action:

```
ALERT = alert

.INIT : .add_attrs

.add_attrs : .VIRTUAL .MAKE .FORCE
 targ_attr_defs_end += alert='\""$$$($(ALERT) $$(<) $ecode)"\"'

a :: a.c
```

This defines a new attribute called `alert` containing the output of the **alert** command. **alert** is a user written executable shell script located on the `PATH`, and is passed the name of the current target and the exit code of the current action. A simple example of **alert** might be:

```
# alert - send an alert of problem during a build

targ=$1
```

```
rcode=$2

if [[ $rcode != 0 ]]
then
 echo target $targ rcode $rcode >> ~/tmp/build.err
 # echo target $targ rcode $rcode | mailx -s 'build alert' $USER
 echo $?
fi
```

This logs build errors in a file. Alternatively, the command could send an email or other indicator providing real-time notification that a build problem occurred. The makefile is run using command

```
xmakelog -f alert.mk makelog_format=xml:tg/@nm,@hn,@st,@et,@ec,@alert
```

This will put `alert` attributes on all `tg` elements, containing the return code from the alert procedure if the action failed.

8. Notes

The feature is implemented on top of the existing makelog functionality. This has the advantage that it works with concurrent builds, and also exploits the implicit "tee" in makelog. Structured logs containing tags are generated into a secondary file called `makelog.xml` (in the case of XML structured files) while regular unstructured build command output appear on standard output as usual. `makelog.xml` is created in the build directory from scratch for each run. This is different from the makelog file created by the **nmakelog** command which appends to an existing makelog file. The structured build log feature is implemented such that there is essentially no performance impact if the feature is not used.

Additional filtering may be performed while writing the structured log file. In the case of XML format, XML special characters are translated as appropriate. Also, `tg` attributes generated when an action block completes are combined with attributes generated before the action starts and output together as attributes on the corresponding `tg` element.

9. Summary and Conclusions

We have described an approach to capture of nmake build logs in an XML structured format.

XML seems ideally suited for representing build logs in a form suitable for automated processing. Its hierarchical structure naturally models the recursive structure of actual builds. XML text nodes are well suited to capture of action block output, and XML attributes naturally represent meta-information about parts of a build such as target name, error code, and timing data. Overheads seem reasonable given that the most common target elements have been optimized. An important advantage of use of XML is wide availability of XML processing tools, languages, and libraries allowing convenient development of processing scripts.

A. Structured Build Log DTD

```
<!ELEMENT buildlog (#PCDATA | buildinfo | makelog)*>
<!ELEMENT makelog (#PCDATA | makeinfo | tg)*>
<!ELEMENT buildinfo (stime | info | args | ecode | etime | emapver)*>
<!ELEMENT info EMPTY>
<!ELEMENT args (arg)*>
<!ELEMENT makeinfo (make | makelib | makeversion | makelog_format | pwd |
    vpath | path | host | version_os | id | stime | makefile | makelevel |
    etime)*>
<!ELEMENT tg (#PCDATA | makelog | es)*>
<!ELEMENT es (f+)>
```

```
<!ELEMENT f (#PCDATA)>

<!ELEMENT arg (#PCDATA)>
<!ELEMENT ecode (#PCDATA)>

<!ELEMENT make (#PCDATA)>
<!ELEMENT makelib (#PCDATA)>
<!ELEMENT makeversion (#PCDATA)>
<!ELEMENT makelog_format (#PCDATA)>
<!ELEMENT pwd (#PCDATA)>
<!ELEMENT vpath (#PCDATA)>
<!ELEMENT path (#PCDATA)>
<!ELEMENT host (#PCDATA)>
<!ELEMENT version_os (#PCDATA)>
<!ELEMENT id (#PCDATA)>
<!ELEMENT stime (#PCDATA)>
<!ELEMENT makefile (#PCDATA)>
<!ELEMENT makelevel (#PCDATA)>
<!ELEMENT etime (#PCDATA)>
<!ELEMENT emapver (#PCDATA)>

<!ATTLIST info
  name CDATA #REQUIRED
  value CDATA #REQUIRED
>

<!ATTLIST tg
  nm CDATA #IMPLIED
  st CDATA #IMPLIED
  hn CDATA #IMPLIED
  ec CDATA #IMPLIED
  et CDATA #IMPLIED
>
```

As of nmake 13, the structured build log DTD is included in the distributed nmake package at location `lib/builddata/buildlog.dtd` relative to the install location root.

References

[1] *nmakelog, xmakelog - run nmake producing serialized build log*. nmake manual page. http://nmake.alcatel-lucent.com/manual/current/xmakelog.html.

[2] *buildlog2html - transform XML build logs to HTML*. nmake manual page. http://nmake.alcatel-lucent.com/manual/current/buildlog2html.html.