Transbuild

Tutorial and Reference Manual
Version 0.8.2

Hoylen Sue

Table of Contents

1 Introduction

Transbuild is a program for generating and managing Web sites. It is targeted at small to medium sized Web sites which contain mostly static content. It can be used to automate some of the repetitive tasks associated with creating, updating and maintaining a Web site. If you have two or more pages on your site, you may find Transbuild useful!

Transbuild uses XML and XSLT to generate a Web site. Content is created and stored in XML, a presentation independent form. Transformations are applied to those files to build the Web site. Changes to navigation and presentation styles can be performed throughout the entire site by simply changing the XSLT script once and regenerating the entire Web site. Transbuild makes it easy to maintain a consistent Web site.

1.1 Scope and purpose

This document is the user's manual for Transbuild. It has a tutorial and a reference section. It is written for people who wish to use Transbuild to create and manage Web sites.

The reader should understands XML and is competent in writing XSLT scripts. They should alsk know how to create Web sites using HTML or XHTML. It is hoped that they also know how to create good Web sites, which are standards compliant, usable and accessible!

1.2 Philosophy behind Transbuild

Transbuild is a simple program based around some simple concepts. However, putting those simple concepts together in different ways can achieve powerful results. The simplicity also allows you to use Transbuild in conjunction with other programs. The concepts behind Transbuild are:

Use XML and XSLT as the foundation.
: The XML format is simple and widely deployed. It is a presentation independent format, allowing the style of the Web site to be easily changed without needing to change the source content. In the XML world, XSLT is the standard transformation language to use.

Store content as normal files.
: The content is stored as ordinary files in ordinary directories. No special database nor server is needed. This allows you to use whatever tool you want to manipulate the content - all programs support files in directories. For example: you can choose to use a text editor or a specialised XML editor to edit the content; version control can be handled using existing tools like CVS; archives can be made using tar, gzip, or zip.

Avoid storing redundant data.
: Redundant data means more work for the maintainer to keep up to date and to keep it consistent. Ideally, making a single change to the site would require only a single change to the source. This is especially important when dealing with site navigation links, which must appear consistently in many pages on the site.

Generate static files.

The output produced by Transbuild is a set of ordinary files in ordinary directories. You don't need to run any special packages, databases or content management systems on the server. The files can be uploaded to any Web site. Files are also simpler for a Web server to handle: unlike on-the-fly content management systems, no processing overhead is required to serve them; Web caches can handle them better; and they are simpler to deploy.

2 Overview

This section is an overview of how Transbuild works and how it can be used to generate Web sites.

2.1 Source to target

The basic operation Transbuild performs is to take files and directories from under a source directory and generates a set of files and directories under a target directory. The set of subdirectories and files will be called a "tree." Think of it as a fancy recursive directory copy. Each subdirectory in the source tree is recreated in the target tree. Each source file creates a corresponding target file.

The recursive copy doesn't just copy files, but can transform them. The most common way is to apply an XSLT script to the source file to create the target file. For example, taking every individual XML file in the source tree and transforming them to HTML files in the target tree. It can also detect which source file has been changed, and only rebuild that file. Think of it as a fancy version of `make`, or a program that applies XSLT to a batch of files.

Rules can be written so that different types of files can be treated differently, applying a different set of transformations to them. Also, the names of the files can be modified according to rules. For example, '`.xml`' source files can be renamed to '`.html`' target files after applying the transformations.

2.2 Annotations

Transbuild allows access to context sensitive information from the source tree to create a richly interlinked set of target files.

Normally, XSLT transformations are applied to one XML source file. With Transbuild, they can be applied to the source file plus a set of annotations. These annotations come from the other source files and directories near the source file being processed.

Many items can be easily accomplished using annotations. The list of immediate subdirectories can be used to generate links to subsections in the Web site. Navigation links to higher sections can be found by examining each parent and ancestor directories.

Annotations are dynamically determined from the structure and contents of the source tree. This makes constructing and maintaining navigation links a breeze. In most other content management packages, you would have to manually enter hard-coded navigation links - simple site-wide changes can be very costly. With Transbuild, all of this information is dynamically obtained - site-wide changes can be very easily done.

For example, adding a new section to the Web site is simply done by adding the extra directories and files for that section. The annotations mechanism will automatically detect the new material and generate the navigation links to it. Just create the files: no need to perform any reconfiguring. The same applies to moving a file to another subdirectory. The annotations will pick up its new context, and all the links will be automatically readjusted.

2.3 Flexibility

Transbuild does not impose any particular format for your source files. You can use any XML vocabulary you want, and write your own XSLT scripts to process them.

You don't even have to use XML. In addition to XSLT processing, you can run arbitrary Unix commands to perform the transformations (use Perl, AWK, grep, whatever).

Combine multiple transformations together into a filter chain. Mix XSLT with custom programs. Transbuild is very flexible.

2.4 Comparison to other tools

Transbuild brings together several existing ideas into a single useful tool. This section lists some other similar tools, and briefly compares them to Transbuild.

The Java http://www.ananas.org/xm/ XM (XSLT Make) program is similar to Transbuild in taking a source directory of files and building the target Web site pages from it. However, each source XML file is treated individually. It does not have an annotations mechanism, so generating navigation links between the pages is not possible.

The Web Meta Language (WML) http://www.thewml.org/ is similar to Transbuild in that it uses a source file to automatically generate a set of files. Instead of using XSLT, it uses a series of processing steps that combines Perl, M4 and proprietary HTML processing.

Zope, PHP, JSP, AxKit, Cocoon, are tools that many Web developers have heard of. However, they are in a different category from Transbuild because they require a server to be installed and/or the content to be stored in a specific database or format. You cannot build and test the site locally without the server. Some of the tools need low level programming and integration to get a working site up and running.

2.5 Summary

Combining structured data, transformations, and dynamic annotations gives you a powerful tool to build and maintain a Web site. You can ensure that your site is consistent - that all pages have the design you want. You can manage changes better, and perform site-wide changes without the tedium of editing every single page in the site.

To see what Transbuild can do, have a look at the test examples or follow the tutorial to see how it is done.

3 Tutorial

This tutorial describes how Transbuild is used to build a Web site. It will describe the process step-by-step, building the site up from scratch and showing how the main features of Transbuild can be used.

You will find the tutorial files under the '`test/tutorial`' directory. Each step has its own build script file, and you can run it with a command like the following:

```
$ cd test/tutorial
$ transbuild -a -f tb-step01.xml
```

The '`-f`' option is used to specify the build script file to use. If it is not specified, the program will try to look for a file named '`Transbuild.xml`' in the current directory. In this tutorial, the build script files will not be called this, so you will need to explicitly specify it with the '`-f`' option.

The '`-a`' option tells Transbuild to build every file. Normally, Transbuild will only build a target file if doesn't exist or its corresponding source file is newer. This dependency checking can speed up rebuilds. However, it is not flawless: it fails to detect changes to associated files (the annotation files described later), changes to the XSLT scripts, and changes to the build script itself. During this tutorial, most of the changes will be to the XSLT scripts and the build script, so you will need to use the '`-a`' option to ensure that the target files are updated.

Each step will produce a target tree in the same directory called '`target`'. You might want to delete the target directory between runs, so that old files do not clutter up the target tree. You can specify a different target directory with the '`-T`' option. For example:

```
$ transbuild -a -f tb-step02.xml -T target02
```

3.1 Step 1: Creating the source tree

The first step is to create the source tree. This is a single directory that contains the files and subdirectories to be transformed. In this document, we will refer to a directory and everything nested under it as a "tree," and use the term "directory" to mean a single directory.

Before you begin a new site, there are three main decisions you should make. You could change your mind later on, but deciding these issues early on will make your site easier to manage. The issues are:

The navigation structure of the Web site and how it is logically laid out. This will influence the directories you create under the source tree. Getting the directory structure right will make it easy for the site to automatically generate the navigation links.

The types of pages in the site and the XML vocabularies used in the source files. Transbuild does not restrict the types of source documents you use: for example, you could use a standard schema like DocBook, use something very close to the site pages like XHTML, use a standard data oriented schema, or create your own XML vocabulary. A site may use only one XML schema for all its files, or have many different types - it's your choice.

The naming convention for the files in the source tree. This is important, especially when using more than one XML schema, so that Transbuild can distinguish between

the types of files and process them differently. Each directory should have a master file (with exactly the same name) which will be used as the destination when generating navigation links. This master file is usually the same one that will generate the '`index.html`' file of that directory.

In this tutorial, we will be creating a Web site for a small company. The site will have a number of major sections, some (such as the hardware and software sections) will be further divided into subsections. The directory structure used is shown below. The symbol '`~source~`' will be used to refer to the top directory of the source tree, and '`~target~`' for the top directory of the target tree.

```
~source~
 +- about/
 +- hardware/
 |   +- omp/
 |   +- mp100/
 |   +- mp110/
 |   +- mp120/
 |   +- mp130/
 +- software/
 |   +- nos10/
 |   +- nos11/
 |   +- nos13/
 +- contact/
```

Initially, there will be only one type of page, but later on in the tutorial we will add more. A very simple XML vocabulary has been invented for this tutorial. It's DTD is shown below:

```
<!ELEMENT article (title, para*)>
<!ATTLIST article status CDATA #IMPLIED>
<!ELEMENT title (#PCDATA)>
<!ELEMENT para (#PCDATA | ulink)>
<!ELEMENT ulink (#PCDATA)>
<!ATTLIST ulink url CDATA #REQUIRED>
```

We have decided to give all our source XML files names with '`.xml`' extensions (even if they contain different XML vocabularies). This is a good choice if your editor only recognises files with that extension. The other alternative is to use different extensions for files containing different XML vocabularies. The master file in each directory will be called '`index.xml`'. An example of one of these files, the file '`~source~/hardware/index.xml`', is shown below. (The mysterious `status` attribute will be explained later in the tutorial.)

```
<?xml version="1.0"?>
<article status="2">
<title>Hardware</title>
<para>Our products incorporate leading edge technology with award
winning design which is both beautiful and functional.
</article>
```

In the tutorial directory, you'll find this source tree in a directory called '`source-a`'. It has the directory structure described above, one '`index.xml`' file under each directory, and a '`site.css`' file under the source tree root directory.

Transbuild takes the source tree and generates a target tree from it. The generating process is controlled by a build script file. The first build script is very basic, and is found in the file '`tb-step01.xml`'.

```
<?xml version="1.0"?>

<build-script
 xmlns="http://hoylen.com/ns/xmlns/2002/transbuild/buildscript"
 version="1.0"
 source="source-a"
 target="target">

<rule source-suffix=".xml">
  <file-copy/>
</rule>

<rule source-suffix=".css">
  <file-copy/>
</rule>

<rule source-suffix="~"/>

</build-script>
```

The build script is an XML file containing elements from the Transbuild build script namespace of `http://hoylen.com/ns/xmlns/2002/transbuild/buildscript`. The root element is the `build-script` element. It contains a mandatory `version` attribute, which should have the value of `1.0`. It also contains the `source` and `target` attributes which specifies the source tree and target tree. Their values are directory names, specified relative to the location of the build script file. These values can be overridden by options on the command line. The contents of the `build-script` element is an ordered sequence of rules.

There are three rules in this first build script. A rule is represented by a `rule` element. The contents of the `rule` element is an ordered sequence of processing steps which determine how the contents of a file will be processed. The attributes of the `rule` element determines which files from the source tree will use that rule (and will be discussed in the next step of the tutorial).

The most basic rule is a `rule` element with empty content. Files in the source tree which match this empty rule are ignored - they do not cause any file to be generated in the target tree. In this example, the last rule is used to ignore backup files created by the emacs editor - these backup files have names ending in a tilde ('`~`').

The next basic rule is one that contains a single `file-copy` element in it. This causes matching files in the source tree to be copied to the target tree. The file contents are not changed in any way. In this example, the other two rules copy the XML and CSS files to the target tree.

This first step can be performed by running Transbuild with the following command:

```
$ transbuild -a -f tb-step01.xml
```

The output printed by Transbuild will show that it first creates the target directory and all the directories under it, and then builds the files under the target tree. The directories

will only be created if they do not already exist, so if you run this command a second time the directories won't be recreated (unless you have deleted them).

If you get an error about character encodings, see the section on character sets in the reference section.

The fundamental behaviour of Transbuild is to take the source tree and to generate the target tree from it. The directory structure is always replicated exactly. One target file is built from each source file. This simple one-to-one relationship is Transbuild at its purist. There are advanced ways to break it, but they are beyond the scope of this tutorial.

This first step has used Transbuild to create a target tree that is an exact replica of the source tree. Basically, it performs a recursive copy (minus any backup files). Not very useful, but it is a start that will be built upon in the next step.

3.2 Step 2: Performing a transformation

Transbuild gets more interesting when source files are processed to build target files. In this step, we will transform the XML files using XSLT. Modify the first rule from the previous step by change it into:

```
<rule source-suffix=".xml" target-suffix=".html">
  <xslt stylesheet="script/tr02.xsl"/>
</rule>
```

When a source file is processed, it finds the first rule in the build script where the `source-suffix` attribute matches the end of the filename. This is a exact literal character-by-character match, and characters such as '`.`', '`*`', and '`?`' do not have any special meaning.

The name of the generated target file is created by taking the original source file name, removing the `source-suffix` from the end and appending the `target-suffix`. If the `target-suffix` is exactly the same as the `source-suffix`, the filename will be unchanged (in this case, the `target-suffix` attribute can be left out - as in the rule that matches the CSS file). The target file is always created in the same directory in the target tree as the source file is in the source tree.

All the files in the source tree must match a rule, otherwise an error will be generated. This way, no unexpected files will end up in the target tree.

To create the target file, the contents of a source file is processed by the list of processing steps in the rule. The output from each processing step becomes input to the next processing step - like a pipeline. The input into the first processing step is the contents of the source file. The output from the last step is saved into the target file. The `file-copy` processing step can be thought of as a straight-through processing step that does not change the data.

The `xslt` processing step is used in this example. As you may guess, it applies a XSLT transformation to the data (which must be in XML format). The `stylesheet` attribute specifies the XSLT file relative to the location of the build script. In this step, the '`tr02.xsl`' XSLT script converts the XML file into a simple XHTML Web page.

```
<?xml version="1.0"?>

<!-- A very simple transformation of the XML into XHTML -->

<xsl:stylesheet version="1.0"
```

```
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns="http://www.w3.org/1999/xhtml"
>

<xsl:output method="xml"
    encoding="UTF-8"
    standalone="yes"
    doctype-public="-//W3C//DTD XHTML 1.0 Strict//EN"
    doctype-system="http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"
    media-type="text/xhtml"
    indent="yes"
    />

<xsl:template match="/">
  <html xml:lang="en" lang="en">
    <xsl:comment>Do not edit: generated by Transbuild</xsl:comment>
    <head>
      <meta http-equiv="Content-Type"
            content="text/xhtml; charset=UTF-8"/>
      <title>Step 2: <xsl:value-of select="article/title"/></title>
    </head>
    <xsl:apply-templates/>
  </html>
</xsl:template>

<xsl:template match="article">
  <body>
    <h1><xsl:value-of select="title"/></h1>
    <div class="main">
      <xsl:apply-templates/>
    </div>
  </body>
</xsl:template>

<xsl:template match="title">
  <!-- ignore, since already used in h1 -->
</xsl:template>

<xsl:template match="para">
  <p><xsl:apply-templates/></p>
</xsl:template>

<xsl:template match="ulink">
  <a href="{@url}"><xsl:apply-templates/></a>
</xsl:template>

</xsl:stylesheet>
```

The `meta` tag was added because some browsers (e.g. Opera) do not pick up the character encoding from the XML declaration.

Running Transbuild on the second build script, '`tb-step02.xml`', produces a target tree containing a collection of XHTML files (with '`.html`' file extensions) and a CSS file that is a direct copy of the one in the source tree.

```
$ transbuild -a -f tb-step02.xml
```

Transbuild does not delete any files from the target tree. So if you run this after running the first step, in addition to the new '`.html`' files you will find the target tree cluttered up with the old '`.xml`' files. It is recommended that you delete the entire target tree to start afresh (`rm -r target`).

These XHTML Web pages are good, but they are still individual pages. A Web site must be connected together with hyperlinks. In the next step, links between the pages will be added.

3.3 Step 3: Making links

Hyperlinks can be easily added using the XSLT stylesheet. However, linking to other pages is complicated by the fact that the target files might have a different name to the source files. This problem is solved by a filename mapping XPath function provided by Transbuild. To use it, you will first need to declare the function's namespace `http://hoylen.com/ns/xmlns/2002/transbuild/function` in the XSL stylesheet. And since you don't want this namespace declaration to appear in the generated XHTML, add it's prefix to the `exclude-result-prefix` XSLT attribute.

```
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns="http://www.w3.org/1999/xhtml"
xmlns:TBF="http://hoylen.com/ns/xmlns/2002/transbuild/function"
exclude-result-prefixes="TBF"
>
...
```

Transbuild provides an XPath function called `href` which converts a source tree name into its counterpart in the target tree. Its argument can be a name relative to the file being processed, or an absolute name. In this context, "absolute" means treating the source tree directory as the root directory (not the file system's root directory). It works with directory names as well as filenames, but most of the time we'll be using it with filenames. We will refer to the function as `TBF:href` so it won't be confused with other hrefs. Here's the modification we made to the XSLT script to add a link to the site's home page.

```
<xsl:template match="article">
  <body>
    <div class="nav-top">
      <ul>
        <li><a href="{TBF:href('/index.xml')}">Home</a></li>
      </ul>
    </div>

    <h1><xsl:value-of select="title"/></h1>

    <div class="main">
```

```
      <xsl:apply-templates/>
    </div>
  </body>
</xsl:template>
```

The mapping function will give the correct filename based on the name translation described in the build script. In this case, the function will take '`index.xml`' and return '`index.html`'. If you changed the rule's name translation in the build script (for example, to have a `target-suffix` of '`.htm`'), the function will ensure everything still works.

The home link was placed in a unordered list. This is so the list of links will be accessible to non-CSS aware browsers. Later on, there will be more links in the list. Using CSS, the list will not look like a list on the Web page.

We'll also make the following changes to create a link to the CSS file. We know the file name of the CSS file is not going to be changed, but still use the `TBF:href` function because it has an additional function besides just name translation.

```
<xsl:template match="/">
  <html xml:lang="en" lang="en">
    <xsl:comment>Do not edit: generated by Transbuild</xsl:comment>
    <head>
      <meta http-equiv="Content-Type"
            content="text/xhtml; charset=UTF-8"/>
      <title>Step 3: <xsl:value-of select="article/title"/></title>
      <style type="text/css">
        <xsl:text>@import url(</xsl:text>
        <xsl:value-of select="TBF:href('/site.css')"/>
        <xsl:text>);</xsl:text>
      </style>

    </head>
    <xsl:apply-templates/>
  </html>
</xsl:template>
```

The result produced by the `TBF:href` function is actually in the form of a relative path from the target file being generated. For example, it will produce a link to '`site.css`' in the '`~target~/index.html`' file; a link to '`../site.css`' in the '`~target~/about/index.html`' file; and a link to '`../../site.css`' in the '`~target~/hardware/omp/index.html`' file. This is very handy, because the links will work when the generated files are browsed directly from the file system. If they were all hard-coded as '`/site.css`', the files will have to be uploaded to a Web server before you could test them.

The pages produced by the build script '`tb-step03.xml`' now uses the CSS stylesheet, and they all have a link to the site's homepage. However, it is still not a usable Web site because there are no other navigation links - that is the subject of the next step.

3.4 Step 4: Children annotations

In the previous step, the rule that applied the XSLT transformation had performed an implicit step. The XSLT processing step needs XML data as its input, so it needed to first

parse the source file into XML before applying the XSLT to it. The explicit processing step to parse raw data into XML is called `xml-load`. So the same rule could be written explicitly as the following:

```
<rule source-suffix=".xml" target-suffix=".html">
  <xml-load annotate-with-source="no"/>
  <xslt stylesheet="scripts/tr04.xsl"/>
</rule>
```

The raw data from the source file goes into the `xml-load` processing step and comes out as XML data. The XML data goes into the `xslt` processing step and (in this case) comes out as XHTML XML. Finally, this XML data is serialized and written into the target file.

As a bit of trivia, if you had a rule with just a single `xml-load` processing step in it, the behaviour will be similar to a straight file copy. The difference being the XML file might not be an exact syntactical copy because of changes caused by the XML parsing and serialization process (e.g. processing entity references, changing character encodings, etc.) And, of course, it will fail to work on non-XML source files.

It should be pointed out that you can have many processing steps in a rule. For example, you could load the XML, transform it with XSLT, transform that result with a different XSLT script, apply yet another XSLT script, and then save the result to the target file. See the reference section for further details. However, one XSLT script is usually sufficient.

The `xml-load` processing step can be used to annotate the parsed XML. The XSLT script can then use those annotations to generate a richer Web page. There are a number of different annotations that can be applied. We'll start by examining the `children` annotation. The annotations are placed inside the `xml-load` element, changing the rule to:

```
<rule source-suffix=".xml" target-suffix=".html">
  <xml-load>
    <children file-name="index.xml"/>
  </xml-load>
  <xslt stylesheet="scripts/tr04.xsl"/>
</rule>
```

The children annotation is useful for creating navigation links to lower subsections of the Web site. It is here where the directory structure of the source tree becomes important. The children annotation starts with the directory where the currently processed source file is in. It looks in all the subdirectories under that directory, and finds the list of files whose name exactly matches the `file-name` attribute. For example, if it was processing the '`~source~/hardware/index.xml`' file, the children files are '`~source/hardware/mp100/index.xml`', '`~source/hardware/mp110/index.xml`', '`~source/hardware/mp120/index.xml`', '`~source/hardware/mp130/index.xml`', and '`~source/hardware/omp/index.xml`'.

The `children` annotation will append an element to the contents of the root element of the parsed source file. This element will have the name `children` and come from the Transbuild annotation namespace of `http://hoylen.com/ns/xmlns/2002/transbuild/annotation` Inside that element, it will place a copy of the root element from parsing those children files. Those root elements will be further annotated with a `source` attribute (from the annotations namespace) indicating which source file it came from. For example, with the

‘~source~/hardware/index.xml’ file, the output from the xml-load processing step will be XML data containing something like:

```
<article
 xmlns:TBA="http://hoylen.com/ns/xmlns/2002/transbuild/annotation"
 TBA:source="transbuild://hardware/index.xml">

  <title>Hardware</title>
  <para>Our products incorporate leading edge technology with award
  winning design which is both beautiful and functional.</para>

  <TBA:children>
    <article TBA:source="transbuild://hardware/mp100/index.xml">
      <title>MP100</title>
      <para>The MP100 improved on the original design.</para>
    </article>

    <article TBA:source="transbuild://hardware/mp110/index.xml">
      <title>MP110</title>
      <para>An upgrade of MP100.</para>
    </article>

    <article TBA:source="transbuild://hardware/mp120/index.xml">
      <title>MP120</title>
      <para>An upgrade of MP110.</para>
    </article>

    <article TBA:source="transbuild://hardware/mp130/index.xml">
      <title>MP130</title>
      <para>The MP130 is a more advance model.</para>
    </article>

    <article TBA:source="transbuild://hardware/omp/index.xml">
      <title>OMP</title>
      <para>The OMP is the first in our product range. A world first
            when it was released.</para>
    </article>
  </TBA:children>
</article>
```

The value of the TBA:source attributes appear in the form of URIs using the "transbuild" scheme. This is the same as a filename of a file in the source tree, and can be used in the same way (e.g. passed to the TBF:href function). If you ever see a URI of this form, treat it a reminder that it is something from the source tree file space.

Notice that the document's root element also has a TBA:source attribute added to it. It's value refers to the currently processed file. If you want to suppress it, add a annotate-with-source attribute to the xml-load element, and set its value to no (as was done in the first example in this step). If not present, the value is yes. You can also suppress the other

`TBA:source` attributes by putting the same attribute in the `children` element. However, you'll probably never do this because the source attribute is very useful.

All `TBA:source` values generated by annotations are in a canonical form. This means you can test if two files are the same by using a string equality test on the `TBA:source` attributes.

If you want to see what annotations are added there are two ways to find out. The first is to remove the `xslt` processing step from the rule, causing the annotated XML to be written straight into the target file. It won't be pretty printed like the above example, so you may need an XML viewer or editor to make sense of it all (a browser like Mozilla 1.2 will view XML files if they have a '`.xml`' extension). The second method is to turn on the debugging trace (see the reference section for details).

In the XSLT stylesheet, the "article" template is modified to use the children annotations to create links to them. The `TBF:href` XPath function is used on the value from the `TBA:source` annotation attribute to create the hyperlink.

```
<xsl:template match="article">
  <body>
    <div class="nav-top">
      <ul>
        <li><a href="{TBF:href('/index.xml')}">Home</a></li>
      </ul>
    </div>

    <h1><xsl:value-of select="title"/></h1>

    <div class="nav-sub">
      <xsl:if test="/article/TBA:children/article">
        <ul>
          <xsl:for-each select="/article/TBA:children/article">
            <li>
              <a href="{TBF:href(@TBA:source)}">
                <xsl:value-of select="title"/>
              </a>
            </li>
          </xsl:for-each>
        </ul>
      </xsl:if>
      <xsl:text disable-output-escaping="yes">&amp;nbsp;</xsl:text>
    </div>

    <div class="main">
      <xsl:apply-templates/>
    </div>
  </body>
</xsl:template>
```

The `xsl:text` containing a single non-breaking space is a work-around so that the page will render properly when there are no children. (Is this a browser bug? Is there a better solution?)

Since the parsed XML data now has the extra annotations, we don't want them to be processed by `xsl:apply-templates`. An extra template is added to make sure that the XSLT processor will ignore them.

```
<xsl:template match="TBA:*">
  <!-- ignore annotation elements -->
</xsl:template>
```

This step can be tested by running Transbuild on the build script file '`tb-step04.xml`'.

3.5 Step 5: Order of annotations

The root elements in annotations is ordered according to the lexicographical order of the file or directory names. If we want to process them in a specific order, one approach could be to name the files and directories following some naming scheme. However, this usually leads to ugly names (such as '`01mercury`', '`02venus`', '`03earth`', '`04mars`'). It is also difficult to maintain (for example adding '`asteroid-belt`' between earth and mars might need the existing files to be renamed. The sorting may also depend on the locale.

The most reliable way of ensuring a certain order is to perform the sorting inside the XSLT script. Data representing the sorting criteria may need to be added to the source files.

In this tutorial, the `status` attribute in the source files will be used as the sorting value. This will be a number that indicates the file's relative order amongst its siblings. We add a `xsl:sort` element to the loop to ensure that the children appear in the correct order.

```
<div class="nav-sub">
  <xsl:if test="/article/TBA:children/article">
    <xsl:for-each select="/article/TBA:children/article">
      <xsl:sort select="@status" data-type="number"
                order="ascending"/>
              <li>
        ...
```

Instead of writing the select value as `/article/TBA:children/article`, you might want to write it as `/*/TBA:children/*` - this way, it will work independently of the XML vocabulary of the source files, making it easier to reuse this XSLT code.

This step can be tested by running Transbuild on the build script file '`tb-step05.xml`'. In the nest step, some of the other annotations will be used.

3.6 Step 6: More annotations

The `ancestors` annotation searches each of the parent directories of the currently processed file. It is useful for creating a 'you-are-here' or 'breadcrumbs' style navigation links. Unlike other annotations, the ordering does not depend on the file or directory name - the order is the directory hierarchy from top to bottom. The first file in the annotation will always be the one for the root of the source tree. The last file will be the one for the directory containing the currently processed file. If the currently processed file happens to also match the `file-name`, it will also appear in the annotation.

The `members` annotation picks up other files in the current directory. However, we won't be using it in this tutorial. (A hint to remember: all these annotations have names which are plurals - e.g. "members", not "member".)

Design the layout of the source tree directories with annotations in mind so that they can be used to generate the site navigation links. The `children` annotation picks up files in immediate subdirectories. The `ancestors` annotation picks up files in the parent directories. The `members` annotation picks up files in the current directory.

If a directory doesn't have a file with a matching name, that directory is just skipped. That might be useful if you want to have a directory that is not reflected in the Web site's navigation hierarchy.

So far, annotations have been calculated relative to the directory containing the currently processed source file - that can be changed using the `dir` annotation. It has a mandatory `path` attribute which specifies the new working directory. This path can be relative or absolute, and may optionally be prefixed by `transbuild://` as a reminder that it is in the source tree file space.

Annotations listed inside the `dir` element are processed relative to that new directory. In the output, a `dir` element is created and the annotations in the `dir` are nested inside it.

You can add multiple annotations in one `xml-load`. The annotation elements will be appended in the same order as supplied. For example, we'll change the rule to:

```
<rule source-suffix=".xml" target-suffix=".html">
  <xml-load>
    <dir path="/">
      <children file-name="index.xml"/>
    </dir>
    <ancestors file-name="index.xml"/>
    <children file-name="index.xml"/>
  </xml-load>
  <xslt stylesheet="scripts/tr06.xsl"/>
</rule>
```

This will parse the source file and annotate it to create a large body of XML data containing something like:

```
<article
 xmlns:TBA="http://hoylen.com/ns/xmlns/2002/transbuild/annotation"
 TBA:source="transbuild://hardware/index.xml">

  <title>Hardware</title>
  <para>Our products incorporate leading edge technology with award
  winning design which is both beautiful and functional.</para>

  <TBA:dir TBA:source="transbuild://">
    <TBA:children>
      <article
       TBA:source="transbuild://about/index.xml">...</article>
      <article
       TBA:source="transbuild://contact/index.xml">...</article>
      <article
```

```
        TBA:source="transbuild://hardware/index.xml">...</article>
      <article
       TBA:source="transbuild://software/index.xml">...</article>
    </TBA:children>
  </TBA:dir>

  <TBA:ancestors>
    <article TBA:source="transbuild://index.xml">...</article>
    <article TBA:source="transbuild://hardware/index.xml">...</article>
  </TBA:ancestors>

  <TBA:children>
    <article
     TBA:source="transbuild://hardware/mp100/index.xml">...</article>
    <article
     TBA:source="transbuild://hardware/mp120/index.xml">...</article>
    <article
     TBA:source="transbuild://hardware/mp110/index.xml">...</article>
    <article
     TBA:source="transbuild://hardware/mp130/index.xml">...</article>
    <article
     TBA:source="transbuild://hardware/omp/index.xml">...</article>
  </TBA:children>

</article>
```

Notice that the ordering of the imported root elements is sorted in lexicographical order, except for the `ancestors` annotation. In this example, the data from the '`~source~/hardware/index.xml`' file appears three times (as the document itself, in the children of the directory annotation, and in the last ancestors annotation). This may seem very inefficient, but don't worry about it - most of the time your files are small and Transbuild creates these annotations very efficiently by caching the parsed XML data.

The XSLT script is modified to use the information from the annotations. The top level children are used to create navigation links into the different major sections of the site. The ancestor annotations will be used to create a "bread-crumbs" navigation showing all the levels above the current page.

```
<xsl:template match="article">
  <body>
    <div class="nav-top">
      <ul>
        <li><a href="{TBF:href('/index.xml')}">Home</a></li>
        <xsl:for-each select="/article/TBA:dir/TBA:children/article">
          <xsl:sort select="@status" data-type="number"
                    order="ascending"/>
          <li>
            <a href="{TBF:href(@TBA:source)}">
              <xsl:value-of select="title"/>
            </a>
          </li>
```

```
      </xsl:for-each>
    </ul>
  </div>

  <h1><xsl:value-of select="title"/></h1>

  <div class="nav-hier">
    <xsl:for-each select="/article/TBA:ancestors/article">
      <xsl:if test="position() != 1">
        <xsl:text> &gt; </xsl:text>
      </xsl:if>
      <a href="{TBF:href(@TBA:source)}">
        <xsl:value-of select="title"/>
      </a>
    </xsl:for-each>
    <xsl:text disable-output-escaping="yes">&amp;nbsp;</xsl:text>
  </div>

  ...
```

This step can be tested by running Transbuild on the build script file '`tb-step06.xml`'. It produces a set of Web pages with a set of navigation links.

The usefulness of Transbuild is now apparent. By using Transbuild annotations, the navigation links are all generated automatically without any hand editing of the source files. By carefully writing flexible XSLT scripts, you can future-proof your site. For example, if we wanted to add a new section to the site (e.g. for support), all you have to do is create a directory '`~source~/support`' and a single file '`~source~/support/index.xml`' and re-run Transbuild. A new set of Web pages will be generated, with every page having a top level link to that new section. It's that simple - try it! Or, try adding a new hardware product by creating a single directory and one file.

3.7 Step 7: More sophistication through XSLT

You've now seen the main features of Transbuild: the processing model to generate the target tree, the file renaming mechanism, the annotations and how to make use of them.

In this step, we will enhance the XSLT script to detect if the current page is a section page and disable that link in the section navigation bar. This is done by comparing the `TBA:source` attribute of the annotation file to that of the current file. When they are different a link is generated, when they are the same a link is not generated.

```
<xsl:template match="article">
  <body>
    <div class="nav-top">
      <ul>
        <li><a href="{TBF:href('/index.xml')}">Home</a></li>

        <xsl:for-each select="/article/TBA:dir/TBA:children/article">
          <xsl:sort select="@status" data-type="number"
                    order="ascending"/>
```

```
            <li>

              <xsl:choose>
                <xsl:when test="@TBA:source = /*/@TBA:source">
                  <span class="current">
                    <xsl:value-of select="title"/>
                  </span>
                </xsl:when>

                <xsl:otherwise>
                  <a href="{TBF:href(@TBA:source)}">
                    <xsl:value-of select="title"/>
                  </a>
                </xsl:otherwise>
              </xsl:choose>

            </li>
          </xsl:for-each>
        </ul>
      </div>
      ...
```

We'll also hide the 'bread-crumb' navigation link on the home page. The non-breaking space is needed so the spacing for the 'bread-crumbs' navigation appears even though there are no links in it.

```
...
<div class="nav-hier">
  <xsl:if test="/*/@TBA:source != /*/TBA:ancestors/*[1]/@TBA:source">
    <xsl:for-each select="/article/TBA:ancestors/article">
      <xsl:if test="position() != 1">
        <xsl:text> &gt; </xsl:text>
      </xsl:if>
      <a href="{TBF:href(@TBA:source)}">
        <xsl:value-of select="title"/>
      </a>
    </xsl:for-each>
  </xsl:if>
  <xsl:text disable-output-escaping="yes">&amp;nbsp;</xsl:text>
</div>
...
```

This step can be tested with the '`tb-step07.xml`' build script.

A large part of the power of Transbuild comes from the inherent power and flexibility of XSLT: become an expert in using XSLT and you'll become an expert in using Transbuild!

3.8 Step 8: Timestamps

Transbuild also provides an extension XPath function to create timestamps. The function is called `timestamp`.

It can be used to output times in a user defined format. It can generate several different times: the last modified time of the source file, the time Transbuild is run, or the time from text in the source file. It can also convert the time into any desired timezone. However, in this tutorial the simple form will be used to display the last modified time of the source file, in the default format, and in the default UTC timezone. The other options are documented in the reference section.

```
    ...
    <div class="main">
      <xsl:apply-templates/>
      <p class="timestamp">
        Last updated: <xsl:value-of select="TBF:timestamp()"/>
      </p>
    </div>
  </body>
```

If you define your own time format, it is best to stick to the ISO 8601 standard or the W3C Date Time Format profile of it.

This step can be tested with the '`tb-step08.xml`' build script.

3.9 Step 9: Processing different files differently

So far we have been transforming just one type of file. In this step we will add other types of files into the source tree.

For this step, extra files have been added to the source tree. You will find a new source tree under the directory '`source-b`'. It extends the previous source tree by adding a few more XML files and subdirectories, adding data-sheet XML files for the hardware products, and a timeline XML data file. These contain XML data using different XML vocabularies from the one we have been using so far. Also, some JPEG and PNG images have been added.

These different XML files need to be processed using different XSLT scripts. We have two main options:

Using the existing build script, and expand the single XSLT stylesheet into one big stylesheet that detects and processes each vocabulary appropriately.

Create separate XSLT scripts to process each different XML vocabulary, and add extra rules to the build script.

In most cases, you'll choose the second option. With the second option, there must be a way to ensure that the correct rule is used according to the source file's XML vocabulary. Rules are picked according to the filename suffix, and the order they appear in the build script. Since we have already decided that all XML files will have a '`.xml`' extension, we will need longer extensions based on that. This is possible because the `source-suffix` value does not care about file-system extensions (the '`.`' has no special significance in it).

The filename convention this tutorial will use the suffix `-hdat.xml` for hardware data-sheets, and the suffix `timeline.xml` for files containing the timeline XML vocabulary. All other files (i.e. the '`index.xml`' files) will remain unchanged. It just happens that there is only one timeline vocabulary file in the whole site, and its entire name is '`timeline.xml`' (Transbuild does not care if the suffix matches the entire filename or just part of it.)

The new build script now looks like this:

```
<?xml version="1.0"?>

<build-script
 xmlns="http://hoylen.com/ns/xmlns/2002/transbuild/buildscript"
 version="1.0"
 source="source-a"
 target="target">

<rule source-suffix="-hdat.xml" target-suffix=".html">
  <xml-load>
    <dir path="transbuild://">
      <children file-name="index.xml"/>
    </dir>
    <ancestors file-name="index.xml"/>
    <children file-name="index.xml"/>
  </xml-load>
  <xslt stylesheet="scripts/tr09-h.xsl"/>
</rule>

<rule source-suffix="timeline.xml" target-suffix=".svg">
  <xslt stylesheet="scripts/tr09-t.xsl"/>
</rule>

<rule source-suffix=".xml" target-suffix=".html">
  <xml-load>
    <dir path="transbuild://">
      <children file-name="index.xml"/>
    </dir>
    <ancestors file-name="index.xml"/>
    <children file-name="index.xml"/>
  </xml-load>
  <xslt stylesheet="scripts/tr09-a.xsl"/>
</rule>

<rule source-suffix=".png"><file-copy/></rule>
<rule source-suffix=".jpg"><file-copy/></rule>
<rule source-suffix=".css"><file-copy/></rule>

<rule source-suffix="~"/>

</build-script>
```

Notice that we are still using the old rule that matches `.xml` suffixes. It is placed after the other rules with longer suffixes so they will match first if possible. File copy rules have been added for the two new image file types.

The XSLT stylesheet to process the hardware fact-sheets is similar to the one we have already created to process the '`index.xml`' files. The difference is in the input XML it processes.

For something different, the '`timeline.xml`' will be processed by a XSLT stylesheet to generate a SVG file. The Scalable Vector Graphics (SVG) format is an XML vocabulary for vector graphics. Don't worry if you don't know how to write SVG. However, you will need a browser plug-in to display it (such as the one from Adobe.)

The XSLT processor can transform XML data into different formats. We have been using XHTML, and now SVG - both XML vocabularies. Remember, XSLT can also generate normal HTML and arbitrary text files too. And if you need some other form of processing, there are other processing steps available besides XSLT (see the reference section for details).

We want to embed the SVG file and other graphic images into the XHTML pages. So, we'll extend the source XML vocabulary for the '`index.xml`' files to:

```
<!ELEMENT article (title, (para|imagedata)*)>
<!ATTLIST article status CDATA #IMPLIED>
<!ELEMENT title (#PCDATA)>
<!ELEMENT para (#PCDATA | ulink)>
<!ELEMENT ulink (#PCDATA)>
<!ATTLIST ulink url CDATA #REQUIRED>
<!ELEMENT image (#PCDATA)>
<!ATTLIST image
          fileref CDATA #REQUIRED
          format (PNG|JPEG|SVG) #REQUIRED
          width CDATA #IMPLIED
          height CDATA #IMPLIED>
```

The XSLT stylesheet has been modified to handle the image element. Some of the XML source files use the `image` element to reference image files.

The stylesheet has also been modified so that arbitrary external hyperlinks (using HTTP and FTP protocols) can be created as well as links to internal pages.

```
<xsl:template match="ulink">
  <xsl:choose>
    <xsl:when test="starts-with(@url, 'http:')">
      <a href="{@url}"><xsl:apply-templates/></a>
    </xsl:when>
    <xsl:when test="starts-with(@url, 'ftp:')">
      <a href="{@url}"><xsl:apply-templates/></a>
    </xsl:when>
    <xsl:otherwise>
      <a href="{TBF:href(@url)}"><xsl:apply-templates/></a>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>
```

This step has shown how different types of files can be distinguished by careful use of filename suffixes. It can be tested with the '`tb-step09.xml`' build script.

3.10 Step 10: XSLT parameters

Parameters can be passed into the XSLT scripts. This is an XSLT feature, and we won't be explaining how to write XSLT stylesheets that use parameters here. All that we'll be explaining is how to supply those parameters to the scripts.

Parameters are supplied using the `param` element in the contents of the `xslt` element. This is an empty element that specifies the name and value of the parameter as attributes. The name is defined by the `name` attribute. The value is defined by either the `value-xpath` or `value-string` attribute.

The parameter values are XPath expressions. In simple scripts, you will probably be passing in a literal string value. However, if you forget to quote the string properly, the processor treats it as an XPath expression which does not match any nodes and appears as a blank string where it is used - this is very confusing for beginners and experts alike. For this reason, there are these two ways of supplying the value: `value-string` which automatically quotes the string value, and `value-xpath` whose name is a reminder that it expects an XPath expression.

The following rule illustrates two parameters being passed into the XSLT stylesheet.

```
<rule source-suffix=".xml" target-suffix=".html">
  <xml-load>
    <dir path="transbuild://">
      <children file-name="index.xml"/>
    </dir>
    <ancestors file-name="index.xml"/>
    <children file-name="index.xml"/>
  </xml-load>
  <xslt stylesheet="scripts/tr10-a.xsl">
    <param name="cssfile" value-string="/site2.css"/>
    <param name="titletext" value-xpath="/article/title"/>
  </xslt>
</rule>
```

Since the `value-string` is just a convenience mechanism, you could get the same results by using `value-xpath` with a quoted value.

```
...
  <param name="cssfile" value-xpath="'/site2.css'"/>
...
```

Two alternative stylesheets have been provided for you to experiment with: '`site1.css`' and '`site2.css`'.

Since these Web pages start with many navigation links, we've also added a "skip navigation links" hyperlink at the very beginning of the page. This makes it more accessible to people using voice browsers or who are restricted in how they navigate through a page.

Finally, you should always always check that your site is standards compliant: using valid HTML markup and valid CSS. Check that it is accessible and usable, test it with a number of different Web browsers, and test it on a variety of different platforms.

This is the end of the tutorial. If you want to learn more, have a look at the other examples in the test directory and read the reference section.

4 Using Transbuild

This section contains some general tips and tricks about using Transbuild.

4.1 Common mistakes

Use the correct namespace in XPath expressions. One of the commonest mistakes is forgetting to specify the correct namespace for annotation elements and attributes.

```
<xsl:value-of select="/@source"/>      <!-- wrong -->
<xsl:value-of select="/@TBA:source"/>  <!-- correct -->
```

Root elements from annotation files are loaded as the contents of the annotation element. It is always one extra level down. For example, to process all the children annotation files:

```
<xsl:for-each select="/*/TBA:children">    <!-- wrong -->
<xsl:for-each select="/*/TBA:children/*">  <!-- correct -->
```

4.2 Dynamic Web sites

Although Transbuild only generates static files, it does not mean your Web site cannot be dynamic. You could use Transbuild in conjuction with other Web technologies. For example, use it to generate Server Side Include (SSI) pages or Java Server Pages (JSP) files.

You could even use Transbuild to generate CGI scripts. For example, have XSLT scripts which output Perl code. This is not as crazy as it sounds, because one advantage would be consistency between the static and dynamic parts of the site. The HTML result pages from the CGI scripts can be made consistent with the rest of the site.

4.3 XML and XSLT extensions

Transbuild is implemented using libXML and libXSLT. Those implementations support features which can be used in Transbuild. For example, XInclude can be used inside the build script and source XML files. Various extension elements and XPath functions are also available in the XSLT processor.

See the libXML and libXSLT documents at `http://www.xmlsoft.org/` for details.

4.4 Version control with CVS

Transbuild can be used in conjunction with Concurrent Versions System (CVS) to manage changes to a Web site. The source tree can be checked into CVS. However, make sure that your CVS has been configured to handle binary files (like images and PDF documents) properly.

The `timezone` extension XPath function can use a time formatted by the CVS `Date` keyword. In the source file, include the keyword:

```
<article>
  <metadata>
    <checkin>$Date: 2003/04/03 10:26:44 $<checkin>
```

```
  </metadata>
...
```

When the file has been checked in, the date will be updated:

```
<article>
  <metadata>
    <checkin>$Date: 2003/04/03 10:26:44 $<checkin>
  </metadata>
...
```

And it can be extracted and formatted in the XSLT stylesheet:

```
<p>
  <xsl:text>Last modified: </xsl:text>
  <xsl:value-of select="TBF:timestamp('yyyy-MM-dd',
                                      /article/metadata/checkin)"/>
</p>
```

Note that directories in the source tree called '`CVS`' are ignored by Transbuild.

4.5 Rule to match all files

If you want a rule to match all files in the source tree, make the `source-suffex` an empty string. This rule would have to be placed at the end of the build script, otherwise none of the other rules would be reached.

Do not be tempted to use this as a way of disabling the error that occurs when no rule matches a file. That error is a safety net to detect unexpected files in the source tree.

5 Reference

5.1 Running Transbuild

Transbuild processes a source tree to generate a target tree. In this document, a "tree" is a single directory and all the files and subdirectories nested under it. Directories are replicated under the target tree with the same names as it appears in the source tree. Source files are processed by rules and the results saved in files in the corresponding target directory, with a new name as defined by the rule.

The processing rules are defined in a build script file. The locations of the source tree and target tree may also be defined in the build script file.

All files and directories under the source tree are processed. There is one exception: directories called '`CVS`' (and content under them) are totally ignored.

Transbuild runs in two phases. In the first phase, it finds all the files in the source tree, determines which rule should be used for all of them, and examines the target tree to see which files are out-of-date and need to be rebuilt. In the second phase, missing directories are created in the target tree and then the files that need building are processed.

5.1.1 Command line options

Transbuild is a program run from the command line. It recognises a number of options.

Command line options usually have a long form and a single character form following the GNU conventions (e.g. '`--verbose`' and '`-v`'). For a summary list of available options, use '`--help`' (or '`-h`'). On some platforms, only the short form is available.

The options are:

```
-h, --help               prints a short
-f, --file filename      read file as the build script
-a, --all                force all files to be
-k, --keep-going         keep going when source files match no rules
-n, --dry-run            don't build files
-i, --ignore-errors      ignore errors in file builds
-d, --debug flags        print debug information
-s, --silent, --quiet    don't print progress
-v, --verbose            verbose
-T, --target dir         target tree to use
-S, --source dir         source tree to use
-o, --options spec       set options
-V, --version            show version information
```

–help, -h

This option displays a short summary of the available options and major environment variables used.

–file, -f

Specifies the filename of the build script. If this option is not present, the program will try to look for a file called '`Transbuild.xml`' from the current directory.

–all, -a

Force all files to be rebuild. By default only files which need to be updated are processed. It does not process a source file if its target file already exists and its modified time is after that of the source file.

The dependency checking very simplistic, assuming that a target file only depends on its one source file. It does not take into account annotations or any usage of the XPath `document`. The '`--all`' option is useful for rebuilding everything to ensure that all changes in the source tree have been incorporated.

Transbuild does not delete old files from the target directory. If source files are renamed or the rules for naming target files have been changed, there may be old files left lying around in the target tree. For this reason, it may be necessary to occasionally delete the entire target directory and completely regenerate it.

–keep-going, -k

Force Transbuild to ignore errors detected in the source tree during the first phase of processing. Normally, it is an error if a file in the source tree does not have a matching rule in the build script. This option causes Transbuild to ignores those files and not to generate an error.

This option controls error handling in the first processing phase. To control errors in the second phase, use the '`--ignore-errors`' option.

–dry-run, -n

Finds the rules for the source files and determines which ones need to be built, but do not perform the build. Only the first phase processing is performed. No rules are executed, and the target tree is left unchanged.

–ignore-errors, -i

Normally, when an error occurs during an execution of a rule the program stops running. This option indicates that it should not stop, but keep processing the other files.

This option controls error handling in the second processing phase. To control errors in the first phase, use the '`--keep-going`' option.

–debug, -d

Print out debug trace information. This option takes a comma separated list of flags to determine which debugging information to show. The available flags are:

'`script`' Prints information from the parsed build script.

‘`tree`’ Displays the source tree files and directories. An asterisk indicates that the target file is out-of-date or missing and needs to be rebuilt.

‘`tree2`’ Shows extra details about the source tree: the last modified time of the source and the full path names of the source file and target file. This flag automatically enables the ‘`tree`’ flag as well.

‘`xml-load`’
Shows the annotations made to each XML file, and the files used in the annotations.

‘`cache`’ Prints out XML cache statistics at the end of the run.

‘`all`’ Enables all the above debug information.

If an unknown flag is supplied, an error will occur and a short summary of the available flags printed.

The debug information is printed to stderr.

–silent, –quiet, -s

Do not print out details of the build process. If this option is not specified, the first phase processing will print out information about how many files need to be rebuilt and the second phase processing will print out a message for each directory and file built.

–verbose, -v

Print out verbose messages while running.

Currently, this option has no effect because there are no verbose messages.

–target, -T

Specifies the target directory. This value will override any target directory specified in the build script.

–source, -S

Specifies the source directory. This value will override any source directory specified in the build script.

–options, -o

Used to specify configuration options. The argument is a comma separated list of name-value pairs (with the name and value separated by an equal sign).

‘`charset`’ Specifies the operating system character set and encoding to use.

‘`cachesize`’
Size of the XML cache, as a non-negative integer.

‘`validate`’
Validate against DTDs if a DTD has been specified in the source XML files. Value must either be “yes” (the default) or “no.” See the `SGML_CATALOG_FILES` environment variable for specifying where the DTDs can be found.

–version, -V

Prints the program name, version number and copyright message.

5.1.2 Environment variables

Some special environment variables are used by Transbuild. Other standard environment variables (e.g. those for the timezone and locale) are also used, but are not described here.

SGML_CATALOG_FILES

The `SGML_CATALOG_FILES` environment variable specifies a colon separated list of catalog files. These are Open SGML entity catalog files which describe the mapping from external references in DTDs to local files. One use is to allows the XML validation to be performed using local copies of the DTDs rather than using the ones specified in the system ID.

An example is shown below. Here it is mapping public identifiers into local files. When these identifiers are encountered in DOCTYPE declarations, these local files are used.

```
PUBLIC "-//W3C//DTD XMLSCHEMA 200102//EN" "/usr/local/w3c/xsd.dtd"
PUBLIC "-//W3C//DTD SVG 1.0//EN" "/usr/local/w3c/svg10.dtd"
PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "/usr/local/w3c/xhtml1-s.dtd"
```

TMPDIR

The `TMPDIR` environment variable specifies the directory in which temporary files are created. If not specified, '`/tmp`' is used.

5.1.3 Character set

This section describes character set and encoding issues relating to Transbuild.

A character set is a mapping which defines that a certain number represents a certain character. There are many different character sets used in the world today. Knowing which character set is being used is necessary to know which character a number it represents. The codes for ASCII is common between most character sets. However, it only defines characters up to 127, and does not have the characters necessary for many non-English languages. Numbers above 127 often represent different characters in different character sets.

A character encoding determines how that number is represented or stored in the computer. Nearly all modern computers use a 8-bit byte as its basic unit of storage, so characters sets with numbers from 0-255 are encoded as one byte per character. With larger character sets, more numbers are required beyond 255 and there are several different ways to encode those numbers in one or more bytes. Unicode is a character set with more than 255 characters.

Transbuild operates in two character set environments: the XML environment and the operating system environment. The XML format uses Unicode characters, although it can use a number of different encodings. Thankfully, the XML parser in Transbuild handles these transparently and converts them internally into Unicode.

The character set used by the operating system environment differs between operating systems and locales. The character set and encoding affects many things: it affects how file and directory names are interpreted, and what output is printed to the display.

Transbuild will run without any character set or encoding problems until information is passed between the two environments. For example, this occurs when a filename needs to be placed into XML (e.g. when the annotation source attribute is being created), or when the name of an XML element needs to be printed out (e.g. in an error message). Some of these translations must be exact (e.g. with the filename), while others can be an approximate representation (e.g. with the error message).

Firstly, Transbuild needs to know what character set and encoding the operating system environment is using. When it starts up, it determines this by examining the locale. If it cannot determine what it is, an error will be raised. You can explicitly indicate what character set and encoding is being used with the '`--option`' option. For example,

```
$ transbuild --option charset=iso-8859-1
```

The first difficulty is knowing what values are recognised. There is no standard set of names (on some systems it might accept `iso-8859-1`, on others `8859-1`). Transbuild uses the iconv library for character set translation and encoding. In newer versions, run `iconv` command with a '`-l`' option to list the available encodings. In older versions, looking at the manual page for `iconv` and `iconv_open` might provide clues to encoding names.

Secondly, Transbuild needs to have a mechanism to of translate from one character set and encoding to another. Sometimes iconv might not have implemented the algorithm. At other times, such translations are impossible to do without loosing information (e.g. translating Chinese characters into the ASCII character set). Transbuild will raise an encoding error if it encounters these problem. You will need to remove, rename or change the offending text.

A more subtle problem is when the wrong character set or encoding has been used. An error might not be generated, but the information will be silently processed incorrectly. Make sure you specify the correct character set if you are setting it.

There is no easy solution when dealing with mixed character sets and encodings. One approach would be to restrict yourself to ASCII characters in both XML documents and for filenames. Alternatively, some modern operating system supports Unicode natively (a good guilde to this is the UTF-8 and Unicode FAQ for Linux/Unix.

5.1.4 XML Cache

Transbuild internally caches XML files parsed for annotations. This can speed up processing if the XML files are reused many times. For example, a top level XML file might be used as an annotation in every file to generate a common set of navigation links.

Normally, you do not have to worry about the cache. However, if you are running low on memory, or have an unusually large number of reused annotations, it might be necessary to adjust the cache size. This can be done by setting the '`cachesize`' with the '`--option`' option from the command line. You can monitor the efficiency of the cache with '`--debug cache`'.

5.2 Build script

The build script is the configuration file which controls the operation of Transbuild. It defines the processing rules, and can also specify the location of the source and target trees.

For precise details about the build script format, see the annotated DTD at the end of this document.

A build script consists of a list of rules. When a source file is processed, the rules are searched from the beginning of the build script until a matching rule is found. The match is performed on matching the suffix of the filename. The target file that will be generated will be in the corresponding subdirectory in the target tree, with a filename that is formed by removing the matching suffix string and appending a target suffix string. An error is raised if a file does not match any rule in the build script.

5.2.1 Processing rules

Inside each rule, there are zero or more processing steps. Data is passed from one processing step to the next. The input to the first processing step is the raw data from the source file. The output from the last processing step is saved into the target file. The only exception is if there are no processing steps the output file is not created.

There are two types of data passed between processing steps: raw data and XML data. Raw data is arbitrary data that is stored in a file (it may be a text file, a binary file or even serialized XML). The XML data is parsed XML data (think of it as a DOM tree in memory).

You can arbitrarily mix processing steps, because they can convert the input data into the required format. For example, `xslt` will parse raw data input into XML before applying the XSLT stylesheet, or `system` will serialize XML input before running the command on it. The only combination that is not permitted is two `xml-load` steps in a row (once in XML form, it makes no sense to reload it). Also, any conversion from raw data to XML data will only work if the raw data is proper XML.

There are four processing steps. They can be used in any order, and repeated any number of times, in a rule (subject to any noted limitations).

5.2.1.1 xml-load

The `xml-load` processing step converts raw input data into XML data. It cannot accept XML data as input.

A significant feature of `xml-load` is that it can add extra information to the parsed XML from other files. These are known as annotations inside the `xml-load` element. Annotations can be nested in other annotations.

Each annotation adds an element to the XML data from the annotation namespace of `http://hoylen.com/ns/xmlns/2002/transbuild/annotation`. Inside that element, the matching annotation files are added.

The members annotation adds files from the current working directory whose name matches a given suffix. If the current source file's name matches, it too will be included in the annotation. They are ordered according to lexical order of the filenames.

The children annotation adds files of a given name from the immediate subdirectories under the current working directory. They are ordered according to the lexical order of the directory names.

The ancestors annotation adds all the files matching a given name from the list of directories above and including the current working directory. The are added in order, from the root of the source tree down to the current working directory.

The set of annotations immediately under the `xml-load` element uses the directory containing the source file being processed as the current working directory. Annotations nested inside other annotations will use the directory containing the file being annotated as the working directory.

The current working directory can be changed using a `dir` annotation element. This operates like a change directory command, and it can be passed a directory relative to the current directory or an absolute directory name based from the root of the source tree.

By default, annotation files will be parsed as XML and appended to the top XML element of the file being annotated. Only the contents of the root element is added (comments and processing instructions not inside the root element are ignored). This embedding behaviour can be disabled, in which case it will simply add an empty element called `file` from the Transbuild annotation namespace. This is useful for annotations that obtain a list of non-XML files (e.g. image files).

By default, annotation `file` elements and added root elements have a `source` attribute to identify the source file. This attribute is from the Transbuild annotation namespace. It can be omitted by setting `annotate-with-source` to "no" it in the annotation.

All annotation elements can have an `name` attribute. This value will be used to create a `name` in the annotation element in the parsed data. The `name` attribute is from the Transbuild annotations namespace. You can use it as a label to distinguish between the different annotations.

See the tutorial and demonstrations for more discussions and examples of annotations.

5.2.1.2 xslt

The `xslt` processing step applies an XSLT transformation to the input data, and produces XML data output. It accepts both types of input data (automatically parsing raw input data as XML before using it).

Parameters can be passed to the XSLT script using the `param` element. Parameters can either be literal string values or XPath expressions.

5.2.1.3 system

The `system` processing step applies a system command to the input data, and produces raw data output. It accepts both types of input data (automatically serializing XML input data into a text stream).

The command must read its input from stdin and output its data to stdout. If the command does not terminate with a status of zero, Transbuild will exit with an error.

The command is executed from the directory where the build script resides.

5.2.1.4 file-copy

The `file-copy` processing step indicates that the input data should be copied to the target file. It used when the source file is to be copied to the target file with no processing performed on it (by creating a rule with a single `file-copy` processing step in it.)

5.3 XPath functions

5.3.1 exists

Usage: exists(pathname)

This function returns true if the source file or directory exists, false if it does not. The file or directory name can be relative to the current source file or absolute to the source tree root. It makes no difference if the name has a `transbuild://` scheme in front of it.

```
<xsl:if test="TBF:exists('background.png')">
  ...
</xsl:if>
```

5.3.2 href

Usage: href(pathname [, base])

To create hyperlinks to target files, you need to know the name and location of the target files. Transbuild provides this XPath extension function to convert the names (such as those obtained from the `TBA:source` attribute) into a target file name.

Relative paths are always generated (relative to the current target file). This allows your generated HTML files to be tested before putting onto a Web server. They can be tested by opening generated files directly in a browser. If absolute paths were generated those links would not work, because they would be absolute paths in the Web server, but not in the file system.

In general, you should try to only refer to files and directories in the source tree file space, and use the `TBF:href` function to map them into the target tree file space. If source '`.xml`' files are being transformed into target '`.html`' files, always refer to the `".xml"` files and never to the '`.html`' files. Work in the source tree file space, and let Transbuild worry about the target tree file space.

In addition to Transbuild URIs, the `TBF:href` XPath function also operates on absolute paths (absolute to the source tree root directory) and relative paths from the currently processed source file.

The `transbuild://` scheme used in annotations does not serve any real function, other than alerting anyone seeing it that they are working in the source tree file space. For example, there is no difference between `transbuild://abc/def.xml` and `/abc/def.xml`.

An error will be raised if `TBF:href` tries to reference a file or directory which does not exist or is outside of the source tree.

The two argument form of `TBF:href` takes a base file or directory as the second argument. The resulting path will be relative to that base. The base (like the first argument) must be

a file or directory which exists in the source tree, otherwise an error will be raised. The one argument form simply uses the current source file as the base.

The two argument form is only useful in advanced scripts where you are generating extra output files. And then, only if you are creating those output files in a different directory from where the current target file will be placed.

<emphasis>Note: in the current implementation, the base must be a source source file (and not a directory) for it to work properly. Except when the pathname is to a file in the same directory as the base, in which case the base must be a directory. Yes, this is inconsistent, confusing and hopefully will be fixed in future releases. However, this is a low priority since I don't expect many people will be creating files in such a complex manner. It is much simpler putting the extra output files in the same directory as the current file.</emphasis>

5.3.3 size

Usage: size(pathname [, unit])

This function returns the size of a source file. It does not work with directories. This function might be useful for generating text in Web pages to indicate to the reader how big a download file is.

If no unit is supplied, the function will automatically which units to show the file size. A string will be returned containing the file size with appropriate precision and the units (e.g. 10B, 12kB, 3.1kB, 1.4MB).

If a unit is supplied, it is used and a integer value is returned based on the rounded size in that unit. If the rounded size is too small, the value of 1 will be returned (unless the size is exactly zero bytes). Zero is only returned if the file is empty. The valid units are: `B` for bytes, `kB` for kilobytes, and `MB` for megabytes.

5.3.4 sourcepath

Usage: sourcepath(pathname)

Converts a pathname in the source tree file space into the file system's real pathname (relative to the processor's working directory.) This function was created for use with the `document` XPath function, allowing you to open other files in the source tree.

Consider this scenario: the source tree is in '`/home/citizen/projects/web`', and we are processing the file '`/home/citizen/projects/web/index.xml`' and want to load extra data from '`/home/citizen/projects/web/extra.xml`'. Simply running `document("extra.xml")` would not work because the program's current working directory is not '`web`'. The correct approach is `document(TBF:sourcepath('extra.xml'))`. If we are running Transbuild from the directory '`/home/citizen`', the value of `TBF:sourcepath('extra.xml')` will be `projects/web/extra.xml`, which is what the `document` function needs to open the file.

5.3.5 targetpath

Usage: targetpath(pathname)

Converts a pathname in the source tree file space into the file system's real pathname for its corresponding target (relative to the program's working directory.)

This function is useful if you want to create files in additional to the default target file. You will need to use non-standard extensions to do this (such as with the Saxon `output` element). Those extensions need a real filename to create. This function gives you a real directory or filename in the target tree to use.

Consider the scenario described in the previous section where the target tree is in '`/home/citizen/projects/output`'. The '`index.xml`' source file creates the target file '`/home/citizen/projects/output/index.html`'. The way to create an extra output file in '`/home/citizen/projects/output/more.html`' is:

```
<saxon:output file="{TBF:targetpath('.')/more.html}" method="xml">
  <html>
    ...
  </html>
</saxon:output>
```

The expression `TBF:targetpath('.')` will produce the pathname `projects/output`, since the transbuild program was run from the '`/home/citizen`' directory. So the value of the file attribute will be `projects/output/more.html`.

Things start getting very complicated when you move away from the model of one source file creates one target file!

5.3.6 timestamp

Usage: timestamp(format?, timespec?, timezone?)

The `timestamp` XPath function calculates and formats a time. It takes three optional arguments:

The first argument is a pattern string. If omitted, a default format pattern is used.

The second argument determines the time to use. If used, the first argument is mandatory. If omitted, the last modified time of the source file is used.

The third argument indicates the time zone to use. If used, the first and second arguments are mandatory. If omitted, the default is to use UTC.

5.3.6.1 Format

The pattern string specifies how the time should be formatted. It is a subset of the format used by the `java.text.SimpleDateFormat` class in Java.

In the pattern, unquoted alphabetical letters ('A' to 'Z' and 'a' to 'z') are processed as a time component. All other characters are literals, and are copied to the output unchanged. Text inside single-quotes (`'`) are also treated as literals. To output a single-quote, use two consecutive single-quotes.

The recognised time components are shown below. All other letters are reserved, and should not be used. (For debugging purposes, they generate a '`#`' in the output).

`G` era designator (always `AD`)

`y` year (just the century if the width is 2 or less)

`M`	month (1-12)
`D`	day in year (1-366)
`d`	day in month (1-31)
`H`	Hour in day 0-23
`k`	Hour in day 1-24
`K`	Hour in am/pm 0-11
`h`	Hour in am/pm 1-12 (commonly used with am/pm)
`m`	Minute in hour (0-59)
`s`	Seconds in minute (0-59)
`S`	Milliseconds (this will always be zero)
`a`	"AM" or "PM"
`z`	timezone `UTC` or `+`-*dd:dd*
`Z`	RFC822 format timezone `+`-*dddd*

The number of times a letter is repeated determines the minimum width of that field. Numeric fields are left padded with zeros to meet the minimum width. The width has no effect if the value is already equal to or greater than the minimum width. The width also has no effect on non-numeric values (namely G, a, z and Z).

These examples are for 5:29:59pm on the 30th of June, 2002:

```
"yyyy-M-d"                  => 2002-06-30
"yyyy-MM-dd'T'HH:mm:ss"     => 2002-06-30T17:29:59
"yyyy.M.d HH:mm"            => 2002.6.30 17:29
"yyyy.MM.dd hh:mma"         => 2002.06.30 05:29PM
"d/m/yy h:mm:ssa"           => 30/6/02 5:29:59PM
```

5.3.6.2 Time

The time parameter can be one of four values:

`filetime` for using the last modified time of the source file (as indicated by the file system). This is the default if a time parameter is not supplied.

`buildtime` for the time when Transbuild is run to generate the target file. If multiple files are generated in a single Transbuild run, this time will be the same for all of them.

A literal time value according to the W3C Date Time Format. This format is roughly `yyyy-MM-ddTHH:mm:ss` or left truncations of it. Leading and trailing spaces are ignored. (The current implementation does not support fully the W3C date time format: it does not recognise fractions of seconds and timezone designators.)

A literal time value of the form `$Date: yyyy/MM/dd HH:mm:ss $`. The time is interpreted in the UTC timezone. Leading and trailing spaces are ignored.

This feature was implemented to support using CVS to manage the changes to the source files. Scripts can be written to automatically extract the last checked-in time from the source file.

5.3.6.3 Timezone

The timezone parameter specifies which timezone the formatted time should be shown in. It must be a string in the RFC822 timezone format: an optional `+` or - sign, followed by four digits denoting hours and minutes from UTC. For example, Australian Eastern Standard Time is `+1000` (or simply `1000`), USA Pacific Standard Time is `-0800`.

Examples:

```
timestamp('yyyy-mm-dd HH:mm:ss z', 'filetime')
          => 2002-12-25 00:00:00 UTC
timestamp('yyyy-mm-dd HH:mm:ss z', 'filetime', '+1000')
          => 2002-12-25 10:00:00 +1000
timestamp('yyyy-mm-dd HH:mm:ss z', 'filetime', '-0600')
          => 2002-12-24 18:00:00 -0600
```

Since the Web is a medium with global reach, it is recommended that you use always use the UTC timezone for your timestamps.

6 Annotated build script DTD

This section contains a DTD description of the build script file format along with annotated notes. The DTD can be found in the documentation directory. Public identifier for the build script is `-//hoylen.com//DTD transbuild buildscript 1.0//EN`. An XML Schema for the build script is also available.

The root element of a build script is the `build-script` element. Elements and attributes in the build script must be from the XML namespace of `http://hoylen.com/ns/xmlns/2002/transbuild/buildscript`. This element must have a `version` attribute, whose value currently should be `1.0`.

The `source` and `target` attributes specify the source tree and target tree, respectively. Their values must be directories. Relative paths are in relation to the location of the build script file. If they are omitted, their values have to be supplied via the command line when running Transbuild. A build script contains zero or more rules.

```
<!ELEMENT build-script (rule*)>
<!ATTLIST build-script
          version CDATA #REQUIRED
          source CDATA #IMPLIED
          target CDATA #IMPLIED
          xmlns CDATA
  #FIXED "http://hoylen.com/ns/xmlns/2002/transbuild/buildscript">
```

Rules have a mandatory `source-suffix` attribute. The `target-suffix` attribute is optional: if it is not present the target suffix is made the same as the source suffix. An optional `id` attribute can be placed on the rule - currently this is only used for documentation purposes. The contents of the rule is a list of zero or more processing steps.

```
<!ELEMENT rule (file-copy | xml-load | xslt | system)*>
<!ATTLIST rule
          source-suffix CDATA #REQUIRED
          target-suffix CDATA #IMPLIED
          id ID #IMPLIED>
```

The `file-copy` processing step is an empty element. It is only useful if the source file is to be copied to the target file without any processing. Place it as the one and only processing step in a rule.

```
<!ELEMENT file-copy EMPTY>
```

The `xml-load` processing step has an optional `annotate-with-source` attribute. The value of `yes` is assumed if the attribute is not present. The `xml-load` element contains a list of zero or more annotations.

```
<!ENTITY % annot_elems "(dir | ancestors | children | members)*">

<!ELEMENT xml-load %annot_elems;>
<!ATTLIST xml-load
          annotate-with-source (yes | no) #IMPLIED>
```

The four type of annotations each have their own element: `dir`, `ancestors`, `children`, and `members`. They all support the `annotate-with-source`, and all except `dir` support the `embed` attribute. Both of these are assumed to be `yes` if they are not present. The

annotation elements each have one mandatory attribute: either `file-name`, `file-suffix` , or `path` depending on the element)

```
<!ELEMENT dir %annot_elems;>
<!ATTLIST dir
          name CDATA #IMPLIED
          annotate-with-source (yes | no) #IMPLIED
          path CDATA #REQUIRED>

<!ELEMENT ancestors %annot_elems;>
<!ATTLIST ancestors
          name CDATA #IMPLIED
          annotate-with-source (yes | no) #IMPLIED
          embed (yes | no) #IMPLIED
          file-name CDATA #REQUIRED>

<!ELEMENT children %annot_elems;>
<!ATTLIST children
          name CDATA #IMPLIED
          annotate-with-source (yes | no) #IMPLIED
          embed (yes | no) #IMPLIED
          file-name CDATA #REQUIRED>

<!ELEMENT members %annot_elems;>
<!ATTLIST members
          name CDATA #IMPLIED
          annotate-with-source (yes | no) #IMPLIED
          embed (yes | no) #IMPLIED
          file-suffix CDATA #REQUIRED>
```

The `xslt` processing step has a single mandatory `stylesheet` attribute. Its value must be the file name of the XSLT script, relative paths are from the location of the build script. It contains a list of zero or more parameters in `param` elements. These `param` must have a `name` and one of `value-xpath` or `value-string`.

```
<!ELEMENT xslt (param)*>
<!ATTLIST xslt
          stylesheet CDATA #REQUIRED>

<!ELEMENT param EMPTY>
<!ATTLIST param
          name CDATA #REQUIRED
          value-xpath CDATA #IMPLIED
          value-string CDATA #IMPLIED>
```

The `system` processing step executes a system command. The command is placed in the mandatory `command` attribute.

```
<!ELEMENT system EMPTY>
<!ATTLIST system
          command CDATA #REQUIRED>
```