UNIVERSITY OF OSLO
Department of Informatics

Building a better Make

- Implementing PyMek

Morten Lied Johansen
mortenjo@ifi.uio.no

2005-02-22 20:40:24 +0100
(Tue, 22 Feb 2005)

Abstract

This thesis deals with the problems and solutions encountered during the development of PyMek. PyMek is a make-like tool for building software projects. It was written as part of a Cand. Scient. degree at the Department of Informatics at the University of Oslo.

PyMek uses XML-based buildfiles for project description, and MD5 checksums to determine filechanges. The system is designed to use platform-independent tasks for building the project. Several tasks are included in PyMek, but the system is designed with pluggable tasks in mind, allowing third-party developers to create their own tasks should they need them.

PyMek is written in Python, and only uses modules from the standard distribution.

Preface

In my work with this thesis, I have gotten invaluable help from many people, including my supervisor, Professor Hans Petter Langtangen, my friends and colleauges Einar G. Flesaker and Igor V. Rafienko, Frode Vatvedt Fjeld and the other regular posters to the no.it.programmering.diverse and no.it.programmering.python newsgroups, and the often talkative people in #python-cleese and #python-gilliam on irc.FreeNode.org.

Thanks to my family and friends for help with proof-reading, and encouragement when things were going slowly, and a special thanks to my girlfriend for being a good motivator.

Contents

1 Introduction

1.1 What is PyMek?

To answer that, we need to look at what it is supposed to replace. So the question becomes: What is Make?

Make is a tool used for keeping track of which parts of a big project needs to be updated, and performs the commands to do so. The most common use is to build C/C++ programs, but other uses are also possible. Generally, all tasks where a file needs to be updated in response to the change of another can be handled by Make.

Make functions by reading what is known as makefiles, which describes what files are dependent on others. Such a rule is defined by a target, a set of dependencies and a set of commands to be performed if the dependencies have changed. Targets do not have to be files, they can also be so called fake targets, which is used to get Make to perform various tasks.

In addition to these rules, the makefile can contain a number of other things, most notably variables used to simplify the writing of more complex rules. There is virtually no limit to the complexity of the tasks you can have Make do for you, the biggest limitation is your own ability to keep track of the makefile you need to write.

This is perhaps where one of Makes biggest weaknesses lie, one that PyMek will try to amend.

When Make starts its work, it will either look at the target you instruct it to look at, or the target specified first in the makefile. Once it has located its first target, it will look at the list of dependencies, and if any of those are also present as a target in the makefile, Make will repeat the process there, going down the tree to find the roots. Once there, it will work its way up, making sure all files are up to date by running their associated commands if their dependencies have changed, finally finishing with the first target.

A few rules are built-in to Make, because they are so common. Depending on which version of Make you have, the built-in rules can vary, because that particular brand of Make could have a different view of what is “common”.

Make is also capable of defining functions which will generate rules on the spot, comparing the value of variables and choosing a rule based on the result, and more. A truly experienced makefile writer can do some amazing tasks, which would be excruciating to do by hand. The downside is that in order to become so skilled at writing makefiles, it usually require a lot more time learning Make than it would have taken you to do the work by hand.

PyMek will try to solve some of the problems with Make, and although it will never be as powerful, the goal is to do most of the things that most people use Make for, while making it easier to do so.

1.2 What is wrong with Make?

There are many things wrong with Make, some of which will be addressed by PyMek, some of which will be left for others. The problems that are most important to us, and therefor the ones we will look at are:

- Timestamp to track changes
- No built-in commands
- Complicated debugging of makefiles
- No dynamic dependency tracking
- Platform independence

1.2.1 Timestamps

By using the timestamp of a file to track changes, Make has found a simple and effective way to discover changes. Unfortunately, it is so simple that it gets tricked in various situations, leading to severe headaches for developers.

Consider the following example: Two groups work on a project, the first group works on a set of modules that provide some form of functionality to the rest of the application. The second group works on the application itself. During development, the apps group has a local copy of the modules, as they get developed. At one point, the apps group update their copy of the modules to the latest version, which was finished a couple days earlier. Those new modules, full of changes will then still have a timestamp that predates the latest changes in the application code, since the apps group continued developing with the old modules for a while.

This situation will not be caught by Make. PyMek will try to find a viable method to solve this problem.

1.2.2 No built-ins

A makefile can sometimes be a very complicated beast. Every now and then the developer needs to perform tasks that require advanced tools. In Make the developer is limited to whatever commands are available in his current shell environment. In todays world where portability is a major selling point, this approach is littered with problems. Any tool used in the makefile can turn out to be unavailable in some other shell, where some other developer needs to use the makefile to build the project.

1.2.3 Debugging

Because of the limited commands available in the makefiles, debugging your makefile is complicated. The script language used in makefiles was never designed for this level of complexity, and in many cases it simply is not good enough.

This forces developers to use external scripts to do tasks in their makefile and it is easy to lose track of what is going on in your own makefile. Once that happens, simple debugging techniques are useful, but because of the limitations even something as simple as inserting test messages at various points in the makefile is impossible without making it part of a target, which again introduces even more complexity.

Other things are also adding to the complexity and difficult debugging environment. Commands in a makefile has to be on a line starting with a single tab-character. This leads to countless hours wasted looking for the problem when your editor accidentally inserted 8 spaces instead of a tab.

1.2.4 Dynamic dependencies

This is perhaps the most complex of our problems. In order for Make or PyMek or any other build tool to make sure to update your files when dependencies change, it has to know what the dependencies are. In most, if not all build tools today, the developer is responsible for supplying this information in a makefile or similar. In big projects, this is dangerous, as the developer himself might not have a complete understanding of the dependencies involved, or he may have lost track as the project grew.

What is needed is some way for our build tool to figure out some, if not all, of the dependencies on its own, leaving the developer to concentrate on his code instead of updating a makefile.

If we were going to limit PyMek to a single programming language, this would be a relatively simple thing to accomplish, but we want a generic build tool. This means we need to figure out a way to guess dependencies automatically, in a generic way that does not need to know anything about the language used, or it needs to have a big set of rules for all the languages it might encounter.

The latter of those approaches is the easiest to implement, although harder to maintain, while the first is by far the most advanced. The problem is the various ways the languages declare their dependencies, automatically guessing how this language does it will be rather complicated.

1.2.5 Platform independence

In order for Make to work today, most projects of a certain size use a complicated combination of multiple Makefiles and a tool called autoconf.

This is because the various platforms that a project needs to be built on vary in numerous ways, some obvious, and some hidden away deep in standard libraries. This is one of the biggest problem in building projects today, because for many of the issues there is no “right way” of doing it, so the different platforms battle it out without much hope for consolidation into a common specification [WJCB].

Autoconf works by generating files for different platforms based on templates that specify which platform features are needed. These files are then processed in order to create makefiles suited for this particular system, so that Make can do its job even if there are underlying differences between the platforms it is run on.

The end result is that autoconf is a complicated beast that intimidates even people with PhDs, and is difficult to get right even with practice.

1.3 The Solutions

1.3.1 Timestamps

We need to find a simpler and more correct way of detecting changes in our project. One such method is to take a checksum of a file, and compare it with one saved previously. By creating MD5sums on our files and storing them in the build-file, we will be able to detect all actual changes, in a simple and error free way.

1.3.2 Built-in commands

By supplying a set of built-in commands, this problem can be alleviated, and by providing a way to make your own built-ins, PyMek will seek to give the developer all the power he needs, while still maintaining portability.

This requires that we as developers keep this in mind when creating our built-ins, so that they do not rely on platform-specific features. We will seek to provide this in a number of ways, mostly leveraging the power of Python itself.

1.3.3 Debugging of makefiles

These problems can be handled by doing a few things. We will provide a less error prone syntax for build-files, and make extensive use of internal logging that can be switched on by the user on request. By making sure all actions taken by PyMek is logged sufficiently, we will be able to provide the user with all the information he needs.

1.3.4 Dynamic dependency tracking

This is maybe the most interesting problem of all, and also one that is too big for a project like this one. The complexities involved are simply not something we will be able to handle, and is better left for some other time.

1.3.5 Platform independence

Some of the issues related to this problem is being dealt with by the standard Python package distutils, which tries to be a unified system for distributing Python modules and extensions in a platform independent way. We will therefore try to use distutils where we can, most notably when compiling C/C++.

In the SciPy project they are using an experimental extension to distutils for compiling Fortran, and it could be possible to use this in handling Fortran compiling.

2 The competition

The problems with Make has been around almost since its creation. They have rarely grown to more than annoyances though, and for that reason, little or nothing has been done with it in the time passed.

This is now changing, and several tools are either developed or being developed in order to make a new system to do the work of Make, without the problems. PyMek is one small contender in the arena, and when compared to some of the massive open source projects out there, PyMek is small and limited. However, we are focusing on our solutions, and with a bit of luck the things we learn can be put to use in the next iteration of PyMek, or in one of the larger projects.

2.1 Distutils

Distutils is a part of the standard Python distribution. It is designed for Python modules, extensions and applications, with commands for building, distributing and installing these. Its main focus is on the phase that comes after a project has been completed, and not so much for the actual development.

Distutils have been under heavy development and is determined to support all situations where you need to build a project in an environment you do not have complete control over. For that reason, the parts of distutils that deal with building are quite robust and since it is also a part of the standard Python distribution, we will take advantage of this in our development.

As a replacement for Make, Distutils does not measure up. It does not have any idea about dependencies, it simply works on a list of files without any metadata linking them together. It is not able to tell the difference between two versions of a file.

2.2 Jam

Jam is a software build tool that is part of the Perforce SCM System. It can be used by itself, and is widely used in both commercial and academic settings. It aims to do the same as Make, but have yet to reach the popularity of Make.

Jam has some dependency detection built-in for C/C++ projects, but also allow the user to specify dependencies in a Jamfile. Jam has its own script-language for use in the Jamfiles, which in their own words is “simple yet unintuitive”.

Jamfiles have rules, actions, targets and dependencies. A rule describes a procedure that takes parameters and does something to them. The equivalent of a function in most programming languages. An action is a special-purpose rule that specify system commands to be executed. Actions and rules work on targets to do the actual building of the project. A rule can define which targets depend on the target that was passed to it, and in that way define a dependency tree.

Jam is a real contender for Make, solving many of Makes problems. Unfortunately it suffers by having a special-purpose script language that is as they say “less than intuitive”, and being closely linked with the Perforce SCM System. It is possible to use Jam without any connection to Perforce, but that is not obvious from a first glance at their webpage, something which could be part of the reason why it has not been adopted by a wider audience.

2.3 Apache ANT

ANT is an attempt at replacing Make with something that is less OS-bound. They have the same idea of built-in tasks that take care of execution as we have planned for PyMek, and is implementing their system in Java to be truly cross-platform.

ANT has many similarities with PyMek. It uses XML for its build-files, it uses platform-independent tasks for building the project, it is written in a fundamentally portable language and so on.

An ANT build-file, can contain targets, tasks and properties. Properties act as variables that can be used to determine the execution of tasks or to give extra information to tasks.

ANT suffers from an overly complicated build-file, that has taken XML to the extreme. It includes ways of specifying flow-control, variables, and more, all of which conspires to make even simple build-files overly verbose and complicated. Nevertheless, ANT has established a large following, springing out from the projects hosted by Apache.

2.4 SCons

SCons won the make tool contest at the "Software Carpentry" design contest sponsored by the Los Alamos National Laboratory in January 2000. It is based on an older tool called Cons and is implemented in Python.

SCons is designed to be portable and flexible, and as we would expect from a contest-winner, fairly well thought through. It is designed to be modular and easily extendable.

SCons uses SConstruct files to provide information about a project, using Python as its scripting language. The files are written using Python syntax, and use regular Python code to set up an environment that contains all the information SCons needs for building. They have done their best to enable people who have no experience with Python to be able to define their own SConstructs.

SCons includes modules for scanning sources for implicit dependencies, a signature system and a builder system. These modules are easily extended, making it relatively easy to add support for new languages or tools. The system is meant to be platform-independent (in much the same way as ANT and PyMek) and support for several tools are included.

2.5 Rake

Rake is an implementation of Make in Ruby. It does not have that much in extra features compared to Make, but instead of Makefiles it uses Rakefiles where you use Ruby to specify targets and dependencies.

Its advantages over Make is portability, as it will run anywhere Ruby runs, and the fact that it uses Ruby for the Rakefiles, which has a cleaner and more understandable syntax.

For the time being, Rake has not matured into a tool which solves all of Makes problems, although the potential is there. Rake is also relatively new, so interesting developments could happen in the future.

2.6 AAP

AAP is a tool designed to be a part of a larger project called the A-A-P Project. AAP is what they call a Recipe Executive, that takes Recipes and performs the actions detailed in them. Recipes are intended to be able to download, build, distribute, and install software, websites, or files in general. The plans are extensive, but only a small part of the complete project is finished.

AAP is written with portability in mind, and designed to be part of a larger project. It is written to be combined with other modules or existing tools, trying to take advantage of what is already available.

It uses Python as a scripting language for recipes, although the recipes are not written in Python. It has limited support for detecting dependencies, but only for C/C++. It is also implemented with portability in mind, and avoids use of shell commands where possible by providing a set of built-in commands.

AAP is at the base of what looks like a very promising project that goes far beyond the simple features of Make. Unfortunately, there is still much to do before the project reaches its goals. AAP itself is in a basic state, providing the simplest of the ultimate featureset, but is still capable of doing most of the things Make can do.

2.7 Maven

Maven is more than a built tool, it is a project management tool that happens to include a build-system as one of its core components. As such, it is more complex than the others we have looked at here, but also allows for more advanced features such as publication of project pages on the web, complete with changelogs and other relevant information.

Maven uses XML for its project descriptors, but are developing other interfaces, such as RDBMS. The descriptors contain information relevant to the project, not just dependency information about source code.

When compared to Make, Maven is a behemoth of added functionality, and the comparison is in no way fair. The one point where Maven fails is that it is only for Java. For this discussion, this is a major failing, but if you are building a Java project, then Maven is certainly one of the top tools to use.

3 Using PyMek

This chapter will give a tutorial in the use of PyMek, aswell as introduce some of the concepts that are vital for the further understanding of what PyMek does. It will also double as a tutorial for distribution with PyMek, and can be read as a standalone document in the `doc` directory of the distribution.

A more detailed reference may be found in the appendices of the thesis, or in the file `reference.rst` in the `doc` directory of the distribution.

First, let us have a look at our interface to PyMek. PyMek can be used as a package from any Python program, but for most people, they will use PyMek as a standalone program. In this chapter we will be focus on use of PyMek, the program, but in so doing we will also take a look at how the various modules of PyMek work together.

3.1 Getting started

Let us invoke the built-in help command:

```
mortenlj@atlas pymek $ pymek.py --help
usage: pymek.py [options] <target1> <target2> ...

options:
  --version             show program's version number and exit
  -h, --help            show this help message and exit
  --loglevel=LEVEL      Set level of logging to LEVEL. Level is one of:
                        OFF, DEBUG, INFO [default], WARNING or ERROR.
  --logfile=FILE        Write log to FILE, according to loglevel.
                        [default: No logfile]
  -f FILE, --file=FILE  Build project according to FILE. [default:
                        pymekfile.xml]
  -t NUM, --tasks=NUM   Execute at most NUM tasks at once. [default: 2]
  -s OPTION VALUE, --set=OPTION VALUE
                        Set an OPTION to a VALUE.
```

As we can see, PyMek does not have all that many options for use on the commandline, this is because most of its functionality is controlled using the PyMekfile. Still, there are a few options here, we should know what they do.

The `--loglevel` option is how you tell PyMek its verbosity level. DEBUG will print a wealth of information, telling you everything it is doing. On the other end of the scale is "OFF" where it will keep completely quiet, not even printing error-messages. The default is "INFO".

With the extreme amounts of debugging output, it would sometimes be good to save the output to a file. This is accomplished with the `--logfile` option, which sets a filename for PyMek to write the log to. The log will contain the same as the screen output that PyMek generates. If a task calls an external program however, the output from that program will not end up in the logfile, unless the task has taken special steps to capture it.

When PyMek is executed, it will look for a file called `pymekfile.xml` in its current directory, unless the `--file` option is used to tell it otherwise. The PyMekfile contains information about the project, and is where PyMek gets its instructions from. A valid PyMekfile is required for doing anything with PyMek besides printing the help and version.

If you use a multi-processor computer, the default settings for PyMek might not be to your liking. Normally, a single CPU can handle two simultaneous tasks at the same time for maximum efficency. On a computer with more than one CPU, you should adjust the number of tasks by using the `--tasks` option. Normally, you would set it to *Number-of-CPUs* + 1.

The final option is something of a workhorse. It allows you to set any configuration variable to any value. This is so that the commandline does not have to know about all the variables any given task will accept, it will only propagate the variable given to the main configuration, so that the task can pick it up from there.

3.2 Configuration

The `--set` option is a powerful way of adding arbitrary configuration variables to PyMek at each running, but if you want to save your settings, there is a possibility for that aswell.

PyMek will look in a few pre-determined locations for a configuration file. There is no platform-independent way of locating the default system configuration directory, and the same applies for a users configuration directory. For that reason, PyMek will simply use any and all files that match the following six locations on the current system, expanding the values of `%(variable)s`-expressions:

```
/etc/pymek.conf
%(home)s/.pymekrc
/Library/Preferences/pymek.conf
%(home)s/Library/Preferences/pymek.conf
%(profile)s/Application Data/pymek.conf
%(appdata)s/pymek.conf
```

The variables are expanded according to this table:

Variable	Value
%(home)s	The value of the HOME environment variable
%(profile)s	The value of the ALLUSERSPROFILE environment variable
%(appdata)s	The value of the APPDATA environment variable

This file uses the so-called INI-syntax that was made popular by Microsoft Windows. The file is divided into sections, where each section contains a number of variables and a corresponding value. PyMek itself only cares about the PyMek section, but tasks are allowed to have their own section. By convention, a task should use a section by the same name as itself, but there is no enforcement of this, which allows several versions of a Java tasks to share some configuration.

The options set in this configuration file will be overridden by the ones on the commandline, either by the regular options, or by the `--set` option. Unless specified by the use of dot-notation, the `--set` option sets variables in the PyMek section. In order to set a variable in another section, simply prefix the variable with the sectionname and a period, for eksample `--set Java.compiler jikes`.

As a last resort, some tasks will allow the use of parameters in the PyMekfile, which we will return to shortly.

3.3 A simple example

To see how PyMek works, let us look at a small example of a project that does a complex version of the “Hello World!” example for Java. We split our Java program in two files, so that PyMek can actually do some work.

First, the sourcecode for our example, this class takes care of printing:

```
class out
{
    public static void print(String txt)
    {
```

```
        System.out.println(txt);
    }
}
```

Second, the main program:

```
class hello
{
    public static void main(String args[])
    {
        out.print("Hello world!");
    }
}
```

In order to combine these two into a project, we have the following pymekfile to define the project and the dependencies between the two files:

```
<?xml version="1.0" ?>
<pymek xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://folk.uio.no/mortenjo/PyMek
        http://folk.uio.no/mortenjo/PyMek/pymekfile.xsd">

<node>
    <name>Java_hello</name>
    <filename>hello.class</filename>
    <children>
        <node>
            <filename>hello.java</filename>
        </node>
        <node>
            <filename>out.class</filename>
            <children>
                <node>
                    <filename>out.java</filename>
                </node>
            </children>
            <tasks>
                <task>
                    <command>Java</command>
                </task>
            </tasks>
        </node>
    </children>
    <tasks>
        <task>
            <command>Java</command>
        </task>
    </tasks>
</node>

</pymek>
```

Here we have the target of the project, in a <node>-element called Java_hello. It will have the filename hello.class when it has been built. This target also has some children, and a task, defined in the <tasks> part of the <node>.

Each childnode is a new target, defined in its own <node>. We see that the Java_hello target depends on two targets, hello.java and out.class. out.class is very similar to Java_hello, depending on out.java, and using the same task.

The task is defined as the name of a task, in this case the built-in task called Java. Both Java_hello and out.class can be used as targets when we invoke PyMek from the commandline, building the targets neccessary to create the given target. A test run of PyMek building the Java_hello target:

```
mortenlj@atlas pymek $ pymek.py Java_hello
INFO: Executing Java for target out.class...
INFO: Executing Java for target Java_hello...
INFO: Success!
```

..and we can run our program after a successful build:

```
mortenlj@atlas pymek $ java hello
Hello world!
```

3.4 A look at the PyMekfile

The previous example might look like it is overly verbose, and it is, for such a small project. The benefits of using XML does not shine through in such a small example, but once the project expands, there will be more advantages.

The complete XML Schema for the PyMekfile can be found in the appendices. We will however give an introduction to the important elements here.

The basic buildingblock of a PyMekfile is the <node>. A node describes a target for PyMek to work on. The minimal node has atleast one of either <name> or <filename>, like this:

```
<node>
    <filename>somefile.c</filename>
</node>
```

Before we explore the details of nodes, we should look at the only other way of refering to a target. The <noderef> element is also a way to refer to a target, but it has no contents. Instead, it simply points to a node that has been defined elsewhere in the PyMekfile.

In addition to a name and/or filename, a node can have a list of <children>, which lists the dependencies of this target. The children are other nodes or noderefs that describe targets. Each of the nodes in this list can have the same elements as any other node. This is how we define the dependencies throughout the PyMekfile.

Because of the way we can use noderefs, it might be more tempting to define each node at the toplevel and use noderefs to list dependencies. This has the sideeffect that whenever someone runs PyMek with this PyMekfile, all nodes are checked and updated, as PyMek will build all toplevel nodes if not given a specific target.

Obviously, a noderef that points to a node that directly or indirectly depend on the noderef is a cyclic dependency, and as previously discussed, for PyMek to be able to work, it needs a Directed Acyclic Graph, so doing that is not valid.

Most nodes will have a <tasks> element. This lists the tasks that are needed to update this target. Each task is executed once, in the order listed. We will return to how tasks are defined later.

The final element that may be present in a node is <MD5>. This is included by PyMek, and is a storage for the MD5 checksum used by PyMek to track changes. If PyMek does not find a MD5 element, it treats the target as changed.

By using all we have learned, we can now list most of a typical node:

```
<node>
    <name>Target</name>
    <filename>a.out</filename>
    <children>
        <noderef>somefile.c</noderef>
    </children>
    <tasks>
        <!-- some tasks here -->
    </tasks>
</node>
```

In order to create complete and useful PyMekfiles, we need to use tasks. A <task> element defines a task to be executed in order to update the current target. A task looks like this:

```
<task>
    <command>C_compile</command>
    <param>include_dirs</param>
</task>
```

Here, the contents of <command> is the name of the task to use, and the contents of <param> is a parameter to that task. A task has only one command, but can have as many params as it likes.

3.5 So what does it actually do?

When you run PyMek, it will read the PyMekfile, and create what we call a buildgraph, or buildtree. This is made up of the targets described in the PyMekfile, and each target is associated with tasks and names as you would expect.

If invoked without any targetname on the commandline, PyMek will try to build all toplevel targets in the buildgraph, which is different from the default behaviour of Make. If one or more targets are listed, PyMek will try to build those. We can call these targets the destinations.

Explicitly giving PyMek targets to build, or relying on the default behaviour, the process is the same. For each destination target, PyMek will first process any targets listed as a child of the current one, recursively repeating the process until it reaches a target that has no child.

At that point, PyMek will start working. For each target, it will check if any of the child-targets has changed, using an MD5 checksum. If a child has changed, PyMek will execute the tasks associated with this target, and proceed like this until reaching and eventually rebuilding the destination target.

PyMek is not stupid, so if it is started with multiple destinations, that somewhere down the line are dependent on the same targets, that target will only be processed once.

4 Describing the dependency graph

4.1 What do we need?

Our fileformat would need to help us describe the details of our project. It would need to keep track of the dependencies, and all the other associated data that may or may not be present.

We also want a fileformat that is easily readable (and writable) by humans, as we have not developed a tool for creating these files for us.

For simplicity, a format that is as easy as possible to read and write by a program is also needed, we are after all going to do a lot of that.

So our requirements are:

- Keep track of dependencies
- Keep track of metadata
- Human readable
- Machine readable

4.2 What is there to choose from?

There are many possible alternatives that could be used to fulfill our requirements, and it is useful to compare the various alternatives.

4.2.1 Our own syntax

The most direct approach would be to define our own syntax and write our own parser. This would give us the ability to decide exactly what to keep track of, and how it should be written down. We would then be able to write a customized parser specialized in parsing just this kind of files, which could give us a slight efficiency increase. The downside is that writing a parser is hard work, and making sure the syntax we define is complete and does not have holes in it is a complicated task.

4.2.2 Makefiles

Another approach is to simply reuse the well known Makefile from make, this is after all the tool we are trying to replace. This would also save us the trouble of having to educate the developers about how to write their PyMek files, as they would simply use the same techniques as before. The problem with this approach is that a the Makefile syntax as a whole is extremely complex, and we are only using a subset of that. Also, we would still have to write our own parser, since PyMek is written in Python while Make is written in C, so we could not just lift the parser code out of Make and into our own project. Not to mention that PyMek will need completely different datastructures from what Make uses.

4.2.3 Some other projects build-file syntax

There are other build tools, similar to Make and PyMek out there, so why not use one of their syntaxes, and possibly their parsers? Much of the same goes for this approach as the Makefile approach. Their parsers will simply not fit into our program, and we would have to battle with a syntax that is not exactly as we would like it.

4.2.4 XML

XML is a buzzword in IT today, for good reason. If we take a look at XML, we can see that most of the advantages we would get from choosing one of the others are retained, while many, if not all, of the disadvantages are removed. By using XML, we get an already written parsing engine, that just needs a little tweaking to read our syntax. We get to define exactly what elements are included in our syntax, and how they are connected. XML is readable by humans and machines alike, and is a format designed to keep track of data and metadata, so should fit nicely to our needs.

XML seems to be a perfect candidate, and with support for parsing XML already present in the standard Python distribution, it is a natural choice.

4.2.5 YAML

YAML has a less verbose syntax than XML, but is designed to provide more or less the same basic functionality. It is less widespread than XML, but several programming languages have libraries for working with YAML, including Python. In order to be less verbose, it has instead had to add more cryptic markup, in the form of indicators that have special meanings.

YAML provides the same features as XML, while being slightly less intrusive for human readers and writers. Unfortunately, it is not as widespread, and there are fewer tools that work with YAML than XML, so while you might use generic tools to handle XML when it becomes complex, the same might not be possible with YAML. When we add in the fact that Python does not have support for YAML in the standard library, creating an extra dependency, it comes short of becoming our preferred syntax.

5 The PyMek File

Our PyMekfile needs to describe the dependencies between files, track information about the file needed for the processing, and know what to do in order to update it should one of the dependencies have changed.

The most intricate part of this, is keeping track of dependencies. A dependency tree is in many cases, just that, a tree. Unfortunately for us, in the most general terms, it is slightly more complex.

The end file/target of our project relies on one or more other files. These again can rely on other files and so on. Some of the earlier files may rely on the same few files, but at no point will you find a file depending on a file that in turn depend on it. This is called a Directed Acyclic Graph (DAG), and a few of the characteristics we know about DAGs will be of service to us when we seek to define how our PyMekfile should be laid out.

The unfortunate consequence of this is that our hope to use straight forward XML is hindered. XML defines a tree, and as we have just discovered, our dependency tree is not really a tree. So what should we do? Abandon XML and find some other way? If there existed a good way of describing DAGs in a simple textfile that was also human-readable, that might be the solution, but nobody has really found a way to do that. The best way to describe a DAG is typically to describe it like a tree, and insert references to places where the DAG does not fit into the tree model. And if we chose this way of doing it, we might as well use XML anyway, just inserting some form of ID tags to track connections that do not fit in the tree.

Since we are not handling the creation of the PyMekfile, but leaving that to the developer, we need to have a simple form of ID tags, that will be easy for the developer to use when writing the PyMekfile. A natural choice would be simply the name of the file in question, but as we can also have targets which are not actual files, we would need to allow the developer a chance to make up names on his own.

XML gives us the ID and IDREF types, which gives us exactly what we need to track the relationships between a name, and its reference somewhere else. An ID is simply a unique name, and the PyMekfile wont be a valid XML-document if this is not so. Similarly, an IDREF is a reference to an ID, and if it is not, the PyMekfile will again be found invalid.

5.1 Reading and handling XML

There are three models for parsing XML in Python. DOM, SAX, and a third model which is a pythonic view of how XML should be “translated” into Python objects.

DOM, or “Document Object Model”, is designed for processing a whole XML-file and keeping the entire structure in memory at once, in the form of a tree of objects. The model defines a set of objects and how they relate to eachother. An object in the DOM has methods for finding child nodes, siblings, parents and other information you might need. The DOM idea is very close to our requirements for the parsed information, and for that reason we will take a look at various parsers that work with the DOM model.

SAX, or “Simple API for XML”, is designed for speed and flexibility and presents the XML as a stream of events. When the SAX parsers reads something, it will call an eventhandler to process the tag, then proceed to the next bit of information. This makes it possible to process a big XML-file without having the entire structure in memory, but makes it harder to do random access of the information. SAX is not suited for PyMek, since we would need access to all parts of the information at all times, and so we will not be looking at SAX parsers.

The third model is not really a single model, but more of a collection of ideas on how XML best fits into the Python way of thinking. Different people have different ideas and has implemented them in their own parser. We will look at a few of these, since they present a different aspect of working with XML from Python.

We will first look at the two DOM parsers provided with Python, then a few of the commonly used parsers provided by third-parties.

5.1.1 pulldom

pulldom lives in the standard library as `xml.dom.pulldom`. Pulldom tried to combine the best of the SAX and DOM models for XML-parsing. It starts out processing the XML as a simple SAX parser, but when you arrive at the information that interests you, it can switch to a DOM model for a small subset of the entire XML-file, making it possible to use the DOM way of thinking while avoiding loading the entire structure into memory at the same time.

Pulldom is described as one of the best parsers for what it does, but a bad solution for anything else. This is because the mix of models does not always deliver the best of both worlds, but commonly give you the worst. The complexity of SAX is still there, while the problem of loading a big structure is still present if you can not limit your window of interest enough. For most things, other parsers are better, easier to use or simply better known.

5.1.2 minidom

minidom is part of the standard library, in the module `xml.dom.minidom`. minidom is, as the name implies, a light-weight implementation of the DOM model. It tries to provide a simple and small parser module for working with XML. It does not have the full power of the more evolved modules that work with DOM, but makes up for it by being smaller and most importantly, by being in the standard library that ships with Python.

minidom allows us to parse XML into a complete DOM datastructure, containing all the information we need. The nodes have methods for accessing children, siblings and so on, providing all the features of the DOM. Unfortunately, the nodes in the DOM does not match up with the nodes as you would normally think of it when viewing the XML-file, and some processing is needed after minidom has done the raw parsing, in order to get something that is useful to us. This is the case with most of the parsers, so not so much a failing, as a wish for something that maps more closely to how we think of the information.

5.1.3 ElementTree

ElementTree is centered around a data structure for representing XML. As its name implies, this data structure is a hierarchy of objects, each of which represents an XML element. The package includes a parser that will read an XML file into a tree of corresponding Element objects, giving a pythonic representation of the XML-file. Creating new elements in the tree is as easy as creating a new Element object and attaching it to its parent element.

ElementTree does not follow the DOM or SAX models, but is instead a package designed for use in Python, and to be as much in tune with the Python way of doing things as possible.

5.1.4 Amara

Amara is really a toolkit built on top of a lower level toolkit called 4Suite, but for this discussion Amara is the important bit. Amara does much of the same things as ElementTree, in that it tries to give the programmer a pythonic tool for working with XML.

Amaras way of doing things give the programmer a very direct link between the XML he is parsing and the objects Amara delivers. If you have a tag `<name>` in XML, this would end up as an attribute `name` of its parent element. The top element in the XML file would similarly become an attribute of a `binding` object, which is one Amara creates for you when you start parsing a file.

Creating new elements require the use of a factory-function, but once you have a new object, you can simply append or assign it where you want it to go.

5.1.5 pyRXPU

pyRXPU is the unicode version of pyRXP, which is a Python wrapper around the RXP XML-parser. Many will argue that since XML is inherently unicode, pyRXP is not an XML-parser until it supports unicode. For that reason, the developers of pyRXP has created pyRXPU. Unfortunately, pyRXPU is only "alpha", and is not actively supported by its developers at this time.

pyRXPU will parse the XML and create a structure where each node in the XML-file is represented by a 4-tuple. The 4-tuple returned by the parser is the top node, and it has as its elements the name of the tag, a dictionary of attributes, a list of childnodes and a spare for customization. The list of childnodes is simply a list of 4-tuples for each node, which in turn have the same 4 elements.

This structure has the advantage that it is a very effective way of storing the information, without using objects. From studying the documentation, I get the impression that you can add new tuples when you need to add nodes, but all my searching did not turn up any conclusive evidence of this, and neither did I find any mention of a way to generate XML from the datastructure.

5.1.6 pxdom

pxdom is a pure-Python implementation of the W3C DOM standard, with complete support for DOM Level 3. It has its own built-in XML parser and is compatible with Python 1.5.2 and later. It has been designed to be as true to the standard as possible, concentrating on correctness rather than efficiency.

pxdom is compatible with minidom, meaning it provides the same methods for parsing the XML, and the nodes are similar. The difference is that pxdom provides several advanced features and is capable of doing more with the data than minidom is.

pxdom has the advantage that if we are careful to use only the standard features, it is a drop-in replacement for minidom. This way we could distribute PyMek without it, while still allowing users to take advantage of some of the features that pxdom supplies for us. This is something to keep in mind for the future development of PyMek.

5.1.7 Summary

Since we would like PyMek to function without too many external dependencies, it would be beneficial to use one of the parsers that ship with the standard Python distribution. In order for us to be able to update the PyMekfile when we get new information, it is easier if we have a representation of the XML ready at hand. For this, the DOM-model comes in handy, and the minidom-parser is our tool of choice. By using the DOM-model, we can add, remove or change elements in the XML-tree and when we are done, we can tell the tree to print itself to file.

When converting a DOM-tree to a build-tree for our use, we unfortunately do not have a direct mapping between the two. A rather extensive approach is needed, and in order to keep to our tactic of adding elements to the DOM-tree at the right places, we need to keep a link between a node in the build-tree, and an element in the DOM-tree. We also have to keep in mind all the subtleties of handling XML, such as the whitespace between tags being just as much a part of the document as the tags themselves.

5.2 Parsing the PyMek File

A good way to build a parser of this kind is to take advantage of our knowledge of the tree, and split the parser into simple, easily identifiable parts, as done in [DIP9]. We do this by having a general parse-method which calls a corresponding method for each type of node in the DOM-model. In our case, we have the handlers `parse_Document`, `parse_Text`, `parse_Comment` and `parse_Element`.

Python helps us out again, with its ability for introspection. We can create a string, concatenating `parse_` with the DOM-nodes classname to get the name of the specialized parse-method to call. We then use `getattr()` to call the chosen method.

The Document-node is the top node in the DOM-model, and does not actually contain anything useful to us, but by starting there we can have our general parse-method work its way down, simply by parsing all childnodes.

Because of the way we dispatch parsing to the various methods, we need `parse_Comment` even though we plan on ignoring comments all the way. `parse_Text` is also quite simple, by just storing the text in a variable until another handler picks it up for use. The real action is controlled by the `parse_Element` handler, because all XML-tags are Element-nodes in the DOM-model.

Recycling a good idea, `parse_Element` does a similar dispatch as the first parse-method, only this time it chooses a handler based on which tag is currently being processed. So in the same way as we have `parse_<something>` to handle the various nodes in the DOM-model, we now have `do_<tag>` to handle the various tags present in our XML.

5.3 Building a Build-tree

When I first started working on the parser, I had already written a version of the Node-class used in the build-tree, in order to have something to work with. By doing this, I had an idea of how the parser would need to string the parts together, and the start of an algorithm for the actual building later on. It became clear almost immediately while working on the parser that the Node-class I had written was badly designed in several key areas, and I had to rewrite it.

For each `do_<tag>` method, we know what kind of childnodes it should contain. This knowledge is useful in determining how to handle the current node and its children. Every `parse_` and `do_` method will return something that is a good representation of the data it has parsed.

A call to `do_node`, which handles the `node` tag, will return an instance of `Node`, initialized with the correct values according to the contents of this DOM-node. A call to `do_filename`, will return the contents of the `filename`-tag as a string.

By calling `parse(childNode)` we can get the results that correspond to the current childNode, and because we know what to expect, we can insert that into a datastructure that describes the node we are currently handling. This way, we can process the entire PyMekfile and the original call to `parse` will return a `Node` object that points to the top of our build-tree.

6 Loading Tasks

6.1 Tasks, what are they?

Tasks are the basic building blocks that will do our work. Everything you could possibly want to do using PyMek, needs to be implemented as a task, or a combination of several tasks. Since it would be impossible for any one person to come up with all possible tasks someone would need, we need some kind of mechanism for loading tasks that were not there when PyMek was shipped.

In essence, we need to build a kind of plugin-system for PyMek, where tasks can be loaded and "plugged in" where we need them. This involves loading code objects from anywhere, written by anyone. There are a few complications here that we will look at later. For now we will concentrate on the needs of such a plugin-system.

We need a system where we can load a plugin, and know how to use it without having to do anything special. It would need to be ready for combination with anything, in various ways. At this point it could be useful to remind ourselves that when it comes to plugins, there are three parties. First there is the user, who actually is a developer himself, namely the person using PyMek to build a project. Second, there is us, who made PyMek. The third entity is the plugin-creator, who can be anyone, including one of the first two parties. For a system like this to work, there has to be clearly defined ways for plugins to interact with eachother, PyMek and the world at large. It is our job, as the PyMek developers, to make sure the plugin-creators know what their plugins should do.

Looking at a task in general, it will take something, do something with it, and return something else. In our case, we can assume that the tasks will do something with the dependencies of a target, and return the newly built target. But how does this work when linking several tasks after eachother in order to do what we want? Clearly, a task needs to get the result of the previous task, in order to continue processing of it. However, a task might not need the result of the previous task, as its job goes parallel to the first task. So any task would need two inputs. The original dependencies, and the result of any preceding tasks.

What, then, are the results of a task? It should be assumed that a task will operate in the confines of the file system. It will be given names of files to operate on, and it will return the name of a file which contains its results. This way, we have a simple and effective method of passing data between our tasks, which does not rely on advanced methods of datasharing.

Now that we have informally defined the needs for data transfer between PyMek and the tasks, we need to find out how we should go about loading the tasks into PyMek for execution. Depending on how much liberty we want to give our third-party task-developers, we can choose from a few models.

CORBA allows tasks to be created in almost any programming language, and even be executed on remote servers over a network. However, CORBA is complex and difficult to use, and it is difficult to see how we would need its advanced features for our simple plugin-needs.

Another model we could employ are XPCOM Components, as developed by the Mozilla Project for their plugin-system. It is possible to create XPCOM Components from a variety of languages, again giving us the freedom of choosing a language that suits the task at hand. They are also less complex than CORBA, with no network support and less of the advanced features that comes with CORBA. However, this is a fairly new technology, and not much support exists for it outside the Mozilla Project.

Which brings us to the simplest of all, which might still give us exactly what we want. By using a few tricks for import, we can have tasks defined simply as python modules. This will limit our choices of language a fair bit, and the feature-set is not as rich as CORBA for instance, but the upside is that its really simple to implement, and does not require much effort on behalf of either PyMek or the person writing the plugin. It also delivers on all the needs we have set so far, while not cluttering us with unneeded features.

6.2 Loading tasks

In order to use tasks as modules, as we decided in the previous section, we need to do some thinking about exactly how we will accomplish that. We can not expect all sysadmins to install any third-party task into his python-path,

so we will need to be flexible when locating tasks to load. We should also have a few default locations, where the tasks shipped with PyMek resides, right next to the ones supplied by other developers.

One possibility is that we create a tasks package, that in addition to supplying the modules present in its package directory will search a few extra locations in order to find more modules. But before we get to that point, we should look at the needs of the task developers. Is one module enough? Can we envisage a task that requires a set of modules organized in a package to perform its job?

I am tempted to say no, because such a complex task should instead ship as a generic python package, that can be installed into the python-path independently of PyMek, and only supply a task-module for PyMek to use which calls on tools in the generic package. However, this again requires sysadmins to install packages, and it will be hard to distribute your project if building it requires the user to first install a non-standard task (i.e.. not supplied by PyMek) package he does not need outside this one-time build.

The reason I can continue to say no, is that users can install python packages wherever they like, as long as they can get python to look in the right place for it, and that is an easy thing to do. So for now, PyMek does not need to support anything more than task modules.

Loading a module is luckily an easy task, and we just need to manipulate `sys.path` before doing so, in order to ensure that we are only loading the tasks we want. It would be a bad thing if we inadvertently load a standard python module with the same name as our task module. This extra bit of logic leads us to think more about how we will be using the tasks.

We will define tasks as subclasses of the *Task* class, residing in a task module. When we wish to make use of a task, we need to get an instance of the specific task-object that implements the task we want. This means we can use a sort of factoryfunction to get and load the task for us and return a suitable object. This gives us the ability to move the extra logic needed to find the task into a function of its own, so it does not complicate matters where we are using the tasks.

6.3 Dangers involved

When writing a program to run code supplied by a third-party, you should always keep in mind the security risks involved. You can never know what the third-party developer does with his bit of code.

I our case, there are two distinct threats. The first is for someone to write a task that harms the system in some way. Unfortunately, we need to give the developer access to anything, because the act of building a project can require anything from the system, including deleting files and creating new ones. There simply is no good cut-off point where you can say that developers should not have access beyond this point.

This scenario is unavoidable, except for simply checking the code itself and looking for harmful code. Manual labor to check what is going on is always prudent, as you should never rely on the computer to provide security. In this case it is even more important, since determining what is harmful and what is not is a problem that requires more than a simple program can provide.

The second threat is code that will in some way disrupt the execution of other tasks, or bring PyMek itself to producing erroneous results. By limiting the interaction between a task and PyMek itself to a few well defined points of contact, the threat can be minimized, but never really removed. Again, this stems from the requirement that tasks should be able to do almost anything, and that results in the possibility that what the task returns to PyMek is not as expected.

We need to keep this in mind when developing PyMek, to make sure to wrap our task-handling code with error detection and recovery in order to minimize any damage. In the end of the day though, there is plenty of ways for things to go wrong, and trust in the abilities of the third-party developer again comes into play.

6.4 Configuring the tasks

The tasks may do the work, but a task needs to be generic for it to have much value. But a generic task might not do exactly as the user wants, so there needs to be a way to configure the task to do things right.

In order for PyMek to present a useful interface to the user, it too needs a few configuration options, and we can tie these two sets of options together. By using the `ConfigParser` module from the standard Python distribution, we can get access to simple and efficient configuration. The `ConfigParser` allows a configuration file to have various sections, where variables are set. We will have a section for PyMek, and sections for each task, named appropriately after the name of the task itself.

PyMek will search for the configuration files in a few locations, based on what would be the normal place for system configuration files and the users configuration files. Unfortunately there is no platform-independent way of identifying either location, so we are reduced to looking in the places where the files are normally kept on a basic system. We have some default paths for linux, Windows and Mac OS, and hope that will be enough.

There are a few things that are not suited for configuration files alone, and with the `optparse` module, we are able to easily parse options on the commandline aswell, incorporating them into the configuration. This allows users to set options for a task from the commandline as well as the configuration file.

Finally, the PyMekfile itself can contain a list of parameters for a task. It is up to the task-developer how he wishes to use these parameters, but one option that we will make easy is to use it too for setting options. We have created a special method in our `config` module designed for this task, `task_config`. It will take a taskname and a list of parameters, and return a dictionary with the values of all options set for the specific task, including the ones from the parameters passed in.

6.5 Creating the standard set of tasks

In order for PyMek to be useful from the start, it needs to be shipped with a suitable set of tasks that will handle the most basic needs of developers. It should include tasks for compiling C/C++ and Java at least. In addition, tasks for document creation of various kinds, like LaTeX compile, dvips and dvipdf, are useful. Other basic needs are simple file operations like copy and move.

Many commands are simple calls to external programs, and if they require no more complicated logic than that, we can have a generic task that simply executes a command-line. We should make sure that our users know that this task is not suited for advanced use, and that they should implement a specialized task for most needs. Especially if they want to have portability for their project.

Portability was an issue we mentioned in the introduction, and one we should pay particular attention to. In many cases, the tools used in the build process are external to PyMek, and as such we are relying on them to be present. We leave it up to the task-developer to make sure that he does not rely on platform-specific features in his task.

For most of our needs, tasks are pretty simple calls to external programs that behave the same on all platforms. When it comes to compiling much used languages like C/C++ and Java, the story is a little different. For both these languages, there exists a multitude of compilers and many of them work differently in subtle ways. We will concentrate on making sure our C/C++ task and our Java task are versatile enough to handle the job.

6.5.1 The C/C++ task

This is maybe the most complex task we will work on, given the multitude of compilers, linkers and options that go into it. Luckily for us, much of the work is already done for us by the distutils package, and we will be using it for most of our needs.

Since distutils is a rather big module, it is not always appropriate to import it whenever someone wants to use a task. Therefor, we import the distutils modules we need only when executing the task. Creating executable C/C++ programs from source requires two steps, compiling and linking. Since these are two distinct steps, we

separate them into two tasks. Much of what we do will be similar, but there are enough differences to justify two different tasks.

The first thing we do in both tasks, is to use the `distutils.ccompiler.new_compiler` call to create a compiler object for us that encapsulates the default compiler for the current platform. Then we need to fix the list of sources for the task, since some of the sources we list are not sources the compiler or linker wants to deal with.

This is especially true for the compiler task, since it will throw an exception if the source it is given does not look like something it can compile. The solution is to check each source against the `language_map` dictionary in our compiler object, making sure to drop those that are not accepted.

Once we have a valid list of sources, it is all a matter of calling `compile` and catching the results. If the compilation fails, we raise a `TaskError`, if not, we return the results of the compilation.

For the link task, it is a bit more complicated. The linker needs to know what kind of target it is building. It will do different things, depending on whether it is building an executable, a shared library or a shared object. We need to find out, so that we can call the linker with the correct options. Since this is hard to guess at, we will simply have a parameter that decides, or use a default setting.

The parameter that decides which kind of file the linker should build is just one of several, that we will be reading from the configuration. The compiler needs to know where to look for non-standard headerfiles, and has an option for specifying `include_dirs`. The linker needs to know where to look for libraries, and if needed, which libraries to look for, and will use the options `library_dirs` and `libraries` to that end.

During development of this task, I uncovered a bug in distutils, which has to do with the use of unicode-strings. Regular strings in python are not unicode, but strings created from XML are. For that reason, all our sources and targets are defined in unicode-strings, instead of regular strings. This should not be a problem, but when doing compiles or linking, distutils will execute a call to `distutils.dir_util.mkpath`, which raises an exception if it is passed anything other than a regular string. This could cause problems on filesystems that is using UTF-8 charsets, but until the bug is fixed, we need to convert all our filenames to regular strings before passing them into distutils.

6.5.2 A Fortran task?

In the SciPy project, they have an experimental distutils module that does the same for Fortran as the CCompiler modules does for C/C++. It would be interesting to take this module and use it to create a Fortran task for PyMek, in much the same way that we created a C/C++ task. Unfortunately, the FCompiler module is not available outside the SciPy package itself, so for us to use it, we need to wait for the SciPy developers to release it for more general use, or require users to have SciPy installed whenever they want to compile something that uses Fortran.

The Fortran task will be near the top of the list of tasks to create at a later date, but for now it will have to wait.

6.5.3 The Java task

Java can be compiled by a number of compilers, and into a variety of actual targets. The GNU Compiler Collection now ships with `gcj` which compiles Java source code into a native executable. There is Jikes from IBM, which aims to do the job of the traditional Java Compiler a lot faster. And there are several independent Java Development Kits, most of which have slight differences.

Most often, developers who want to compile Java, is thinking in terms of the regular Java Compiler, which compiles sourcecode into bytecode which can later be handed on to a Java Virtual Machine. Before the arrival of `gcj`, this was all there was, but now they can compile into native machinecode if they want to. However, since this is not the common thing to associate with compiling Java, our Java task should create bytecode, using a suitable compiler.

Without going into the finer details of which Java compiler does the best job, and why, the possibility of deciding which compiler is best suited for the current job is a task best suited for a human. Since a regular build should not need too much tweaking, we need to decide on a set of standard options, and allow the user to override this should he want to, through the configuration.

A natural selection for primary compiler would be `javac`. It is the compiler that ships with the Java SDK, and even though it exists in different incarnations, if we are careful to only use features common between them, we should have a decent chance of getting it to work. Second on the list would be `jikes`, as it operates in a way to make it interchangeable with `javac` as much as possible. Finally, if all else fails, we can use `gcj`, as it does provide an option to function similarly to the regular Java Compilers.

The java task is easy to implement, only needing to remove non-java files from the sourcelist, and call the selected compiler. Unfortunately, there is a bug in Python that creates a small problem. When using Python 2.3.4, executing `javac` from a Pyhton thread will cause the process to exit with a Segmentation Fault, meaning the compile fails. There is no way around this, other than using a different compiler, or a different version of Python.

7 Scheduling and Execution

7.1 Strategies for handling multiple tasks

There are two extremes when it comes to how we decide to do the actual execution of tasks. The simple case is to start at the bottom, and execute tasks in a linear fashion, doing it one task after the other. The other extreme is to create a thread or an entirely separate process for each task, and doing some kind of scheduling to make sure tasks execute in the right order. If two tasks do not interfere or depend on eachother, they can execute at the same time.

The first of these two cases is the simple and straight forward approach that requires little or no effort on the behalf of PyMek or the developer. It is also the one which is unable to take advantage of multi-processor computers, which are becoming more and more common even in the desktop-market.

The second case is at the other end of the spectrum. It will in extreme cases create so many threads/processes that fight over processortime that hardly anything actually gets done.

Threads are a complex issue to handle and in some cases give little or no extra performance. This is especially true for Python since a thread wont necessarily be able to take advantage of multiple processors due to Pythons global interpreter lock which prevents simultaneous execution of threads. A pure python task will be less useful than one that executes an external program. However, in most cases it will be natural for a task to execute some external program for the actual work, only spending a small amount of time executing python code.

If we should decide to spawn new processes for each task, the complexities involved increase yet another level, and in general will make our job much harder, while not giving us much extra that we do not get with threads that execute external programs.

The model I feel is most useful to us would be one where each node in our build-tree is a separate thread, which creates and starts its child-nodes, and then sleeps until they are done. Once all its children has completed, it will then check to see how many threads are currently executing tasks, and get in line for execution.

This way, we can control how many threads are currently executing tasks, which is where we would get in trouble if there were too many threads. Threads that sleep give little or no overhead, so it would not be a big performance hit to have a big number of sleeping threads waiting for their turn.

It is important that we keep in mind the limits of our model, especially when we are designing and implementing tasks. Tasks that spend all their time executing python code might not be the best approach.

7.2 Scheduling our tasks

For any one target, there can be multiple tasks. The idea is that the tasks, when executed in order produces the target from the given sources (dependencies). Therefore, we only need to spawn new threads for each target, and not for each task. This makes it easy for our threads too, as they will only need to enter a loop where they execute the associated tasks in order.

Every node in our build-tree has the following method:

```
    def threadAction(self, semaphore, lookingFor=None):
        try:
    (1)     self._access.acquire()
    (2)     if not self._done:
                if lookingFor == self._name:
                    lookingFor = None
                if lookingFor == None:
                    self._done = True
```

```
            if self._children:
                threads = []
                for child in self._children:
(3)                 t = Thread(target=child.threadAction,
                               name=child._name, args=(semaphore, lookingFor))
                    threads.append(t)
                    t.start()
(4)             for t in threads:
                    t.join()
                for child in self._children:
(5)                 if not child._errors == []:
                        self._errors.extend(child._errors)
            if self._errors == [] and lookingFor == None:
(6)             self.doTasks(semaphore)
    except PyMekError, e:
        error = "%s: %s" % (self._name, e)
        self._errors.append(error)
(7) self._access.release()
```

Our strategy works like a depth-first search and build. We start at the top of our build-tree, and start a new thread for each target. Each thread is given a semaphore object and the name of a build-target. If no build-target is given, it means all targets are to be built. The semaphore works as a counter, making sure no more than a specified number of threads are doing CPU intensive work at the same time.

For each target, the process is similar to the start. When a thread starts processing, it will first try to acquire the lock for this target (1), in order to make sure it is the only thread trying to work on this target. Once the lock is acquired, it will check for a few things to decide if it should proceed or not (2). There are a few things to consider.

First, if another thread already processed this node, it will have set `self._done` to `True` and there is no reason to continue. If no other thread has been here, it is time to check if the build-target provided was `None` or the name of this target. If it was, the thread knows it will build this target, and it sets `self._done` to `True`.

The thread will start the process again, by starting a thread for all its childnodes (dependencies) (3) and waiting for them to finish (4). When starting the new threads, the semaphore and build-target are passed in to the childs `threadAction` method, so that it can do the same process again.

Once all children are done, it will check to see if they reported any errors, and propagate them in its own `self._errors`. This way errors are reported all the way up to the main thread (5).

If there were no errors, and the built-target says this node should be built, the thread calls this targets `_doTasks` method, passing in the semaphore object. The `_doTasks` method takes care of checking for changes in the childnodes and acquires the semaphore before executing its tasks.

8 What have we learned?

8.1 What did we do wrong?

Looking back at the work on PyMek, we should try to pin-point the places where we went wrong.

One such place is the PyMekfile. The use of XML for this file seemed like a natural choice when we started, but if we look at the code now, we see that a unproportionate amount of code goes into actually parsing the XML file into something we can use. This was one of the main reasons for choosing XML in the beginning, namely that we would not have to spend lots of time and effort in creating a parser from scratch.

The reason this took so much of our efforts is that the XML-parsing modules included in Python are not as good as they let on, meaning there is a lot of work involved in getting from the result of the parser to something that is actually useful. Make no mistake, parsing raw XML would have been a lot more work, but `minidom` stops short of delivering a usable product, leaving too much of the work for us to do. An option would have been to use one of the modules outside standard Python, which have evolved into real tools that parse XML into pythonic datastructures that we could have built on to create our own structures.

Another option would have been to abandon XML completely, using something else, more suited for the task. Some other tools use Python as the language not only for the tool, but also for the buildfile, giving users the added advantage of a powerful script language should they need it. This does open their tools up for problems we wont have, like malicious buildfiles doing harmful things, but on the other hand, they probably have an easier time dealing with parsing the file for use.

Well into the project, another one of our errors became apparent when we learned about the “Global Interpreter Lock” (GIL) in Python. For a long while, this was not an issue we had considered, until we suddenly realized it would severely hamper our plans to use threads for sharing the workload. Because of the GIL, even a computer with ten processors would still only execute one thread at a time, completely wiping out any advantage we were hoping to gain by running tasks in parallel. The reason we still have a tool that works, is that the nature of our tasks mean they will normally spend most of their time executing other programs, which live in separate processes, avoiding the GIL problem.

If we were to redesign PyMek again, the GIL would have to be taken into account from the start, meaning we would have to find other ways of sharing the workload than using task. The logical next step would then be to launch subprocesses that would do our work, with the problems of datasharing being our new problem. The issues surrounding datasharing between subprocesses is an interesting one, that have many solutions, so it would be solvable, and we would gain the advantage that all our tasks are able to execute in parallel on a multi-processor machine, not only the ones executing external programs.

With the new `subprocess` module that came with Python 2.4, this would be even easier, as we now have a platform-independent way of launching subprocesses.

8.2 What did we do right?

Even though our plan to use threads to share the workload, executing tasks in parallel, suffered a setback, I am still convinced the choice to use tasks were a good decision. The tasks can, and should, be made platform-neutral, so that the same tool and the same buildfile can be used whatever the platform. Separating these things into manageable pieces like tasks is a natural next step, allowing third-parties to add tasks when they need them.

Handling the execution of tasks in a different way is also easier when a task is contained in a single piece of code. Redesigning PyMek to use subprocesses instead of threads does not necessarily mean we have to change the tasks at all, they could quite possibly be carried over without change.

Using Python as our programming language was also a good choice. In total, PyMek has 1439 lines of code, which includes generous amounts of comments and blank lines. This is a very low number, compared to what we would need in almost any other language, and the work as a developer becomes all the more easy because of it. There are less code to keep track of, and for that reason less bugs to miss.

Making Python work for us is also easy in many ways. It has an astounding array of useful modules included in the standard library that takes care of much of the drudgery of implementing the same things in another language. Python has its strengths in its versatility and clarity, and the same project would in all likelihood have been a lot more complicated in another language.

8.3 Where should we go from here?

Moving PyMek to a design based on subprocesses instead of tasks would be a good thing to do, but as the tool is today, this is not where we should focus our efforts to begin with. The threaded solution works for most cases, because most tasks will be launching external programs anyway. There are other problems more apparent that should be addressed first.

The tasks included with PyMek are the most basic ones. Still, there are a lot of things PyMek can not do as of now, or that it does poorly. The first focus should be on creating more tasks, and extending the current ones to do more. A buildtool will not gain many users if it does not do the things the users need.

The extreme power of Makes makefiles is still not something we can match, but that is not why we started work on PyMek either. However, some of the features from makefiles would be nice to have, such as variables. This is however not something I think we can achieve until we leave XML behind as a language for our buildtools. The variables are not something that is absolutely necessary either, but something to consider for future development.

While we may wish to stay with using XML for now, we could do with better tools for parsing that XML. If we switch to one of the external modules for parsing XML, we should be able to extend our buildfiles more easily, as they are easier to work with. Both ElementTree and Amara show promise in that regard, and are also faster than the parser we are using at the moment.

Other areas of interest would be the various tasks involved in creating the buildfile for a project. Both automatic scanning of sourcefiles to generate a buildfile with correct dependencies and the automatic configuring of sources in the style of autoconf are important areas where there is a lot to learn. Scanning sourcefiles for dependency information requires extensive knowledge of how to spot a dependency in any given language, and is clearly a big project. Similarly, automatic configure is a problem that requires much work in finding out what exactly is the differences, and how they affect a project. Finding out how to get around the differences will also be interesting, considering the extreme number of different platforms that exist.

8.4 What should we do with it?

Given that we someday manage to create the perfect buildtool, what should we do with it? There is no guarantee that a perfect buildtool will solve any of our problems, if we are not trained to take advantage of the tools possibilities.

One much talked about practice that is possible even with todays tools, but sadly is missing from many projects is the concept of Continuous Integration [CI].

The idea is that during the development of a piece of software, the software is built and tested several times per day, integrating code from the developers into the project continuously, making it easy to discover bugs when they are entered into the codebase. By repeatedly building and testing, bugs are caught almost instantly, and can more easily be addressed. This requires some discipline by the developers, so that they will check in their changes often.

If the proper procedures are in place, CI can make the development process faster and more streamlined, resulting in shorter development time, and less resources spent on hunting down bugs.

A XML Schema for PyMekfiles

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

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
        xmlns="http://folk.uio.no/mortenjo/PyMek"
        targetNamespace="http://folk.uio.no/mortenjo/PyMek">

<!-- Last modified $Date: 2005-02-21 20:29:05 +0100 (Mon, 21 Feb 2005) $ -->

        <!-- Simple elements -->
        <xs:element name="name" type="xs:ID"/>
        <xs:element name="filename" type="xs:string"/>
        <xs:element name="MD5" type="xs:string"/>
        <xs:element name="noderef" type="xs:IDREF"/>
    <xs:element name="command" type="xs:string"/>
    <xs:element name="param" type="xs:string"/>

        <!-- Simple with restrictions -->
        <xs:element name="command">
                <xs:simpleType>
                        <xs:restriction base="xs:string">
                                <xs:pattern value="[a-zA-Z0-9+_-]+"/>
                        </xs:restriction>
                </xs:simpleType>
        </xs:element>

        <!-- Complex elements -->
    <xs:element name="task">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="command"/>
                <xs:element ref="param" minOccurs="0" maxOccurs="unbounded"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>

        <xs:element name="tasks">
                <xs:complexType>
                        <xs:sequence>
                                <xs:element ref="task" maxOccurs="unbounded"/>
                        </xs:sequence>
                </xs:complexType>
        </xs:element>

        <xs:element name="children">
                <xs:complexType>
                        <xs:choice maxOccurs="unbounded">
                                <xs:element ref="node"/>
                                <xs:element ref="noderef"/>
                        </xs:choice>
                </xs:complexType>
        </xs:element>

        <xs:element name="node">
                <xs:complexType>
                        <xs:all>
                                <!-- If you don't have a filename, you MUST
                                have a name. If you intend to use a node as
                                a noderef target, you MUST use a name. -->
                                <xs:element ref="name" minOccurs="0"/>
                                <xs:element ref="filename" minOccurs="0"/>
                                <xs:element ref="MD5" minOccurs="0"/>
                                <xs:element ref="children" minOccurs="0"/>
                                <xs:element ref="tasks" minOccurs="0"/>
                        </xs:all>
```

```
            </xs:complexType>
        </xs:element>

        <xs:element name="pymek">
            <xs:complexType>
                <xs:choice maxOccurs="unbounded">
                    <xs:element ref="node"/>
                    <xs:element ref="noderef"/>
                </xs:choice>
            </xs:complexType>
        </xs:element>

</xs:schema>
```

B README-file for PyMek

```
README
======

:Author: Morten Lied Johansen
:Contact: mortenjo@ifi.uio.no
:Date: $Date: 2005-02-22 20:40:24 +0100 (Tue, 22 Feb 2005) $

This is PyMek, a make-like tool for building software projects.

The files included in this distribution are:

README.txt
        This file.
setup.py
        Distutils setupscript.
scripts/pymek.py
        PyMek itself.


PyMek contains the following packages and modules:

PyMek/__init__.py
        PyMek package.
PyMek/config.py
        Configuration module.
PyMek/log.py
        Logging module.
PyMek/exceptions.py
        Exceptions module.
PyMek/build.py
        Build module, deals with actual building.
PyMek/parser.py
        Parser module, deals with parsing PyMekfiles.

PyMek/tasks/__init__.py
        Tasks package, provides baseclass and support.
PyMek/tasks/pymek_tasks.py
        Contains the default set of tasks.


In addition, the following files are distributed as part of PyMek,
but are not a part of the PyMek Python package:

doc/pymekfile.xsd
       XML Schema describing the PyMekfile format.
doc/tutorial.rst
        Tutorial on using PyMek, in reStructuredText format.
doc/reference.rst
        More detailed reference for PyMek and the tasks in rst format.


INSTALLATION
============

PyMek uses distutils, and installing PyMek is as easy as running the following
from your commandline:

        $ python setup.py install

Depending on your wishes, you may run ``setup.py`` with some options, use
``--help`` to see what they are.
```

C Reference

This text is written both as a reference for distribution with PyMek, and as a chapter in the Cand. Scient. thesis that described the creation of PyMek. Some areas of the text may reflect this.

PyMek comes equipped with ten tasks as default. It is relatively easy to add new tasks, as long as you know enough Python to write it.

At the time of writing, some of these tasks could use more work in order to be genuinely platform-independent, but on most typical platforms, they will perform their duty well.

C.1 Creating new tasks

Creating a new task is simply a matter of creating a module for it to live in, and writing a class that subclasses the `Task` class in `PyMek.tasks`. Actually, due to Pythons ingenious "duck-typing", there is no need to subclass, as long as your task behaves like a `Task` object.

`Task` defines a constructor and a single method. The constructor takes four parameters, while the `run` method takes none.

As can be expected, `run` is called in order to execute the task. It relies on the constructor to provide it with details about what it is supposed to do.

The parameters given to `__init__` (the constructor) are as follows:

Parameter	Meaning
`target`	The filename to create.
`param`	A list of parameters given in the PyMekfile.
`deps`	A list of filenames, the dependencies of this target.
`sources`	A list of filenames, the files created by the previous task.

The dependencies and sources deserve a closer explanation. Each task is required to return a list of files created. If the target has multiple tasks, the list of files created by the previous task is passed in to the next task in the sources parameter. If sources is not given, the default constructor will use deps instead.

Each parameter passed in at creation is stored in a variable with the same name, with the addition of an underline at the start. So the `target` parameter is available in the `self._target` variable.

Once you have written your class, you place it in a module and drop it into one of the locations PyMek searches for tasks. If you are not able to use the default directory, PyMek accepts options for new locations to search.

If you created a module `mytasks`, and the task `mytask` in that module, you would refer to it in a PyMekfile by the name `mytasks.mytask`. For the time being, PyMek does not support task packages, so a single module will have to do.

C.2 The default tasks

Here we will give a short introduction to each task, and the parameters it accepts. Parameters can be given on the commandline, in the configuration file, or in the PyMekfile.

C.2.1 Move and Copy

The tasks `move` and `copy` implement platform-independent move and copy operations, using Pythons shutil module. They take no options, and assume only one sourcefile.

C.2.2 reStructuredText

`rst_compile` will take one source, and run a suitable tool from the Python Docutils to create the target given.

In the configfile, or on the commandline, you can define a handler for a specific filetype. The options end with `handler`, and start with the filetype of a target. The value of that option will be used as the command to run in order to create a target of that type. If no handlers are defined matching the filetype, the two default handlers `rst2html.py` and `rst2latex.py`, for html and latex respectively, will be used.

If parameters are present in the PyMekfile, these are prepended with `--` and added as options to the command that is being run.

C.2.3 LaTeX

`latex` will compile a LaTeX file to a DVI file. Any parameters given in the PyMekfile are added to the commandline, without processing. If the option `latex` is present in the configuration, this defines a new name for the `latex`-executable.

C.2.4 dvips and dvipdf

These two tasks are modeled after the LaTeX task, and behave identically, with different executables and optionnames as you would expect.

C.2.5 Java

The Java task will compile a Java source to bytecode. It takes one parameter, `compiler`, which can come from any of the three configuration sources. `compiler` defines which compiler to use. If none given, it will use the first found of `javac`, `jikes` and `gcj` with the `-C` option.

C.2.6 C/C++ compile

`C_compile` will compile C or C++ sources using the distutils `CCompiler` class. It accepts one option, `include_dirs`, a colon-separated list of directories with includefiles for the compiler.

C.2.7 C/C++ link

`C_link` will link C or C++ objects using the distutils `CCompiler` class. It accepts three options: `type`, `library_dirs` and `libraries`.

`type` defines which kind of target is being linked. The valid values are `LIB`, `OBJ` and `EXE` for shared library, shared object and executable, respectively.

`library_dirs` is a colon-separated list of directories the linker should use for finding libraries.

`libraries` is a colon-separated list of libraries to use during linking, in a platform-indepentent form. This usually means that there should be no fileextention, such as `.so` or `.dll`.

C.2.8 Generic Command

For all those cases where you can not accomplish the things you want with the available tasks, and you have no time or inclination to create one, you can probably get it to work with the `GenCommand`.

The parameters are joined together with spaces, and executed. A parameter that matches the text `$target$` will be replaced with the value of `self._target`, while a parameter that matches `$sources$` will be replaced with the sources, separated by spaces again.

C.3 Configuration options

In addition to the options accepted on the commandline, PyMek will accept the same options in a configurationfile. One option is only available in the configurationfile, and that is `taskpaths`. It contains a colon-separated list of directories to search for task-modules. These directories will be searched *after* the default location, so there is no way to override a default task with your own, apart from changing the PyMekfile.

References

[DIP9] Dive into Python, Chapter 9; Mark Pilgrim; http://diveintopython.org

[WJCB] Why Johnny Can't Build; Paul F. Dubois, Thomas Epperly & Gary Kumfert; Computing in Science and Engineering, Volume 5, Issue 5

[CI] Continuous Integration; Martin Fowler & Matthew Foemmel; http://www.martinfowler.com/articles/continuousIntegration.html