Lecture 6: Build Management

Kenneth M. Anderson
Software Methods and Tools
CSCI 3308 - Fall Semester, 2004

Today's Lecture

- Discuss Build Management
- Introduce make

Build Management

- During the implementation phase, the process for constructing a system should be engineered
  - What are the steps to build a system?
    - e.g. what subsystems need to be built before the system can be built?, what libraries are needed?, what resources are required?, etc.
  - Who is authorized to build a system?
    - Small projects: individual programmers
    - Large projects: build managers and/or configuration managers
  - When are system builds performed?
    - e.g. perhaps a system is so large that it can only be built at night when there are enough resources available…

Build Management, continued

- Most modern programming environments have build management capabilities built into them
  - For instance, a Java development environment typically has the notion of a "project" and it can compile all project files in the correct order (and it only compiles files dependent on a change)
- These capabilities free developers from accidental difficulties
  - having to remember the correct compilation order
  - correctly identifying all files dependent on a change

Unix Build Management

- In Unix environments, a common build management tool is “make”
  - Make provides very powerful capabilities via three types of specification styles
    - declarative
    - imperative
    - relational
  - These styles are combined into one specification
    - “the make file”

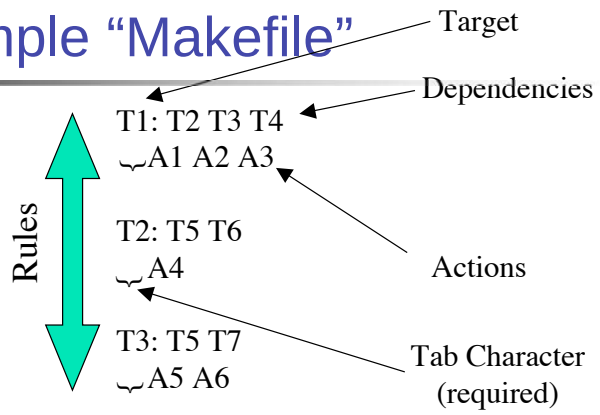
Specification/Modeling Styles

- Operational (or Imperative)
  - Described according to desired actions
  - Usually given in terms of an execution model
- Descriptive (or Declarative)
  - Described according to desired properties
  - Usually given in terms of axioms or algebras
- Structural (or Relational)
  - Described according to desired relationships
  - Usually given in terms of graphs
    - e.g. entity-relationship diagrams

Make Specification Language

- Hybrid Declarative/Imperative/Relational
  - Dependencies are Relational
    - Make specifies dependencies between artifacts
  - Rules are Declarative
    - Make specifies rules for creating new artifacts
  - Actions are Imperative
    - Make specifies actions to carry out rules

Example “Makefile”



If a dependency changes, a rule’s actions are executed to (re)create a rule’s target

More on Make

- Make is well-integrated into a Unix/C environment
  - Primitive Components are Files
  - Actions are “shell commands”
  - Rules are placed in a file and denote the “specification”
    - Rules make explicit the dependencies of the system and what to do about them
- Note: make is not just for source code!

Make, in more detail

- Make can automatically compile source code to produce an application’s executable
  - You could write a shell script to do this…

```
#!/bin/tcsh
g++ -c main.cpp
g++ -c input.cpp
g++ -c output.cpp
g++ main.o input.o output.o -o program
```

  - …however, a shell script will compile every file each time it is run
    - Make is much “smarter”

A second example make file

```
program: main.o input.o output.o
    g++ main.o input.o output.o -o program
main.o: main.cpp defs.h
    g++ -c main.cpp
input.o: input.cpp defs.h
    g++ -c input.cpp
output.o: output.cpp defs.h
    g++ -c output.cpp
```


More on Actions

```
target: dependencies
    actions
```

- target and dependencies are typically files.
- If any dependency is modified more recently than its target then `make` performs the rule’s actions.
- An action can be any shell command, one per line.
  - Each action must begin with a tab.
- Actions typically create the target file from the dependency files.

Examples

- **Given the following directory (higher number == newer)**
  - **main.cpp: 1, main.o: 4,**
  - **input.cpp: 2, defs.h: 3**

Example 1 Makefile

```
main.o: main.cpp defs.h
        g++ -c main.cpp
```

Output of Example 1

```
make: `main.o' is up to date.
```

Example 2 Makefile

```
input.o: input.cpp defs.h
        g++ -c input.cpp
```

Output of Example 2

```
g++ -c input.cpp
```

Example 3 Makefile

```
output.o: output.cpp defs.h
        g++ -c output.cpp
```

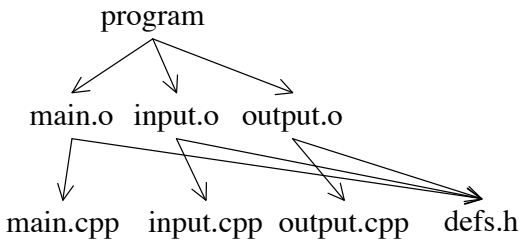
Output of Example 3

```
make: Fatal error: Don't know
how to make target `output.cpp'
```

What would happen if you typed "make" again after example 2?

Make dependency graph

A makefile can be modeled as a dependency graph.
The make algorithm performs a traversal over the graph.
Each node is checked after all of its children, and the actions are run if any child has a timestamp greater than its parent



make command line

- % make
  - make will look for a file called "makefile" or "Makefile"
  - make looks inside the file for its first target
    - the first target is made the goal for this execution
- Different goals can be specified by listing them on the command line and a specific make file can be specified with the -f option
  - make main.o -f program.makefile

More on Actions

- Actions do not have to invoke a compiler
  - they can be any shell command
- Additionally, targets do not have to be files

```
clean:
    rm *.o
```

- Targets like "clean" with no dependencies and no files created in response to their actions are called "phony targets"
  - The actions of a phony target always execute if the phony target becomes the current goal
  - any target that depends on a phony target will always have its actions executed
    - why?

More on phony targets

- Phony targets can be useful for deployment

```
install: ~/csci3308/arch/sun4/bin/program

~/csci3308/arch/sun4/bin/program: program
    cp program ~/csci3308/arch/x86/bin
```

- If you type “make install”, `make` checks to see if the file “program” in the current directory is newer than the one in the `install` directory. If it is, `make` copies (or installs) the new version into the `install` directory

Make “Macros”

- Make has variables known as “macros”
  - They work similar to environment variables
  - Some shell variables are available
    - Such as $(HOME) and $(ARCH)
- We can thus rewrite the previous example

```
INSTALLDIR = ~/csci3308/arch/$(ARCH)/bin

install: $(INSTALLDIR)/program

$(INSTALLDIR)/program: program
    cp program $(INSTALLDIR)
```


Action/Target mismatch

- Actions do not have to create their target
  - when this occurs, you have a mismatch

```
main.o: main.c
    lpr -Pakira ~/.cshrc
```

  - What happens when you type “make”?
    - Assume main.c is newer than main.o
  - What happens if you type “make” again?
- You typically do not want to do this; but make has no way to prevent the creation of this type of rule

Next Week

- More (much more) on Make
  - We’ll also take a look at a more modern type of build system, called ant
  - Lab 3 also provides more detail on make