Reliable software rebuilding

Alan Grosskurth

Software Architecture Group (SWAG)

School of Computer Science

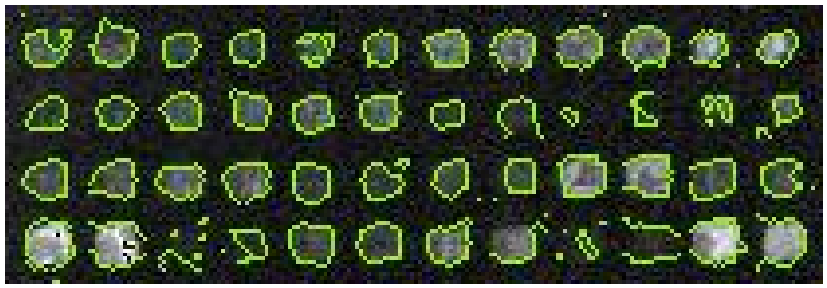
University of Waterloo

agrossku@uwaterloo.ca

2006.04.10

University of Toronto (2000–2004)

- Courses in compilers, OS, formal methods, AI, machine learning
- NSERC scholarship: Ontario Cancer Institute (2002, 2003)
- DNA microarray image analysis software—lives on GNU Savannah



Grad school

```
% phd.m
%
% author: Cecilia
% date: 09/08/05

load THESIS_TOPIC

while (funding==true)
    data = run_experiment(THESIS_TOPIC);
    GOOD_ENOUGH = query(advisor);
    if (data > GOOD_ENOUGH)
        graduate();
        break
    else
        THESIS_TOPIC = new();
        years_in_gradschool += 1;
    end
end
```

University of Waterloo (2004–)

- Courses in text databases, generative programming, software evolution, software architecture
- Studied architecture of Mozilla (2.5 mLOC)
- Published paper at ICSM 2005: *A reference architecture for web browsers*
- Focued interest on build issues

Problem: Building large systems correctly

- Single OS, single language, minimal config options isn't too bad
- Gets difficult when you add more platforms, languages, configuration options
- Mozilla: 4-5 languages, 7-8 OS, 130 config options
- **War story**: I've been told by someone who spent some time at Sun that there was a period of time (a few months) during which they couldn't build Solaris :)

Harder problem: Correct rebuilds

- Small changes shouldn't mean long build times
- Slow turnaround, wasted productivity
- Nobody really trusts the correctness rebuilds—tinderbox, buildbot, etc. typically perform clean builds
- Compiler caches (ccache) help

Typical approach

- Environment configuration
- **Build complete dependency tree**
- Expand macros, etc.
- Recurively build targets top-down

Problem: “implicit dependencies”

- Engineers don’t want to have to explicitly state which headers each object file depends on
- “scanners” can scan source files and output included headers
- Typically run before starting the build
- Problems with generated headers (perhaps created using other targets)
- Due to conditional compilation, it’s most reliable to use the compiler to figure this information—depends on CFLAGS

Problem: configuration information has dependencies

- One solution: hard-code the order of checks
- Cache previous results
- Slow to re-run `configure` if you just want to change one option

A different approach: `redo`

- Conceived by D. J. Bernstein at UIC
- One dependency tree that is constructed dynamically
- “builders”, “configure tests” are nodes like everything else
- Avoid big up-front cost if you only want to rebuild a small part
- Dependencies are *embedded* in the construction commands
- Keeps state, uses suffix matching
- I’m writing a prototype in Bourne shell script

Basic `redo` example

`bar.do` might say:

```
redo-ifchange foo
tr x y < foo
```

Analagous to:

```
bar: foo
	tr x y < foo > bar
```

But as safe as:

```
bar: foo
	tr x y < foo > bar---redoing
	fsync bar---redoing
	mv bar---redoing bar
```

How `redo` works: targets and sources

- When asked to create a file it hasn't heard of before, presume the file is a source if it exists, target otherwise
- For latter, immediately save decision to disk so that subsequent creation of target doesn't change decision

`redo` example: dynamic prerequisites

`default.o.do` might say:

```
redo-ifchange compile "$2.c" "$2.o.deps"
redo-ifchange `cat "$2.o.deps"`
./compile "$2.c" "$3"
```

How `redo` works: prerequisites

- After building a target, save prerequisites in `.redo`
- Next run looks at `.redo` and can quickly figure out whether target is up to date
- For latter, immediately save decision to disk so that subsequent creation of target doesn't change decision

`redo` example: dynamic prerequisites

`compile.do` might say:

```
redo-ifchange warn-auto.sh conf-cc
cat warn-auto.sh
echo exec "`head -1 conf-cc`" \
  '-c "$1" -o "$2"'
chmod 755 $3
```

And `conf-cc` might say:

```
gcc -g -O2 -Isrc
```

`redo` example: dynamic prerequisites

`default.deps.do` might say:

```
redo-ifchange ccdepfind "$2.c"
./ccdepfind "$2.c"
```

And `ccdepfind.do` might say:

```
redo-ifchange warn-auto.sh conf-cc
cat warn-auto.sh
echo exec "`head -1 conf-cc`" \
  '-MM "$1" | cut -d" " -f2-'
chmod 755 $3
```

`redo` example: dynamic prerequisites

`compile` is generated as:

```
#!/bin/sh
# WARNING: This file was auto-generated.
exec gcc -g -O2 -Isrc -c "$1" -o "$2"
```

`ccdepfind` is generated as:

```
#!/bin/sh
# WARNING: This file was auto-generated.
exec gcc -g -O2 -Isrc -MM "$1" \
  | cut -d" " -f2-
```

Generated headers

`uint64.h.do` might say:

```
redo-ifchange choose compile load \
  tryulong64.c uint64.h1 uint64.h2
./choose clr tryulong64 uint64.h1 uint64.h2
```

Based on results of compiling, linking, and running `tryulong`, `uint64.h` becomes either `uint64.h1` or `uint64.h2` (`typedef uint64` is either `long long` or just `long`)

Targets depend on nonexistent files

- Compile a program a program which has a line:

```
#include "vis.h"
```

- But you forget to create `vis.h` in the current directory
- The compiler uses `/usr/include/vis.h` instead
- If you create `vis.h` and rebuild, nothing happens.

Targets depend on nonexistent files

```
redo-ifchange vis.h
```

means current target should be rebuilt if an **existing** file `vis.h` is **modified** (or removed)

```
redo-ifcreate vis.h
```

means current target should be rebuilt if a **nonexistant** file `vis.h` is **created**

- Useful for optional build parameters
- Used internally to find the right build script, e.g. `file.o.do` vs. `default.o.do`

Location of derived artifacts

- Also known as *Objdir* or *VPATH*
- May be possible without special support need from `redo`
- Choices:
  - Start in separate build directory, specify sources using *srcdir*
  - Start in source directory, specify targets using *targetdir*

Location of build scripts

- `redo-ifchange` looks for in the same directory as `target` for the build script `target.do`
- Useful to store build scripts in an different directory so they can be reused by different products
- `REDO_SCRIPTS` environment variable (planned)

Limitations

- Cycles aren't be detected until midway into the build
- Separate processes for `redo-ifchange` results in more overhead than separate threads
- Not clear how to allow creation of multiple targets from single build script
- Quoting can be tricky
- Bourne shell doesn't make Windows happy... Python?

Build system testing

- Build tool vs. build scripts
- `redo-ifchange` and `redo-ifcreate` can be tested individually
- Build scripts can be tested in a scaled-down sandbox

Conclusions

- Composable mechanisms can be used to trigger appropriate rebuilding when code and build scripts are changed
- Need to implement a build system for a large scale product (mLOC) to get practical data on ease of maintainability and scalability

Acknowledgements

- D. J. Bernstein for conceiving `redo`:
  `http://cr.yp.to/redo.html`

Questions?