Makr

A build tool for
C/C++ development on Linux written in Ruby

Motivation

Build Tool Requirements

- Out-of-source builds
- Build variants separated
- Easy, intuitive interface
- Programmable and extendable in standard scripting language (preselection: ruby (or python))
- Flexibility (small number of predefined concepts)
- Most basic: build only changed parts (what is change?)
- Centered around C/C++-development (on Linux)

Existing Tools

Make

- Old-school, quirky syntax (tab-spacing,etc.)

CMake

- Widely adopted
- Generates native build files (but not runnable w/o cmake!)
- Proprietary scripting language (syntax could be better)

Scons

- Python-based
- Complicated way of out-of-source-builds, copies files
- Slow

Existing Tools – 2

Waf

- Python-based
- Out-of-source-builds, copies files
- Variant builds could be easier, a lot of predefined concepts/targets (configure, build, clean...)

Rake

- Ruby-based
- Not tailored to C++/C-builds (same as Ant)

(see https://github.com/ewuenf/Makr/wiki/Comparison-with-other-build-systems from time to time for an updated list of comparison)

Basic concepts - 1

- Ruby as base language, variable scope can be chosen (not everything is global)
- Out-of-source variant build directories (./Debug/, ./Release/, etc...), variants are completely user-defined
- no standard arguments, user-defined and -parsed (typically variant and target, see example files)
- Doubly-linked Dependency-Directed-Acyclic-Graph (DAG) of Tasks
- Hierarchical configurations using conventions

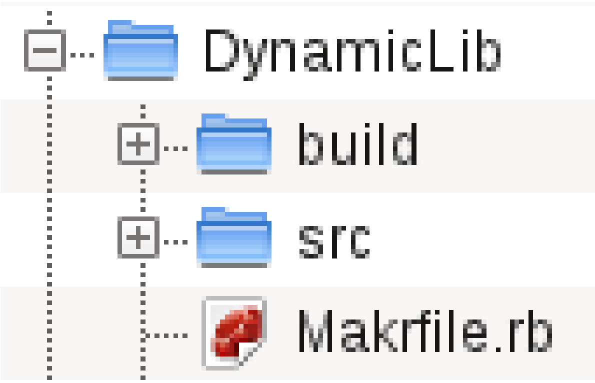
Basic concepts - 2

- Central Build class
  - Manages Tasks and Configs
  - marshalled to/from disk in the out-of-source variant build directory
- Multi-threaded build
- flat output into build directory, no copying of source files
- Build error handling (target deletion, see extra slide)
- Automatic adaption to source files added/removed
- Extensions (plugins) possible (very simple in ruby)
- Reduced basic functionality (main source: ~1500 LOC)

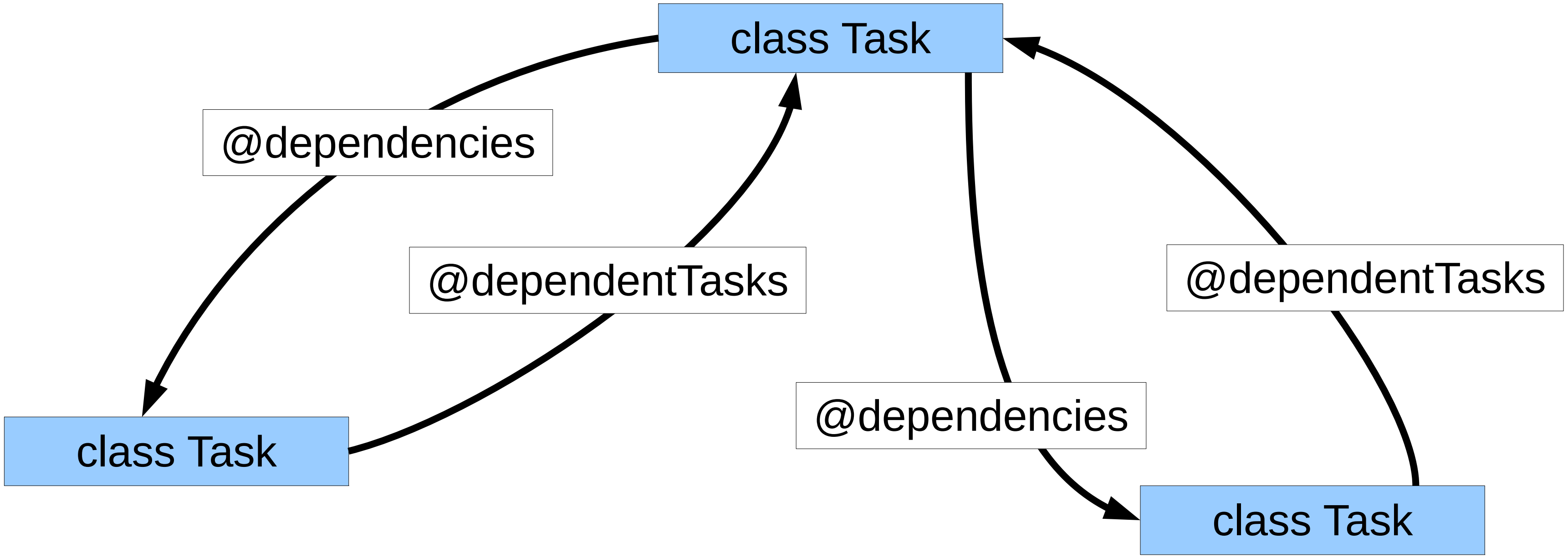
Basic usage

Setup

- All you need is a ruby script called **Makrfile.rb** in your current directory
- Just run **makr.rb** and give optional arguments
  - typically two arguments are used:
    - first argument: the build variant directory
    - second argument: the target to build (like “all”, “clean”, “configure”, a single file, whatever you want)
- Arguments are “parsed” by user in Makrfile.rb



Task DAG



A task represents a build step (checking for file changes, compiling, etc), the DAG represents their dependencies with respect to each other

Every instance of Task has:

- A unique name
- An array of tasks it depends on (dependencies)
- An array of tasks that depend on it (dependentTasks)

Purpose of dependentTasks is to go up the graph upon build after finding the leaves

Task – other Attributes

class Task
@config:Config @state:String @targets: Array of Strings

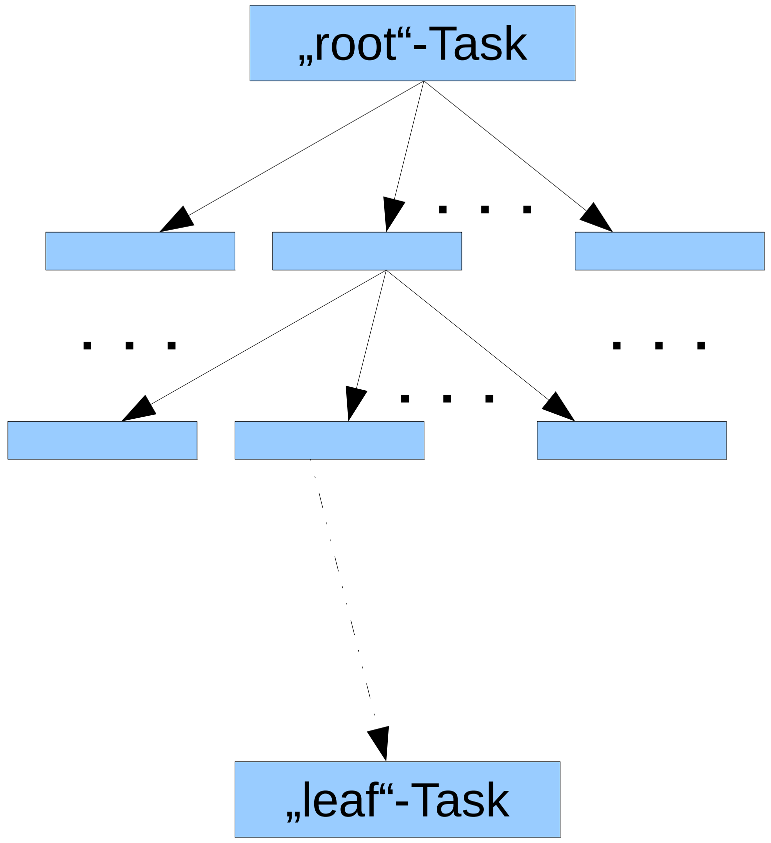
class Config

- hash-like interface (key-value pairs)
- typical (convention-based) usage: *config[„compiler.cFlags“] = „* -*fexceptions* -*Wall “*
- has a parent and/or childs
- asking for a key returns
  - associated value of internal hash or
  - if key is not found, the value of the parent for the key (or parents parent, or p …)

The @state-member of a Task is used to determine if an update is necessary. It must be filled by Task instances upon update (see next slides).

@targets describe the target files the Task produces, if any. Default behavior is to delete them before update (in case an update is necessary).

Building a task



Begin: call Build.build with a root task (or build.defaultTask is used)

- First stage: calls preUpdate() on **all** tasks of the build in a single thread
- Second stage:
  - Descends recursively from the root task to the leaves (tasks with no deps) and marks visited tasks in a single thread
  - Walks up towards root in parallel respecting task dependencies and updating marked tasks that need an update because dependencies changed (as **leaf nodes** have no deps, they **are always updated**)
  - error handling explained later
- Third stage: calls postUpdate() on all tasks that have been updated in a single thread

DAG and update()

DAG modification, why?

- @dependencies change due to changing #include-statements in C/C++-source
- ...other reasons?

DAG modification by tasks, when?

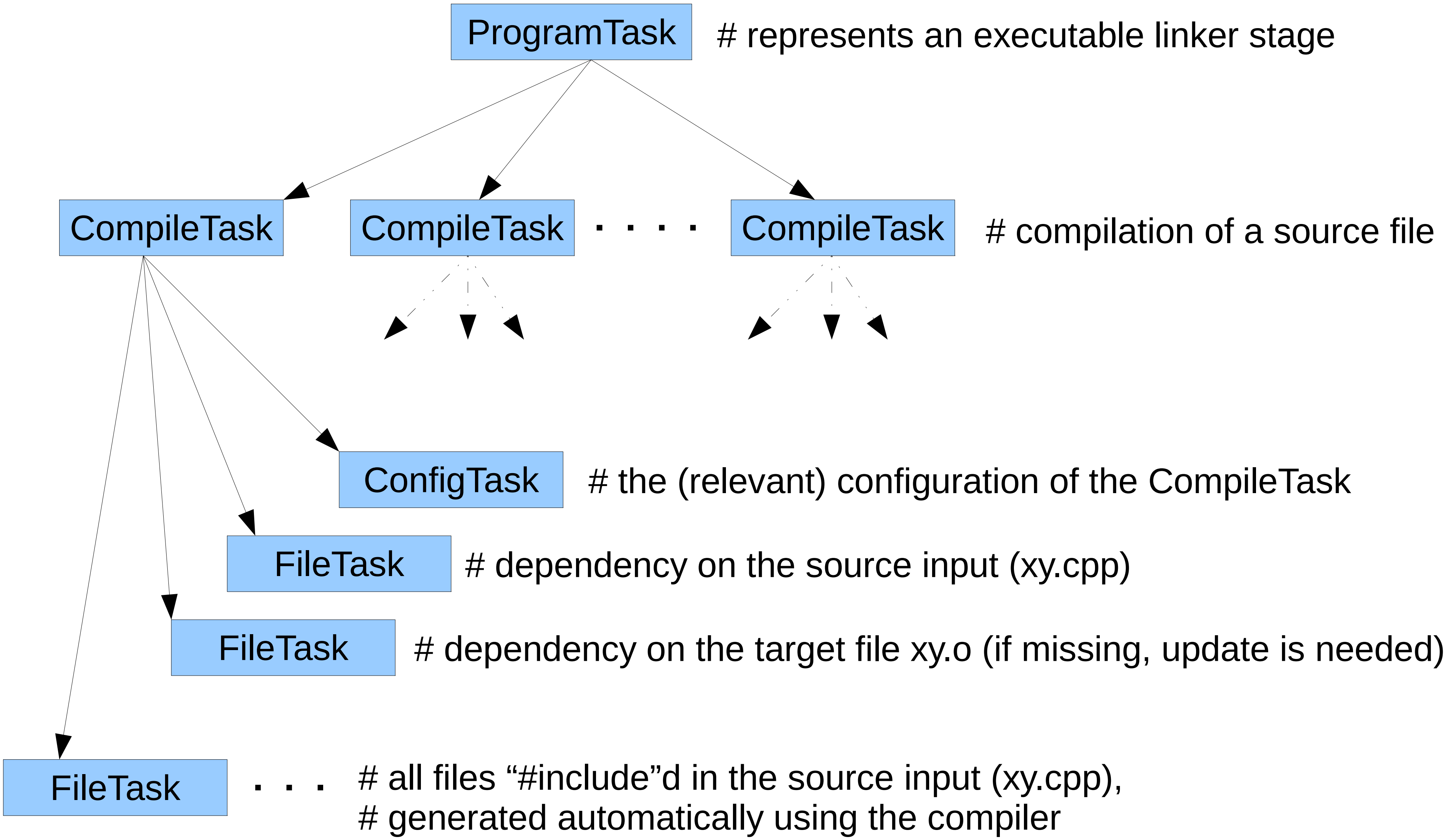
- During preUpdate()-Phase: **OK**
  - called on a list of **all** tasks without using DAG in a single thread
- During update(): **NO**
  - called by multi-threaded UpdateTraverser class, **walks DAG**
- During postUpdate(): **OK**
  - called on Tasks that updated with success (@state != nil) in a single thread from a list in update order w/o using DAG

Target deletion

Build failure behavior or what to delete upon error?

- for sure all dependent targets up to the currently selected root task in the DAG (which may only be a single object file a single file compilation)
- deleting the dependant targets up to the root of the DAG is enabled by default for safety concerns but is an option settable by the user

Typical Task DAG



ProgramTask # represents an executable linker stage

CompileTask CompileTask . . . . CompileTask # compilation of a source file

ConfigTask # the (relevant) configuration of the CompileTask

FileTask # dependency on the source input (xy.cpp)

FileTask # dependency on the target file xy.o (if missing, update is needed)

FileTask . . . # all files “#include”d in the source input (xy.cpp),
# generated automatically using the compiler

What is change?

Change relevant to a build system can be:

- File attributes like size or modification time
  - “make” notion of time order flawed, time **change** matters
- File content (hash sums like md5 or sha-1)
  - used in several modern build systems, more costly than mtime
- Configuration changes (additional compiler flags etc.)
- ...much more !/?

Generalization / Implementation

- A change is a change of the @state of a Task
- @state is represented by a String
- @state of inner nodes of the DAG is typically the concatenation of the @state of the @dependencies

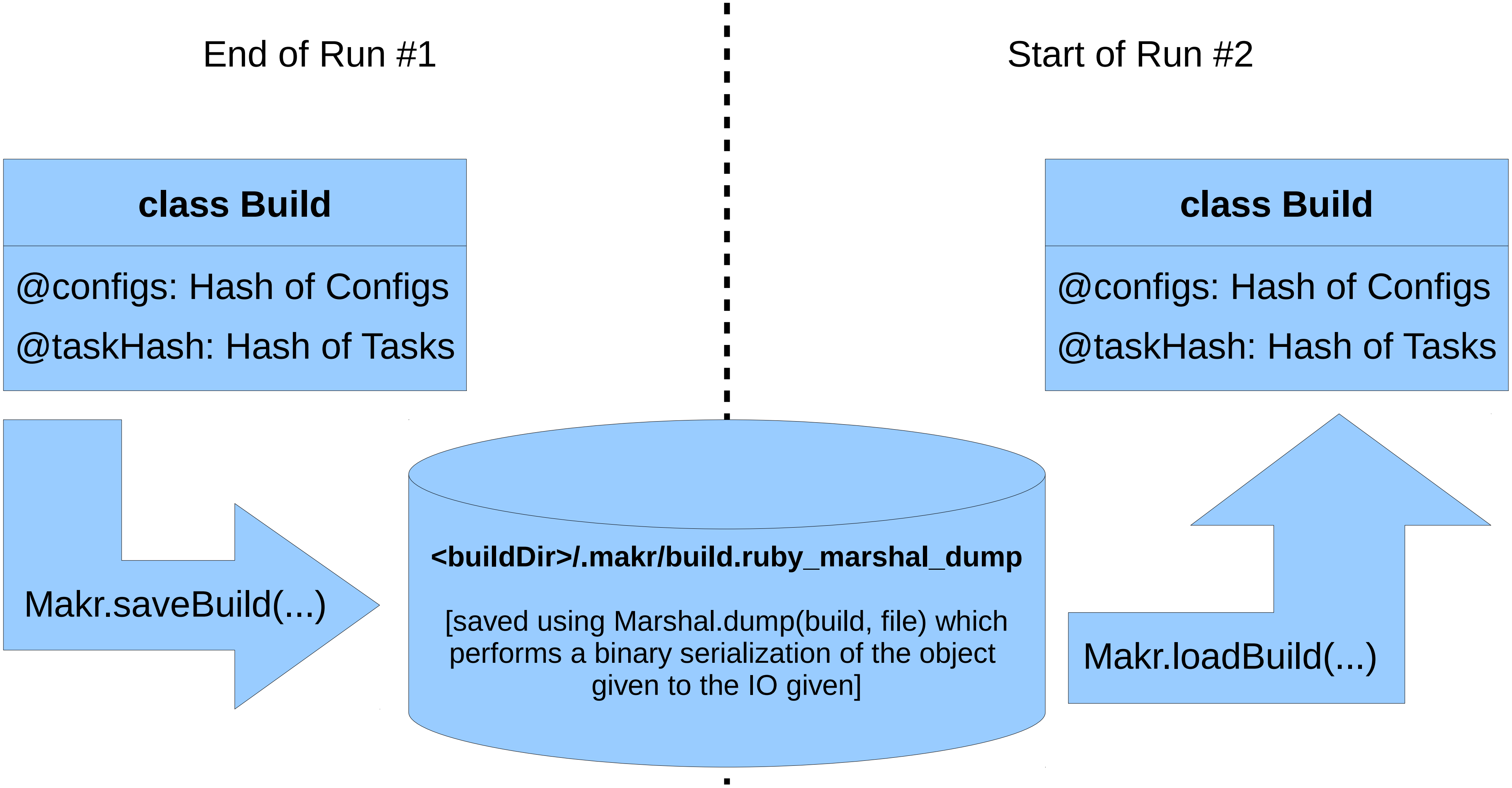
How is change detected?

Answer: basically by “leaf”-Task-classes:

- class FileTask
  - represents a file (like a header file, source file, binary target, generated source file, etc...)
  - size- and access-time-based change detection *OR*
  - hash-based file change detection (md5 currently)
- class ConfigTask
  - detects changes in the concatenated configuration strings of the @config-member of a Task

Disk cache

Change detection between tool runs:



End of Run #1

Start of Run #2

class Build

@configs: Hash of Configs

@taskHash: Hash of Tasks

class Build

@configs: Hash of Configs

@taskHash: Hash of Tasks

Makr.saveBuild(...)

<buildDir>/.makr/build.ruby_marshal_dump

[saved using Marshal.dump(build, file) which performs a binary serialization of the object given to the IO given]

Makr.loadBuild(...)

Change is (in source):

```
class Task
  ...
  def needsUpdate()

      # @state = nil indicates initial build or error in previous builds
      return true if not @state

      # although not called for leaf nodes, we keep this condition for safety
      return true if dependencies.empty?

      # if one of our deps has an update error, it does not make sense to update
      return false if not concatStateOfDependencies()

      # this is the central change detection
      return true if (@state != concatStateOfDependencies())

      # otherwise nothing changed and we dont need to update
      return false

  end
  ...
end
```

@state in update()

```
class MySpecialTask < Task
 ...
 def update()

   @state = nil # first set state to unsuccessful build, which the nil-value indicates

   doSomething()

   # indicate successful update by setting state string to preliminary
   # concatenated string (set finally in postUpdate()) to propagate change
   @state = concatStateOfDependencies() if successful

 end
 …
 # default implementation from Task
 def postUpdate()
   @state = concatStateOfDependencies() if @state and (not @dependencies.empty?)
 end
 ...
end
```

@state is set again in postUpdate(), because DAG structure can change in postUpdate() and thus the @state of @dependencies

errors during update()

Indication

- when compiler errors etc occur: @state == nil

Propagation

- nil-@state is propagated to @dependentTasks upon DAG-Traversal

Handling

- user decision:
  - abort calling update() on tasks upon first error or
  - going on with all tasks that can update (= no dependency had an error)

errors during update() - 2

Discussion: Error-Propagation, yes or no?

- Yes
  - users expect the dependent targets to be deleted upon error in build process (like the resulting binary)
  - if targets get deleted, tasks need to be build next time
- No
  - If the user fixes the error by **reverting** the erroneous change, the next build will only rebuild the erroneous target
    - reduced build time
    - requires file-hashing for change detection (increases build time slightly)
- Approach taken here: “Yes”

Source files collection

class FileCollector

- collects files (recursively) from a directory
- can be given patterns like “*.cpp” for inclusion and exclusion of files
- captures added files automatically

Removed source files

- are deleted from Task DAG by Build class automatically upon load
- their @dependencies and @dependentTasks are deleted recursively too upon load

Extensions

Extensions, why ?

- Keep main source clean and short

How?

- ruby source files in extensions-directory
- loaded upon user request: *loadExtension("name")*
- ruby source is loaded and executed and typically introduces new methods and classes or modifies existing classes in module Makr
- ruby makes extension writing easy and fun (see extension "SourceStats")

The command line

Makr makes no assumptions

- The user is free to define his own meaning and processing of command line arguments
- ruby classes such as OptionParser could be used

Argument stack is provided

- If sub-directories with own Makrfile.rb are build, arguments are pushed on a stack, popped after return
  - arguments can be added
  - callee arguments are not tainted
- See class ScriptArguments/ScriptArgumentsStorage

Makrfile.rb example

Go through a Makrfile.rb from the examples dir?

subdir calls

Subdirs containing their own stand-alone Makrfile.rb

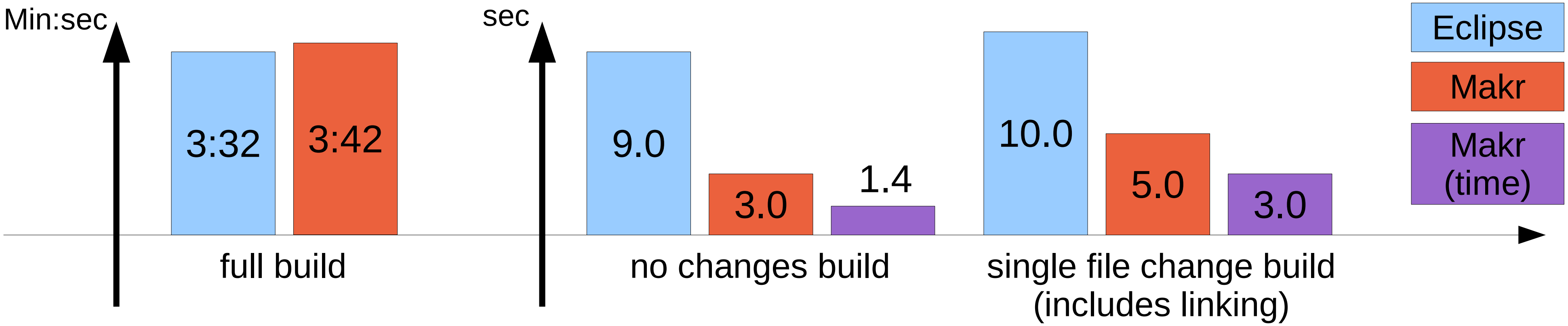
- can be called and build from another Makrfile.rb
  - use Makr.makeDir(dir, additionalArguments)
  - dir contains the subdir to “makr”
  - additionalArguments is an optional argument that contains an array of arguments (like ARGV)
- The classes ScriptArguments/ScriptArgumentsStorage provide the stack functionality for arguments to subdir-scripts for independent recursion
- subdir-call arguments can be constructed very individually by each higher-level Makrfile.rb → flexibility!

Performance

Setup

- Eclipse with managed build vs. Eclipse with Makr using file attributes for change detection
- Timing initial build, incremental build w/o changes
- Machine: Dual-Core Athlon 2 GHz, 2G RAM
- Makr also measured on command line using “time”

Results (788 source files, ca. 4.5 MB total)



Performance – 2

Interpretation

- eclipse generates dependency files for each processed source during compilation
  - this speeds up first compilation
  - Makr, in comparison does two compiler calls, one for dependency generation and one for compilation on the files, as gcc wont output deps during compilation if not to a file (kind of gcc limitation maybe a pipe possible?)
- all the dependency files for each source are loaded by make upon later builds, this generates the long loading time

Would you like to know more ?

[insert copyrighted image here]

Read the source, its short and fairly well documented ;-)