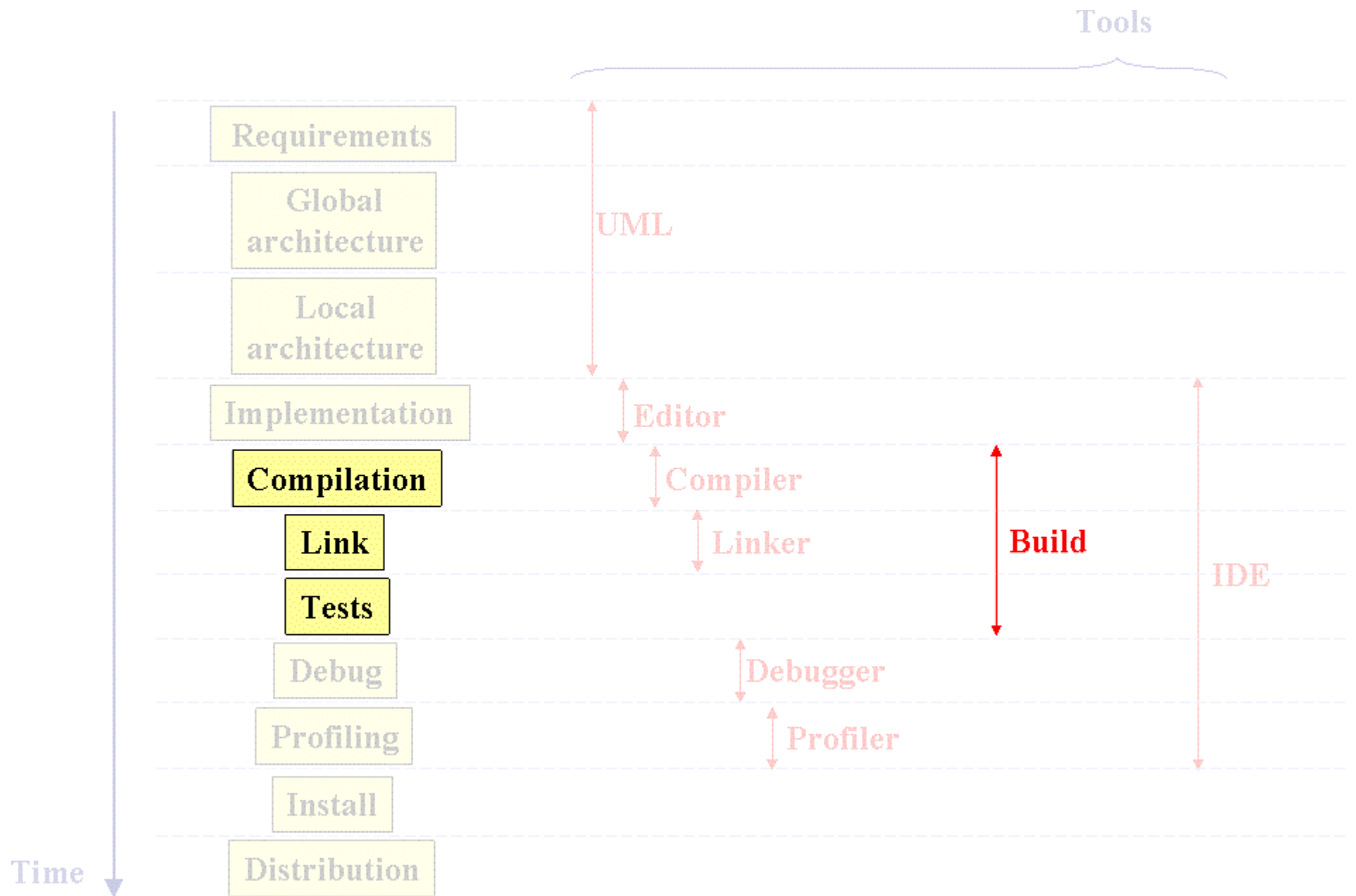# Building Software

Mathieu Lacage - DREAM

Outline

- Understand the compilation workflow

- What is painful about it ?

- How can we automate it ?

Compilation

Typical compiled languages (C,C+,Java) require lots of intermediate steps before execution:

- Preprocessing
- Parsing
- Code generation
- Object file generation
- Linking
- Loading

INRIA
SOPHIA ANTIPOLIS

Basic Definitions

- *Source and header text files* : foo.h, bar.cc

- *Object file* : binary file which contains the compiled version of a source file.

- *Library* : a collection of object files stored in a single binary file. Used to package independent "components".

- *Executable* : a single binary file which can be loaded and executed on a system

- *Process* : a version of an executable running on a given system
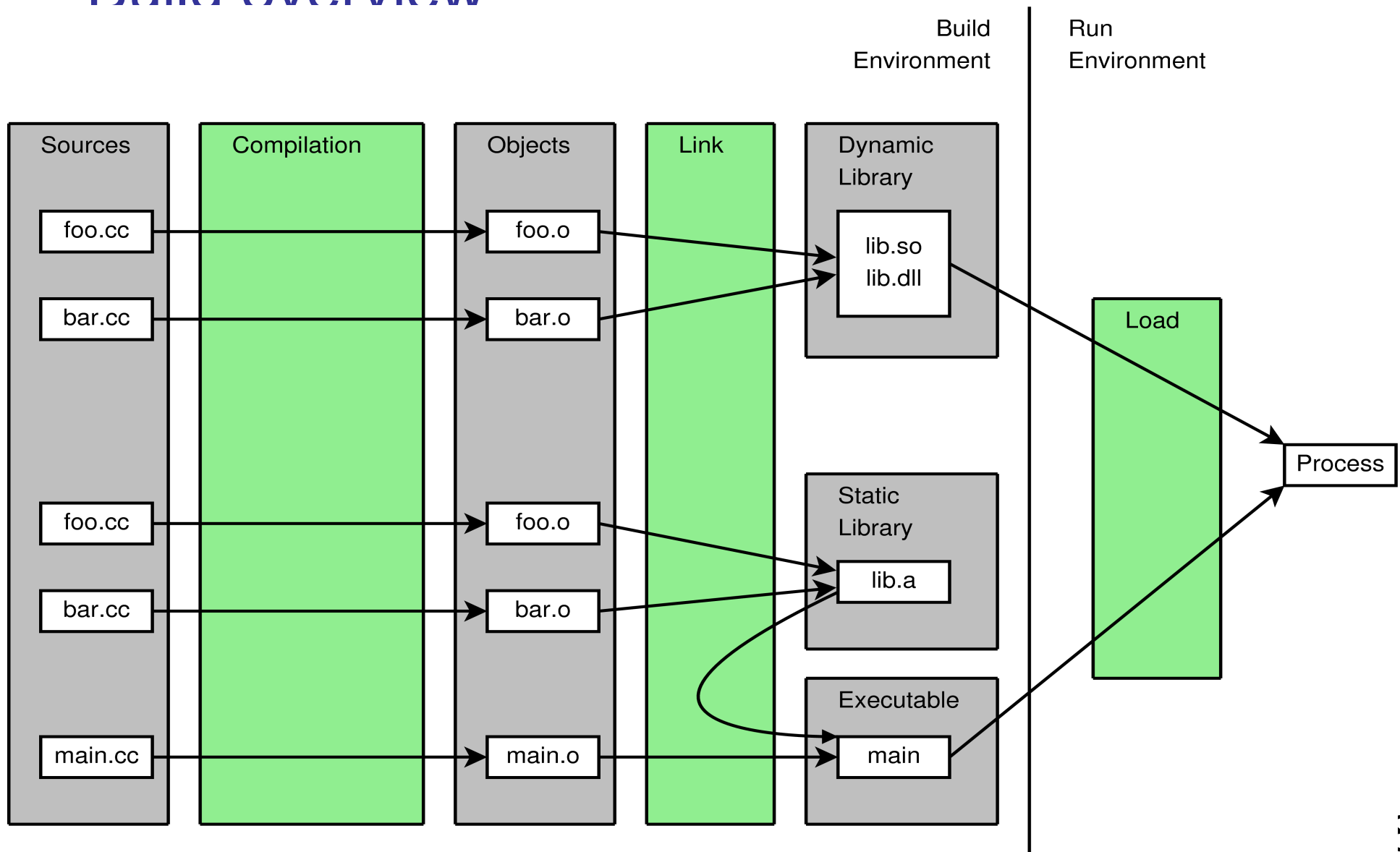
5

INRIA
SOPHIA ANTIPOLIS

Tool Definitions

- *Compiler* : the tool used to generate an object file from a source file.

- *Linker* : the tool used to generate an executable or a library from a set of object files.

- *Loader* : the tool used to create a working Process from an executable and a set of dynamic libraries.

6

INRIA
SOPHIA ANTIPOLIS

Library Definitions

- Executable X uses library Y

- *Static library* :
  - During executable link : copy the object files needed into the final executable.
  - Ignored during executable loading.

- *Dynamic library* :
  - During executable link : record a dependency into the final executable.
  - During executable load : lookup the dynamic library and load it in memory before starting execution

I N R I A
SOPHIA ANTIPOLIS

Build overview

Build Environment

Run Environment

Sources

Compilation

Objects

Link

Dynamic Library

Static Library

Executable

Load

Process

foo.cc

bar.cc

foo.cc

bar.cc

main.cc

foo.o

bar.o

foo.o

bar.o

main.o

lib.so lib.dll

lib.a

main

INRIA
SOPHIA ANTIPOLIS

What is hard about a build ?

- Using libraries correctly and efficiently : this is very low-level and platform-specific.

- Speed: do not rebuild everything when you change one small file

- Dependencies : ensure a correct build

INRIA
SOPHIA ANTIPOLIS

Dependency tracking

- What is a dependency ?
    - foo.cc includes foo.h and qt.h
    - bar.java uses package com.sun.something
    - libgtk-2.0.so depends on libX11.so
    - test-binary depends on libmy.so and libgtk-2.0.so
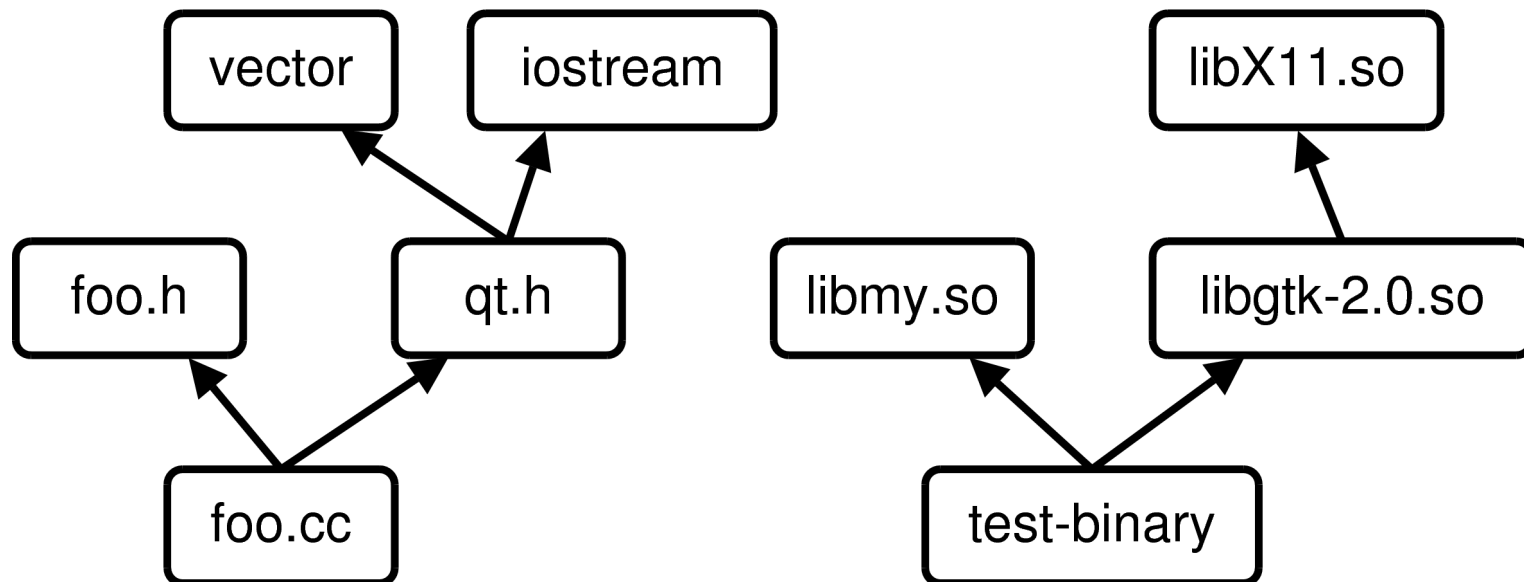
A real-life example

- A moderate-sized project, ns-3 :
    - All builds are correct
    - All builds are repeatable
    - Very accurate dependency tracking
    - Build is automatic

- Uses Scons but every other build tool works the same way.

```
[mathieu@mathieu ns-3-dev]$ scons
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Building targets ...
g++ -g3 -Wall -Werror -fPIC -DRUN_SELF_TESTS
-DNS3_DEBUG_ENABLE -DNS3_ASSERT_ENABLE -Ibuild-dir/dbg-
shared/include -c -o build-dir/dbg-
shared/src/simulator/high-precision-128.o
src/simulator/high-precision-128.cc
...
```

INRIA
SOPHIA ANTIPOLIS

A simple example

- Platform : Linux

- Tools :
    - Gcc
    - GNU make

- Files : main.cc, a.cc, a.h, ... c.cc, c.h

- a.cc includes a.h and c.h

- b.cc includes b.h

- c.cc includes c.h

- main.cc includes a.h, b.h and c.h

- If a header changes, only the files which include it should be recompiled

Make Principles

- *Makefile:* describe the configuration of the project
  - Dependency tree: prerequisites -> target
  - Commands to generate a target from a set of prerequisites

- *Target* : a file to generate

- *Prerequisites*: a set of files which are needed to generate a target

- *Command* : a command to run to create a target once all the prerequisites are available

- Syntax : target is built by command from prerequisites req1, req2, req3, and req4. [tab] is a real ascii tab character:

```
target: req1 req2 req3 req4
[tab]command
```

INRIA
SOPHIA ANTIPOLIS

Make on our example

```
# a.o is generated from a.cc. a.o depends on a.h, c.h

a.o: a.cc a.h c.h

b.o: b.cc b.h

c.o: c.cc c.h

main.o: main.cc a.h b.h c.h

# main executable depends on main.o, a.o, b.o, and c.o

main: main.o a.o b.o c.o
```

- By default, make knows the command to use for every target based on its extension

- Make keeps track of up-to-date files with file timestamps

INRIA
SOPHIA ANTIPOLIS

Running our example

```
[mlacage@garfield seminar-build]$ make main

g++    -c -o main.o main.cc

g++    -c -o a.o a.cc

g++    -c -o b.o b.cc

g++    -c -o c.o c.cc

cc   main.o a.o b.o c.o   -o main

[mlacage@garfield seminar-build]$ touch c.h

[mlacage@garfield seminar-build]$ make main

g++    -c -o main.o main.cc

g++    -c -o a.o a.cc

g++    -c -o c.o c.cc

cc   main.o a.o b.o c.o   -o main

[mlacage@garfield seminar-build]$
```

INRIA
SOPHIA ANTIPOLIS

A small bug

- In the previous example, we can see a small bug: make links our c++ program with the 'cc' command rather than the 'g++' command. Easy to fix:

```
main: main.o a.o b.o c.o

        g++ -o $@ $^
```

- We override the default rule with our own: '$@' and '$^' are two variables which identify the name of the target and the list of prerequisites respectively.

INRIA
SOPHIA ANTIPOLIS

Other Build Tools

- Choice depends on :
  - Operating Systems targeted
  - Programming Language used
  - Libraries used?
  - ...

- For example :
  - *Autotools*
  - *Ant*
  - *Cmake*
  - *Scons*
  - *...*

Build tool summary

	Languages	Platforms	Documentation	Learning curve
Make	All	All	Good	Average
Automake Autoconf Libtool	C,C++	Unix, cygwin	Good	Bad
Cmake	C,C++	Unix, win32	Good	Good
Qmake	C,C++	Unix, win32, osx	Good	Good
Ant	Java	All	Good	Good
Scons	C,C++	Unix, win32, cygwin, osx	Good	Good

Conclusions

- If you use libraries, you must learn how they work on your platform:
    - On linux, http://www-sop.inria.fr/dream/intro-devel-env.html
    - "Linkers and Loaders" by John Levine, http://www.iecc.com/linker/

- For small projects, using GNU make is very easy:
    - http://www.gnu.org/software/make/manual/
    - "Managing projects with GNU Make", by Robert Mecklenburg