Build System Plan v1.1

Jonas Larsen
ESO/SDD/ALMA

March 22, 2010

Contents

1 Introduction

Developers have found that using CASA's build system, carried over from the AIPS++ project, can be anything from easy over cumbersome to impossible at worst. The purpose of this memo is to identify what the issues are,

and to describe how these issues are solved.

The direct goal of reworking the build system is to reduce the time that developers spend on building CASA.

As an indirect effect, it can be hoped that a more reliable and reproducible build process will have a positive impact on the perceived lack of stability experienced by developers, testers and end-users.

Section 2 describes the requirements of an improved build system, section 3 discusses build tools, and section 4 outlines the new build system.

2 Requirements

I have solicited internal and external CASA developers about what the current problems are, and how the build experience can be improved. In no particular order the requirements/requests/wishes/recommendations were as follows (some of the requirements may already be satisfied):

R1. The build system must work portably on all development platforms. The build system should be portable. It should be possible to develop CASA on the current supported development platforms (for the purpose of this memo taken to be RHEL 4.x, RHEL 5.x in 32bit and 64-bit variants, and OS X 10.5 and 10.6) and on likely future platforms.

R2. Better documented how to install CASA from scratch. It should be more accurately documented how someone installs the prerequisite packages, checks out the source code and compiles the code, on all supported development platforms. We often document the procedure as evidenced by

https://safe.nrao.edu/wiki/bin/view/Software/CASAInstallFromSource

*https://safe.nrao.edu/wiki/bin/view/Software/CASACoreMigration

https://safe.nrao.edu/wiki/bin/view/Software/CASABuildInstructions

https://safe.nrao.edu/wiki/bin/view/Software/CASADevelopmentBuildCycle

https://safe.nrao.edu/wiki/bin/view/Software/DevelopingCasaMac

https://safe.nrao.edu/wiki/bin/view/Software/BuildingWithoutMacPorts

*https://svn.cv.nrao.edu/casa/osx_distro/developer.notes

(marked with * the ones I think are most up to date), but we are not very good at maintaning this documentation, which has a tendency of being too sketchy and always more or less out of date. While email support is always available, it takes much head-scratching for external as well as internal developers to create a development build.

R3. Flexible and configurable locations of dependency packages. This is for developers who wish to experiment with different versions of dependency packages. It should be possible to configure the location of packages and have more than one version of an external package installed at the same time. It is currently not possible (or easy) for example with python which is distributed as RPM (3rd party tar ball on Mac) and is installed in a non-configurable place.

R4. Possible to verify external dependencies before building. This is to reduce the time that developers spend on configuring their system, where the current work cycle (sneeze + yum install + sneeze + yum install + ... ad lib) can be very tedious and non-obvious in relating error messages to missing packages and/or a wrong makedefs. The verification step should include a check that the version numbers of external packages are appropriate for CASA.[1]

R5. Support for debug builds. Apart from being able to build production code, the build system should support creating debug builds (turning on the -g compiler flag for GCC and define appropriate preprocessor flags -DAIPS_DEBUG -DCASA_DEBUG (TBC)).

R6. Documentation. It must be possible to build the documentation from the source (the make targets docsys and docscan). This should not be part of compiling the code, but as optional operations.

[1]This is the functionality of install/configure, except the script has drifted away from the code.

R7. Use widespread technologies rather than unknown and unproven technologies. All other things being equal, the advantages of using a widespread technology include

- Being more in use implies being more tested implies being more likely to work better (more mature),
- Documentation is widely available (e.g, for developers who want to understand the implications of a specific error message, or for build maintainers who can consult external experts on how to best implement a given feature),
- There is possibility of knowledge transfer for developers who work(ed) with other projects than CASA,
- Maintenance will be easier, because the CASA team does not carry the maintenance burden (for example, porting to a future platform).

R8. A single build command must build completely and correctly. Developers should not spend time and tedious efforts on figuring out which sequence of commands produce a correct build. The build system mechanics should rebuild exactly the files that are necessary, no more and no less. The current build system does not meet this goal,

- It is often necessary to clean up object files, in order to force a rebuilding (e.g. to avoid unresolved symbols in libraries). This often occurs in connection with generated code.
- When removing or adding a header file, dependency information may break and must be remade somehow.
- Sometimes the build will hang and needs to be stopped manually (for example when updatelib was killed by CTRL-C, or if the build trips on a file starting with #-character (likely to happen for Emacs users)).
- It is sometimes necessary to remove files containing ":" (such as casapy log files) from a source directory, in order to avoid build failures.
- Generated code is a multiple step process that requires editing/copying a generated header file, although there are no fundamental reasons why generated code cannot be treated like any other intermediate build products.

- It is sometimes (e.g. for linking casapy to dbus on RH4) necessary to define $LD_LIBRARY_PATH, in order to avoid linking errors at build-time and run-time.

While most of these issues are simple for a developer to work around after debugging what the problem was, they add unnecessary complexity to the build process, and resulting libraries and executables may depend on the sequence of steps which the developer chose.

R9. Developers should use the same method for building. Currently, some developers use SCons to build casacore, others use GNU make. On some occasions, this has lead to incompatibilities between different developers' builds. It has caused incompatibilities between Linux and Mac release builds (where Mac releases are created using SCons, and Linux releases are created using GNU make).

R10. It should be possible to work (edit/build/test) in a single subdirectory. Because it can be inefficient if the entire project needs to be recompiled (and dependencies regenerated) when a developer makes small changes in a module. Also, a developer may want to break client code temporarily and integrate a module's changes with the full build just once.

R11. Flexibile location of data used for unit tests. The build system should not impose any particular directory structure for unit test data. I think this requirement may already be satisfied with the current build systems used.

R12. Faster build times. The build system should support parallel builds. That parallel builds (invoked with `make -j`) do not work in the current build system is another symptom that the makefiles do not accurately describe the true dependency graph. Additionally, incremental builds and null builds should be faster by avoiding to redundantly recompile what is already up to date. Parallel builds, which process subdirectories sequentially, are possible (and under-documented) by defining `PARALLEL_MAKE := $(MAKE) -j <n>` in makedefs.

R13. Support for out-of-source builds. It should be possible to put build products in a directory separate from the source code. Advantages of out-of-source builds are that it is easier to identify added source files (thus harder to forget to check them in), it is possible to have several builds based on the same source (e.g. production and debug builds, or builds using different compiler or library versions). This is not possible in the current build system.

R14. The development and release builds should follow the same directory structure. Also, header files and relevant libraries should be included in release distributions. This is to make it easier to switch between environments, and to allow using a binary distribution for (some limited kinds of) developments. Exported header files should be namespace protected, e.g. by placing them in `<prefix>/include/casa/*`.

R15. Increase the unit test coverage. The purpose is to catch more problems earlier.

R16. Modular unit testing framework. It should be quick to run a single unit test. It should be possible to hook up a debugger on the test executables. This seems to be impossible with the current build system, where invoking "make runtests" from a directory will always build, then run, then remove all of the test programs as a monolithic operation.

R17. It should be easy to identify which version of CASA is running. CASA's so-called build number, shown as part of the startup message, closely follows but is different from the source code revision number. This makes it difficult for developers and testers (including developers and non-developers) to identify which code changes have made it into a given build. While the code version number is not shown on startup, it is available at run-time as `casa['source']['revision']`.

R18. Do not create new active/stable builds at AOC if certain baseline tests fail. Exactly which tests are to be considered baseline tests for each branch is TBD.

R19. Easier to synchronize non-committed changes. Developers may sometimes have several local builds, and want to share local, uncommitted changes between such builds. The problem with checking in things under development is that it might step on other unrelated developments (or be stepped upon by unrelated developments). As a solution for developers making such bigger chunks of developments, I would suggest (a currently non-existing practice) to create a "private", temporary branch starting from the active branch. When developments have finished, the branch is merged/copied back to the active branch, and deleted.

R20. Do not delete intermediate build products. Because debugging with gdb works better on Mac, when intermediate object files exist.

This concludes the list of requirements that I received, and I agree with all of these. The only of my own wishes for a better build system, which was not already listed is

R21. Verify the version number of external libraries at runtime Because even if external library version numbers are verified during the build process, the casapy executable may still accidentally pick up a different version at runtime. The consequences of this range from nothing to surprising and difficult to understand behaviour. If the compile time and runtime version numbers differ, a warning should be shown. Also, to ease debugging, it should be possible to display the version numbers of all external packages from within a casapy session.

2.1 Discussion

Many of the requirements (R2, R4, R8, R9, R17, R21) are related to the same issue: reproducability and lack thereof. When two different persons use the same version of the code, they should get the same behaviour. If that is not so, time is being wasted on disagreeing with the behaviour ("It works for me."). The problems with unreproducible behaviour include

- If a developer cannot reproduce a reported problem, it is difficult/impossible to solve.

- Conversely, seeing a local problem in module XYZ (for example a compilation error or a runtime crash) can be a major obstacle that has to be debugged locally, because the expert on module XYZ could not reproduce the problem.
- If the build is not reproducible, it is easier to break ("It worked for me when I checked it in!").
- After having seen much transient, unreproducible, "flaky" behaviour which turned out to be a build issue not reproducible elsewhere, it is a reasonable, mental response to start ignoring any such behaviour. It makes us blind to the real problems.

The need for human intervention in the build process is not just a consumption of FTEs; it makes the build process less reproducible and reliable, because human beings are much worse than computer programs when it comes to repeating a sequence of operations, and keeping track of what they did.

3 Build technologies

I have looked into the following technologies (read documentation, made experiments, asked around). This is a short (subjective) overview of the different tools available. For more discussion, pro et contra, see also [1] and [2].

GNU make GNU make has the advantages that makefiles are simple to get started with and everyone knows them. GNU make allows but does not provide things like dependency tracking, portable building of executables and shared and static libraries and handling generated code; this is to be implemented by the user of GNU make. While GNU make is ideal for simple projects with a few files, it takes much skill and diligence to implement well-working makefiles for larger projects. The makefiles tend to contain much low-level code and mechanics, compared to the higher level alternatives. See also [4].

AIPS++ makefiles The Current Build System (from now on CBS) adds a layer on top of GNU make, thereby avoiding the complexity of make at the application level. The mechanics is implemented in the generic code/install/makefile.*

which define the rules for building various kind of code (C++, XML, ...). At the highest level, the module makefiles are typically very short and concise. The configuration of the build (compilation options and location of external packages) is defined in the pivotal makedefs.

ACSMakefiles The build system used for building ALMA on-line software is designed similar to the CBS with a core of makefiles that implement the mechanics, and are included by the application specific makefiles. Features include support for generating Java, C++ and Python interfaces from CORBA IDL.

GNU autotools The GNU autotools (autoconf/automake/libtool) were created as a general purpose build tool which addresses some of the problems with GNU make. They raise the abstraction level compared to raw make, by providing support for dependency generation and building executables and static/shared libraries for many languages (C++, C, Fortran, ...), portably on many UNIX-platforms, and comes with a built-in unit testing framework. A configure script and makefiles are generated from configure.ac and Makefile.am specifications. Short Makefile.am's produce portable and correct makefiles with many features (which would be a major undertaking to do by hand using GNU make). The autotools are very widespread.

CMake CMake (Cross Platform Make) is a makefile generator, a high-level, portable build system, similar in spirit to the GNU autotools. A difference is that CMake supports other "back-ends" than GNU make, including Eclipse CDT and XCode on Mac. The back-end makefiles[2] are generated from CMakeLists.txt specifications.

Ant Ant derives from a Java tradition, but has support for CASA relevant languages, too. It is implemented in Java, and the makefiles are written in XML.

SCons SCons is a high-level build system. Its makefile language is python, which has the advantage that there is no new language to learn if you already

[2] "makefile" is used in the generic sense, referring to the configuration filesof a given build tool, without implying a particular technology.

know python. SCons uses checksums to tell if object files are up to date. This is safer than file time stamps (on non-exact file systems), and potentially more efficient: For example, if a comment is changed in a C++ source file, the corresponding object file gets regenerated with a new time stamp but the same checksum. Unlike make-based systems, SCons will know that it is not necessary to relink the library or executable that depends on that object file. Unlike GNU autotools and CMake, SCons does not have separate configuration and build steps, both of which are contained in the SConstruct files.

Boost.Build The Boost.Build system inherits from Jam and was designed to meet the specific requirements of the Boost C++ library, but Boost.Build can be applied to non-Boost applications as well. It specializes in very portable building of C++ code, including non-UNIX platforms (where the otherwise very portable GNU autotools fail). It has built-in support for C++, C and Fortran, and toolsets for other languages can be added. It is currently (since 2007) under refactoring to use python as its makefile language (following the example set by SCons).

qmake qmake is a makefile generator, tailored to support the development of Qt projects but is extensible to support other languages as well.

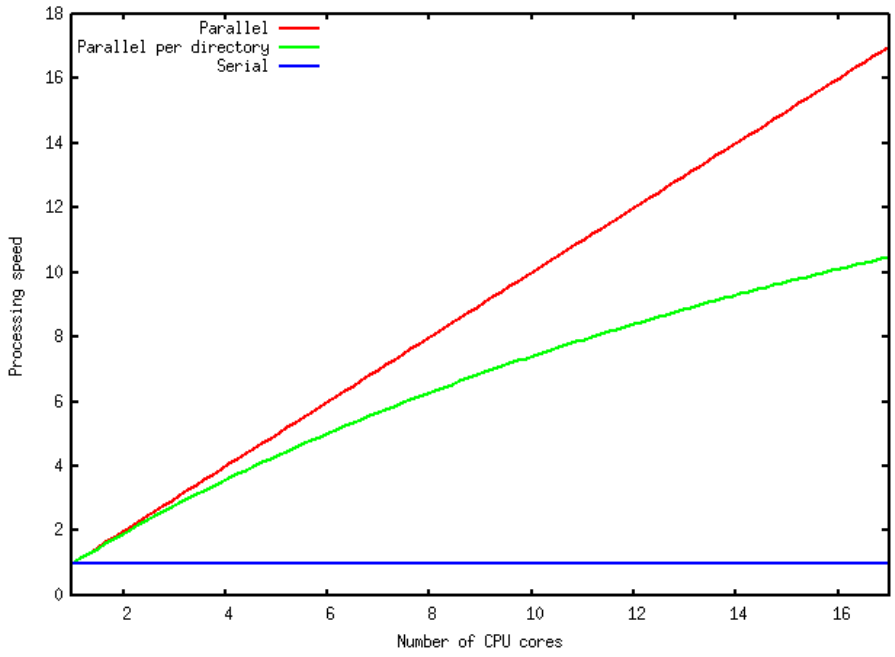
3.1 Parallel builds

As touched upon in the classical [3], recursive make does not play well with parallel building because it sets up artificial boundaries between directories. GNU autotools-based implementations usuallly deal with this[3] by processing subdirectories one at a time (i.e., sequentially) in a user-given, well-defined order which respects dependencies, but the contents of each directory is processed in parallel. For example, in the extreme case with one file per directory, a per-directory parallel build is just a serial build. Other build technologies (CMake, SCons, ...) support true parallel builds by having a global view on dependencies, where directory boundaries is a non-issue.

Here an estimate of the cost of per-directory parallelism compared to true parallelism (back of the envelope, all other things being equal): For a project

[3] although non-recursive Makefile.am's are possible.

with M source files distributed over D directories, a per-directory parallel build with N parallel build threads will get a "penalty" per directory boundary (because less than N processors are in use), which is equivalent to extra $(N-1)/2$ source files in a truly parallel build. Hence the overall processing time is proportional to $(M + D(N-1)/2)/N$, where a truly parallel build takes time M/N (making the assumptions that source files take equally long to compile, and are randomly distributed modulo N in subdirectories). The figure shows the speed (inverse time) of the two types of parallel builds, compared to a sequential build, when plugging in numbers relevant for CASA's C++ code, $M = 1646$, $D = 128$. Conclusion: Parallel builds are much faster than serial builds (which is hardly a surprise), and there is not much difference between the two types of parallel builds at least up to 8 CPU cores.



3.2 Discussion

I received only few recommendations about particular technologies, which were to prefer GNU make. More frequent was "something widely used rather than something less widely used". The most frequent view was "whatever works".

CBS + configure. An approach to solving R4 is to add a configuration step to the CBS. This could be achieved by updating/reimplementing the existing install/configure script, or (more in line with R7) by using autoconf to generate a correct makedefs file[4]. This would solve a big problem. It would not solve the problem of having a build mechanism which does not honor the dependencies between files, takes manual intervention to use (R8), and is slow compared to alternatives (R12). If I could solve these file dependency problems in the CBS (which is not a given fact), we would still be using a not widely-used technology (the AIPS++ makefiles), which has proved for some time to be difficult to maintain and fix. Solving the (less critical) issues R13 and R16 would probably require major rewrites. The CBS is demonstrating how difficult it is to implement a robust, efficient and correct build system using GNU make. In the years after the AIPS++ makefiles were written, immense amounts of thought and man-power has went into solving the highly non-trivial problem of building software. It is better economy to reuse this knowledge and products, instead of trying to do something ourselves, which will never be half as good.

No technology is a silver bullet. The success of any implementation depends on how well the makefiles are written. The essential difference between the CBS and the alternatives is the difference between using raw make and suffering from it, compared to reusing a higher level toolkit closer to our problem domain.

A good reason to stay with the CBS is to avoid the risk of switching technologies mid-run. Much of that risk can be eliminated by pre-emptive planning and testing.

Most of the technologies considered have automatic dependency tracking, support unit testing, support generated code, built-in support for C, C++, Fortran and Python, extensible to support other languages, are portable, support parallel builds. I think the open issues R4, R7, R8, R12, R13, R16,

[4]Note that while automake cannot be used without autoconf, WCSLIB is an example of autoconf being used without automake

R20 are solvable using any of the general-purpose tools.

ACSMakefiles do not seem ideal for CASA's needs. They are not widespread. Portability is not a major concern for ALMA software which must run on a standard machine. It has no configuration step, the location of external libraries are hard-coded in application level makefiles. Being based on raw make, it has a few quirks, does not always rebuild the code correctly, for example developers sometimes must clean up the $INTROOT installation directory and install from scratch (although speaking as someone who has tried both build systems, I have found it much more robust than the CBS).

Being very wide-spread and a proven technology, the **GNU autotools** is an obvious candidate to consider. A common criticism is its convoluted tool chain involving Makefile.am/sh/m4/Makefile languages, which can make autotools based implementations difficult to understand and debug, and to do "simple" things like adding a new dependency library.

Compared to the autotools, **CMake** is conceptually simpler; it uses just a single CMakeLists.txt macro language, and makefiles are generated only from these files. It supports true parallelism and has good performance[5]. It is newer and less wide-spread than the autotools, for example flex/bison support was added only as of November 2009.

That **SCons** is already in use by CASACore, is based on the same scripting language as CASA, and that prototype work was done already [6] makes it relevant to consider for CASA non-core (in addition to being a modern, general purpose build tool). SCons may have a disadvantage of appearing more alien to typical CASA developers, compared to a make-based build system; on the up-side it does not suffer from the problems of make ([4]). While python is an interesting and modern choice for a makefile language, I am personally not convinced that it is an ideal makefile language, or any better than a tailored, domain-specific language. SConstruct files implementation and syntax tends to be more verbose than the alternatives. Moreover, the great power of a general purpose programming language might tempt the makefile implementer to "hack away" and implement whatever does the job, where other build systems that provide only a minimum set of functionality, will force the user to pick the solution that was intended for the particular job. SCons has a reputation of being slow. According to CASACore's main-

tainers SCons has no good support for "install" targets.

I experimented with building shared libraries and executables using **Boost.Build**. The makefile syntax, recycled from Jam, has a few quirks (something about spaces before semi-colons). After studying the available documentation for some time, it was not clear to me how to detect external libraries with Boost.Build (which may be a limitation of the build system, or on its documentation, or on my understanding, but in any case a concern). It is my impression that Boost.Build is more a "portable make" than a full build system.

Ant was not initially designed to be a general purpose build system, and uses verbose XML as its makefile syntax. It integrates well with CruiseControl. The AntContrib C++ extension does not seem to be widely used for C++ projects. **qmake** was not initially designed to be a general purpose build system. I have not investigated these technologies further.

Comparing the various technologies with the list of requirements for CASA, I find that CMake provides most closely what we need (followed by GNU autotools and possibly SCons), also taking into account ease of implementation, ease of use and future maintenance. CMake is very similar to the autotools, but slightly better performing according to benchmarks (I think, mostly due to better caching of configuration results), and with a simpler tool chain. CMake's main disadvantage is that it is less widespread than the autotools, but it is used by large non-trivial projects with requirements on portability, MySQL and KDE, so lack of maturity does not seem to be a real concern. Emacs and Eclipse have support for CMake. I have found that adapting built-in CMake macros is simpler than doing the same with autotools. CMake ships with a number of package finders macros (equivalent to the autoconf macro archive), which appears to be somewhat of a random "zoo". If a certain macro (such as FindBLAS, which does the job of detecting the BLAS library, then sets up internal build variables) does not match exactly CASA's needs, it can be easily extracted from CMake's internals and tweaked as desired.

4 Solution

The following actions are meant to solve all of the open requirements. This is based on adopting CMake as build technology (but most of what follows is rather independent of a particular technology). The actions are listed in order of implementation. Each action is labeled critical or non-critical as a function of the requirement(s) it addresses. Table 1 gives an overview of requirements and actions.

Table 1: List of requirements, their importance (according to no well-defined metric), whether each requirement is already satisfied, and the action(s) solving the open requirements.

	Requirement	Priority	Satisfied		Action
R1	Portability	critical	yes	(A3)	
R2	Documented build procedure	critical	no	A5	Monitor builds
R3	Dependency packages flexible location	non-critical	yes/no[a]	A8	Distribute external packages as source
R4	Verify external packages	critical	no	A4	CMake configuration
R5	Debug builds	critical	yes	(A3)	
R6	Documentation build	non-critical	yes	(A3)	
R7	Widely used technology	critical	no[b]	A3	CMake build system
R8	Simple/correct builds	critical	no	A3	CMake build system
R9	One build method	non-critical	no	A1	Use SCons
R10	Work per module	non-critical	yes	(A1, A3)	
R11	Unit test data	non-critical	yes	(A3)	
R12	Faster builds	critical	no	A3	CMake build system,
				A11	Clean up code tree
R13	Out-of-source builds	non-critical	no	(A3)	
R14	Release structure	non-critical	no	A7	Revisit release contents
R15	Better unit test coverage	non-critical	no	A9	Implement unit tests
R16	Modular UT framework	non-critical	no	A3	CMake build system
R17	Identify version	non-critical	no	A2	Redefine build number
R18	Build only good versions	critical	no	A6	Build server tests
R19	Synchronize uncommitted changes	non-critical	yes		
R20	Keep intermediate object files	non-critical	no	A3	CMake build system
R21	External package runtime version numbers	non-critical	no	A10	Runtime version number checks

[a]Yes: because the package location can be configured in makedefs. No: because some packages are distributed as RPMs (Linux) and 3rd-party tar ball (Mac).

[b]The current build system is built upon GNU make which is certainly widespread. The infrastructure of included makefiles from code/install/makefile.* is not widely used.

4.1 Actions

A1 (non-critical) Use one tool for compiling CASACore. I do not have strong feelings about which build tool (GNU make or SCons) to use, but we CASA developers should try to be consistent if we want consistent behaviour. To the extent that it does the job, I would suggest using SCons because it is used by CASACore's maintainers, so the maintenance is easier, and we are more likely to get the same behaviour as the authors intended. This is an action on developers and on how Linux/Mac distributions are created. It addresses R9. Of course, developers can always go their own ways, but in that case they should be aware of the risk of breaking things for other developers who use the standard build tool.

A2 (non-critical) Avoid confusion between the build number and code revision number. The build number should be either (TBD) redefined to be identical to the code revision number, or it should follow a very different numbering scheme (such as 1, 2, 3, ... or A, B, C, ...) in order to make it clearly different from the code revision number. This action solves R17.

A3 (critical) Implement CMake build system for CASA non-core. The CMakeLists.txt files must contain the correct dependencies. Parallel builds must work. It must be possible to build documentation for CASA. This will solve R7, R8, R12, R16, R20 (and not break R1, R5, R6, R10, R11, R13). CASACore's build system will not be touched.

A4 (critical) Implement configuration step. Checks should be made for external library locations and version numbers. External paths should be configurable. This will solve R4.

A5 (critical) Monitor the builds. Automatic daily/nightly/continous builds is one of the software engineering best practises. We currently have automatic builds (that I am aware of) in Socorro and in Garching. Garching results are published at http://www.eso.org/~dpetry/nightly-sneeze.log, AOC results are available for developers with NRAO accounts who know where to look. They run once per day, on machines with a history as such may not always be reproducible. Mac builds are not monitored. The results

of building earlier revisions are not available.

A build server will be set up to monitor the active and stable branches by checking out the latest updates and do an incremental build. Results will be published for all supported platforms. If hardware resources allow, every revision of the code will be built[5]. Some form of notification will be enabled (exact form TBD), so that introduced compilation errors can be resolved as soon as possible. This action will ensure that our code is more often in a compilable state. In addition, developers will be able to distinguish local build issues from global ones.

Less often (TBD) and in addition to incremental builds, complete builds from scratch will happen. Repeating a full build on the same machine is problematic because later builds may interact with previous installations (after you install CASA once, the machine is "tainted" forever). For maximum reproducibility, the builds from scratch will reuse a virtual machine snapshot of a virgin machine (such a prototype setup exists already using Hudson and VMware. A side-effect of automatic builds from scratch is a script which also serves to document the exact sequence of build steps, on every supported platform. The installation documentation must be extracted from this script. Obsolete build information that has accreated must be deleted. This solves R2. The documentation must include how to build and run the unit and regression test suites.

A6 (critical) Run test suite as part of monitoring the build. After doing a full build, the build server will execute the subset of regression tests that define if a given version of the active/stable branches are "good enough" for dispatching. If the acceptance tests passed, the given version of the source code will get an "approved"-stamp. Implementation details are TBD, but possibly by defining SVN tags which always point to the latest "good" revisions. The build machine at AOC would then check out and build these particular tags. This action solves R18.

[5]Given CASA's recent history with around 10 new revisions per day and three supported platforms, this is feasible as long as the time for an incremental build is less than $24h/(3 \times 10) \approx 1h$.

A7 (non-critical) Revisit release contents. Releases should include header files and have same directory structure as the development environment, which will solve R14. An option is to use CPack (which is part of CMake).

A8 (non-critical) Document how to install all packages from source, so that developers do not require RPM / tar ball. The action is to identify which external packages we modify, and document how to build and install them. A non-RPM build chain (which does not install pre-compiled packages but compiles everything from the sources) could be exercised by the build monitoring system. This would solve R3.

A9 (non-critical) Implement more unit tests. The unit tests should be monitored as part of the automatic builds. Unit tests could be included in the acceptance test suites for the active/stable branches (see A6). This will solve R15.

A10 (non-critical) Implement run-time checks of external package version numbers. A warning (not error) must be displayed, if it is not a supported version number. It should be possible to list external libraries versions. This solves R21.

A11 (non-critical) Clean up source code tree. It is my experience that our code tree carries a significant amount of dead-weight, in form of deprecated code that should no longer be used, and "new" prototype code which was never adopted. In the context of building, less code implies faster builds (R12). There might be better reasons (maintenance) for cleaning up. It is not obvious how to implement this action.

This concludes the list of proposed actions. Note that the following topics are not covered:

- Except that the build system must have better support for unit testing (R16), the details of which tests to be done at which stages (for example when declaring an active/stable branches version "good enough") is left open. (It is a parallel, ongoing effort to look into testing in general.)

- The details of setting up a development environment (except R2 that it must be better documented). I think the current scheme of distributing external packages as RPMs etc. works well enough, as long as CASA's build system itself is sufficiently robust (R4).
- There are no requirements or actions related to ASAP.

4.2 Implementation

This section describes how the proposed actions can be implemented.

A1 and A2 can be implemented right away. The new major developments A3, A4 and A5, will be developed on a branch starting from a "good" reference version of the stable branch. The following (internal) milestones are foreseen

- Create CMakeLists.txt for building casapy and other executables using hard-coded paths to external packages,
- Create CMakeLists.txt for building CASA's documentation,
- Implement configuration step (A4)
- Same unit test[6] results compared to the CBS on all platforms,
- Same regression test results as the CBS on all platforms,
- Create release distributions and verify the regression tests using the distros.
- Verify that GUIs work in releases on all platforms.

The automatic build server (A5) will be setup in parallel with these steps, and will also serve to test the new vs. old build systems (which would be a pain to do manually).

After these steps are complete (using a frozen version of the code), the new build system will be merged back to the stable branch and re-synchronized with any changes to the build configuration that happened inbetween. The

[6]This refers to the existing C++ unit tests.

relevant tests (unit, regression and GUI) will be reverified post-merge. The new build system will be copied to the active branch (where tests cannot be reverified). It is very desirable if the new build system can be implemented without changing the source code in a backwards incompatible way, so that the two build systems can live side by side on the same SVN branch[7]

4.2.1 Roll-out

Dispatching a new build system is a delicate operation that needs to be thought about well in advance.

Until this point, all developments and testing could have involved just one person (although I might ask for help for creating distros and for testing GUIs). The next step is to manufacture the relevant developer's documentation, and ask volounteers to try out the new build system. This testing/evaluation by other developers should cover geographical sites and platforms. Once no more problems are found, the new build system is declared operational, and everyone starts to use it.

Beginning of operation will have a period (maybe 1-2 weeks) where both build systems live in parallel (and as such need to be maintained in parallel). After no critical problems are found in this period the switch is done.

In my opinion, we should aim to not have a lengthy limbo period with two parallel build systems, because that will just be a(nother) source of confusion and unreproducible behaviour. Rather, it should be defined upfront what are the criterions for acceptance, and when these criterions are fullfilled, do the switch.

Of the remaining actions, only A6 depends on A5 having been already implemented. The non-critical A7 to A11 can be implemented anytime, not necessarily by the author of this document and independent of a new build system.

[7]Post-CMake-implementation note: It was possible to make the CMake build system backwards compatible to the source code. But at the cost of adding a bit of complexity to the build system (which will be simplified, when CBS must be no longer supported by the source code).

4.3 Maintenance

Any build system needs maintenance. I foresee no significant changes compared to currently.

Unlike now, the makefiles (CMakeLists.txt) will mention explicitly every source file that is part of the build. Therefore, whenever a source file is added or removed, the CMakeLists.txt must be maintained accordingly. CMake supports globbing (picking up anything that looks like a source file); it is an implementation decision whether to use file globbing in makefiles, or list files explicitly. After a change to a CMakeLists.txt, a run of `cmake` is automatically triggered as a dependency of `make` (when GNU make is the backend). It is a disadvantage of the globbing behaviour that, after a source file has been added or removed, the user would need to remember to invoke `cmake` in order to get a correct build (thus not satisfying R8, that the build should be driven by a single, well-defined command). See [7] for more discussion on this topic.

When a dependency on an external package changes, the appropriate CMakeLists.txt must be changed accordingly. For example, if a change is made to CASACore that is not backwards-compatible to previous versions of CASA non-core, the version number of CASACore must be incremented[8], and CASA non-core's version dependency must be incremented accordingly.

4.4 Usage

The starting point for CMake documentation is www.cmake.org. In short, CMake itself is installed by unpacking the distribution, then

```
> ./bootstrap && make && sudo make install
```

where the bootstrap script accepts a –prefix parameter if you want to install CMake to somewhere else than to /usr/local.

Then to compile CASA non-core, do

```
> mkdir code/build
```

[8]CASACore does not already support such fine-grained version numbers, and that feature will be added.

```
> cd code/build
> cmake ..
> make
```

This will create an out-of-source build (where the name and location of the build directory is up to the developer). In-source builds are possible, if you do not care about putting build products in the source code tree.

4.4.1 Per-module development

In order to develop in a single source tree directory (R10), build from the corresponding subdirectory in the build tree

```
> <edit> code/sub/directory/source.cc
> cd code/build/sub/directory
> make
```

4.4.2 Configuring the build

Options to configure the build can be given on the command line when invoking CMake, for example:

```
> cmake -DCMAKE_BUILD_TYPE=Debug ..
```

Notice that a change of CMake configuration parameters causes existing object code to be considered out of date (which is similar to how SCons work, and different from GNU autotools' behaviour). In order to build different parts of the system with different compilation options, one can do:

```
> cd build
> cmake -DCMAKE_BUILD_TYPE=Debug ..
> make synthesis
> cmake -DCMAKE_BUILD_TYPE=Release ..
> make calibration
```

which will build the module `synthesis` with debugging information enabled, and `calibration` with optimizations enabled.

As an alternative, build options can be set interactively by using a curses interface for CMake called ccmake:

```
> ccmake ..
```

CMake supports backends other than GNU make, e.g. Xcode on Mac:

```
> cmake -G Xcode ..
```

Note, that it is not required to edit any checked in files, in order to configure the build.

4.5 Possible issues

The CBS uses subdirectory names to determine, how code in a given subdirectory should be handled, e.g.

```
code/<module>/implementation
code/<module>/apps
code/<module>/fortran
```

for C++ libraries, executables and Fortran code, respectively. Symbolic links are defined from the directory

```
code/include/
```

to each

```
code/<module>/implementation
```

which allows source code #include directives to not mention explicitly the implementation directory. It may be necessary to follow in the footsteps of CASACore and rename the implementation directory to

```
code/<module>/<module>
```

(or create a directory link with this name to the implementation directory. It would be backwards compatible with the CBS.)

Post-CMake-implementation note: It was possible to overcome this issue by making the CMake build system do the same as CBS (which is to create symbolic links at configuration time).

References

[1] SCons Wiki: *SConsVsOtherBuildTools*, http://www.scons.org/wiki/SconsVsOtherBuildTools[9]

[9] All hyperlinks as of 2010.01.04.

[2] Adrian Neagu: *Make alternatives*, http://freshmeat.net/articles/make-alternatives

[3] Peter Miller: *Recursive Make Considered Harmful*, http://miller.emu.id.au/pmiller/books/rmch

[4] Adrian Neagu: *What is Wrong with Make?*, http://freshmeat.net/articles/what-is-wrong-with-make

[5] Johan Boulé: *Benchmarks of various C++ build tools*, http://psycle.svn.sourceforge.net/viewvc/psycle/branches/bohan/wonderbuild/benchmarks/time.xml

[6] Bojan Nikolic: *Building CASA*, http://www.mrao.cam.ac.uk/~bn204/alma/sweng/casabuild.html

[7] GNU automake manual: *Why doesn't Automake support wildcards?*, http://sources.redhat.com/automake/automake.html#Wildcards