Acrobatomatic Build System - generic make files and companion scripts

AUTHOR	Sebastien DEVAUX

ABSTRACT	ABS (Acrobatomatic Build System) is a set of tool mainly made of make files to support software development. It enables complete software project technical management: chain compilation step for build binaries and distribuable package, fetch dependencies package, run tests, support continuous integration automated builds, and generate documentation.

KEYWORDS	build, compilation, document generation, unit test, software package
CONTEXT	Component abs-3.3.6
PROCESSED	2022-01-01 16:00:41+01:00 / sdevaux@ummon

ISSUE / REVISION STATUS RECORD		
Edition	**Date**	**Changes summary**
3	2022-01-01	Minor layout improvement using the upgraded heml and tex styling feature. Completed with few misstypo fixes.
2	2020-03-03	Update for ABS-3.0 release: misstypo and various small error fixes.
1	2019-01-01	First public issue of document

S u m m a r y

List of Figures

List of Tables

Listings

1 Introduction

1.1 Scope

Acrobatomatic Build System (ABS) is a generic build and packaging system for software. It supports many language and many targets. To be compatible, a project shall match the file and directory organization ABS expects. ABS is mainly made of a set of makefiles (GNU make to be used to run those makefiles). It is completed by some common commands available from any (near) posix like system (including cygwin and mingw) to handle some advance features such as downloading dependencies, running tests, packaging and publishing builds. In some way ABS is similar to the maven/ant couple, but has been first designed to support C/C++ software that is not really supported by maven. Among the features are:

- multi language support: C/C++, java, python.
- generation of source code skeletons from templates: C++ class, unit test class
- target architecture handling (including cross compilation)
- dependencies management
- binaries identification and traceability
- auto installable package generation.
- configuration management helpers and automation for continuous integration.

1.2 References

	Authors Reference	Title Edition
R1	A. Oram, S. Talbott: 1986, O'Reilly	*Managing project with make*
R2	https://en.wikipedia.org/wiki/XUnit	*xUnit, software unit test frameworks familly.*

1.3 Glossary

URL	Uniform Resource Locator

2 Definition

2.1 Features and requirements

abs.lang.1	ABS shall build C/C++ executables and shared libraries
abs.lang.2	ABS shall build java packages
abs.lang.3	ABS shall package python software and libraries
abs.dep.1	buidlscripts shall automatically fetch from remote repository all needed dependencies to build the software
abs.run.1	ABS shall enable the user to run the developed software from its workspace
abs.run.2	ABS shall enable the user to run in a debugger the developed software from its workspace
abs.run.3	ABS shall enable the user to launch the unit tests from its workspace

abs.run.4	ABS shall enable the user to launch in a debugger the unit tests from its workspace

abs.cm.1	ABS shall insert in generated binaries the project's and version's identifiers

abs.cm.2	ABS shall insert in binaries the build context information (who, when, where)

abs.cm.3	ABS shall insert in generated binaries the build parameters (when the project use conditional compilation)

abs.cm.4	ABS shall be compatible with continuous integration automation

Only one build process should be used from both developer's workspace and continuous integration tools such as jenkins to ensure consistency of builds and coherence from various context.

abs.arch.1	ABS shall support several target computing architecture (various processor and operating systems)

The exact list of targets, is not specified and will be adjusted over the time according the user projects' needs. At least, it should enable build of software targeting standard x86 PC running widely used not too old GNU/Linux operating systems. First implementations were started using RedHat 5 and Debian 7, and is currently used on projects targeting CentOs 7, SUSE 12, Debian 8 and Debian 9. Most features should require only GNU/make, a shell and a few additional commands and it should be adaptable to any system where such software is available. Some experimental projects were successfully built with ABS on Android (through termux) and Cygwin/MinGW.

2.2 Design

2.2.1 from make to GNU make towards generic makefile

Make is a tool developed in 1977. It was design to help software developers build binaries from multiple source files in a more efficient way that using a static script to chain all the require compiler call. One major improvement of make is to be able to perform only the necessary operations to build the target files according the current intermediate files that may already be available. In its day to day job, the software developer only edit a few files and need to check its change by building the new software and it is useless to compile again unchanged source files. To enable such capability, make propose to defines rules instead of static action sequences. A rule is defined by three properties:

- the target file: this is the goal of the rule, the file that is expected to be done when applying the rule.
- the prerequisite: the list of files that are needed to build the target file.
- the transformation command: a shell command that generates the target file from the prerequisite files.

The rules inserted in a text file constitute a makefile, that is, a file to use as input script for the make command. When such file is named `Makefile` in the current directory, it is used as the default input by the make command.

Example: a make rule to generate concat.txt file by concatenating to text files a.txt and b.txt

```
1 concat.txt: a.txt b.txt
2     cat a.txt b.txt > concat.txt
3
```

Make also enable to define rules applied to file type (or more exactly on file name extension) to define once for all one type of transformation. It also enable to define macros, or variables to avoid duplication of definitions.

Here is a more detailed example of makefile. It defines the transformation of C source file to object files, use gcc as the compiler, and a final binary built from two object files. Since there is a rule to get object file from C source files, the target binary is finally made from the two C source files.

```
1 CC=gcc
2
3 cmd: a.o b.o
```

```
    $(CC) -o cmd a.o b.o

.c.o:
    $(CC) -c $*.c
```

Using this makefile, and supposing a.c and b.c are available and are containing valid C code, invoking make will chain the required gcc command call sequence to build the final cmd target:

```
$ make
gcc -c a.c
gcc -c b.c
gcc -o cmd a.o b.o
```

Then, working on the software project, if for instance only b.c is changed, invoking make will only perform the needed operation to build the out of date intermediate and target files:

```
$ make
gcc -c b.c
gcc -o cmd a.o b.o
```

As it is shown in this example, make needs to know explicitly the exact list of at least some intermediate files to be able to determine what shall be done exactly. To learn more about make, and how to use it to manage software project, see document [R1] .

To go really generic, we need to find a way to identify the intermediate files just by searching for source files. Unfortunately it is not achievable using only the classical make. Most build system relies on specific scripts, trying to generate the makefile itself from the project parameters and existing files. Another way to do is to use an improve make such as the GNU project's make implementation. It provide new way to define more complex filename patterns in rules and comes with many advanced macro that can support an automatic definition of variable from files or many other processing. So let's improve our previous example using GNU Make to become more generic with the help of some project layout convention:

- source files are in the src directory.
- intermediate files go in the obj directory.

Here is the GNU makefile:

```
TARGET=cmd
CC=gcc
SRCFILES:=$(wildcard src/*)
OBJFILES:=$(patsubst src/%.c,obj/%.o,$(SRCFILES))

$(TARGET): $(OBJFILES)
    $(CC) -o $@ $(OBJFILES)

obj/%.o: src/%.c
    $(CC) -o $@ -c $^
```

Now, if a new source file is added in the src directory, there is no more need to edit the makefile to ensure it will be compiled and the related object code linked to the final target. And the only definition that is project specific is the TARGET variable definition (1st line of the Makefile). Since GNU make is able to include files, it is possible to store the few project specific definition in a dedicated file, then there won't be anymore a reason to edit the Makefile itself.

Another interesting feature of GNU make is when the include directive is used, it adds a virtual prerequisite to the makefile to include. Then when it is not availalbe and there is a rule enabling its creation, the rule is first applied and once the file is newly created, it is included. This specific feature is used by ABS to let the ABS scripts be downloaded from a remote repository. The projects using abs does not need to import in themself a copy of the full abs, but only to use as the default makefile, the ABS bootstrap makefile that have only a very few rules able to:

- include the application specific parameters
- download and extract the ABS packages from a remote repository.
- include the ABS main script

This bootstrap Makefile structure is the following:

```
# include application specific parameters
include app.cfg
# include the main ABS makefile
include .abs/core/main.mk

# This rule enable to download, from a remote repository, the core ABS package providing
# the .abs/core/main.mk makefile among others
.abs/%/main.mk:
    mkdir -p .abs
    wget $(patsubst .abs/%/main.mk,$(ABS_REPO)/abs-%.tar.gz,$@) -O - | tar xvzf - -C .abs --strip-components=1

```

The real bootstrap file will contain some additional rules and variables to be able to use a local repository as well as a remote one, and to manage ABS versions.

2.2.2 Project standardized layout

To let a generic build system works, some common conventions are still needed. The project shall have two level of directory:

- the application level, that is in fact the root level.
- the modules level, an application could be made of several module and each module is hosted in a specific subdirectory of the root directory. Of course a simple and small application may be composed of one single module.

At each level, that is into the root directory and into each module directory, the ABS feature are brought by:

- the abs bootstrap makefile named as `Makefile` to be selected as the default makefile when running the make command.
- a configuration file, named app.mk at application level and module.mk at module level.

The only directories where ABS shall create files are (with `<prjroot>` being your project's root directory):

- `$HOME/.abs` : to store the local copy of ABS itself.
- `<prjroot>/build` : all files generated by the build process.
- `<prjroot>/dist` : stores the distribution package archives.

Those three directories, since containing only outputs of the generation process should never be set under revision control.

Besides the output directories, the `app.cfg` and `Makefile` files, the project's root directory shall only contains directories: one for each module.

2.2.3 Modularity

The GNU make include feature shall be used to:

- avoid any make code, rules and macro definition duplication, by using a common file to be included to store common definition.
- enable project and modules configuration adaptation, by defining specific rules and macros into dedicated file that are included only when needed.

The bootstrap makefile includes a single main core makefile that will itself include the current application, and module configuration, and then include the needed sub-makefile regarding the application and module's parameters.

ABS itself is split in several module. Each module is a set of makefile and companion scripts handling a specific set of tasks. The ABS modules are:

- core: ABS core features

  - application, module configuration parameters handling
  - programming language support: C/C++, java, python.
  - unit tests management.
  - operating system identification.

- doc: document processing features
- mstrans: matlab/simulink to SMP2 wrapping features
- jenkins: jenkins integration helper scripts.

Inside the ABS core module, the technical service and utility features are implemented into the following files:

- main.mk: detects whether the current directory is the application root or a module directory to include the appropriate file.
- app.mk: application level service.
  - builds all modules
  - run tests of all modules
  - build distribution packages
  - creation of new modules
  - complete build, test, packaging and package publication triggered by the single target `cint` [**abs.cm.4**].
- module.mk: module level service. Reads the module parameters to include the required specialized makefile according the module type and the specific needed features and extensions requested by the module configuration.
- module-absext.mk: defines mapping between module types and makefile to include name patterns
- module-extlib.mk: external library management.
- module-util.mk: provide additionnal features (create source files from template, run lua shell, display ABS variables values).

All module type specialized makefile shall provide target that are consistent with the main targets:

- all: default target that builds everything (unique to module.mk to avoid bugs on this target)
- all-impl: implementation of the all target that builds everything. The target all depends of this target.
- clean: remove all output files
- check, test, debugcheck, debugtest: run unit tests without or with debugger.
- run: launch the built software.

When any of the above target is not meaningful for the module or extension (for instance `run` is not relevant for a documentation module), the specific ABS scripts shall be tolerant to the use of the target anyway. That means using such target shall not fail in this case but only report nothing has to be done. Thus all modules, whatever the type, can be handled the same way at the highest level and the automation of module target calling sequence just by simple recursions is feasible.

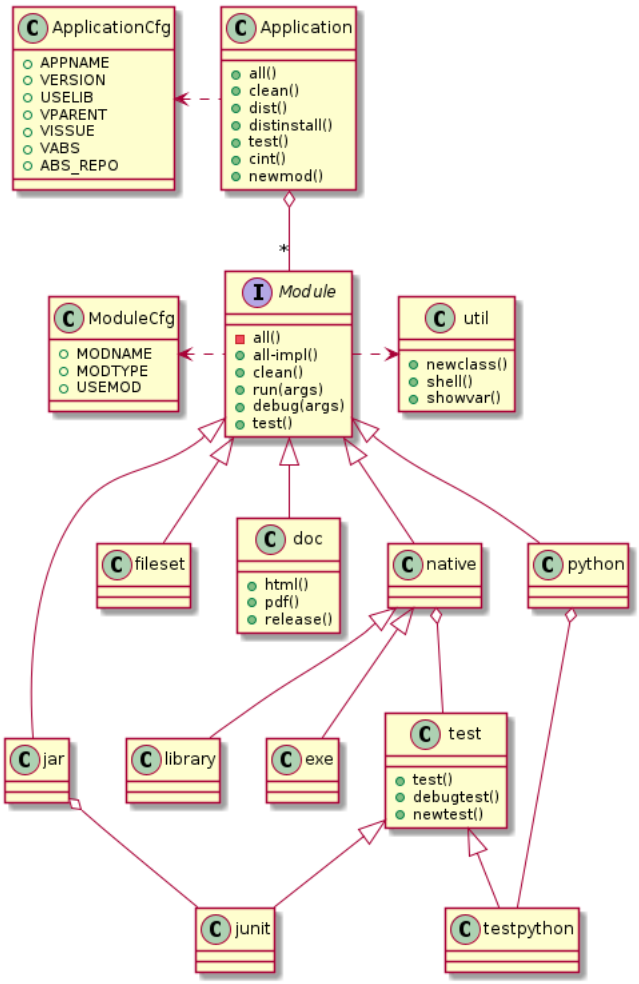

Figure 1: ABS interfaces and module types hierarchy

The above diagrams enumerates the minimal configuration parameters expected at application and module levels. The variable index in section 4.6 , provides the name and description of all variables including internal variables to be used by the module type specific makefiles.

2.2.4 Modules commons

2.2.4.1 Host operating system and hardware identification

Many ABS features are host dependant. The first and main one is obviously the compiler. Then an architecture identifier is used at different stages of the build process, in particular to tag distribution archives to know by its where such archive can be used.

The architecture identifier is made of two parts:

- first is the operating system identifier. It is computed by the script OS.sh from the ABS core package. This script makes use of the `lsb_release` when available that provides Linux distribution identifier, operating system version identifier and few other attributes. It can also use the `uname` command that provide similar information where `lsb_release` is not available. The final identifier, in most case, include the distribution identifier and a major version number. Finer attributes may be added when incompatibilities between operating system releases are discovered.
- second is the CPU architecture name. It is provided by the `uname -m` command.

2.2.4.2 debug and release modes

By default, ABS manages two build mode:

- debug: default mode for the everyday developer work.

- release: default mode for the distribution package.

Each sub-makefile handling a specific module type shall adapt its part of the build process according the mode. The debug mode shall provide debug information and disable any optimization is the final targets, while the release mode shall remove the debug information and can apply some optimizations.

2.2.4.3 Output layout

All intermediate and output files of the build process are stored in a dedicated directory tree. Its naming pattern starting from the project root is `build/<arch>/<mode>` , where:

- `<arch>` is the operating system and architecture identifier (see § 2.2.4.1)
- `<mode>` is the mode name (see § 2.2.4.2)

From this directory, a standard unix layout is applied, and it may contain the following subdirectories:

- `bin` : executable commands.
- `etc` : configuration files.
- `include` : header files.
- `lib` : stores libraries.
- `log` : log files.
- `obj` : intermediate files, any kind. It includes of course object files produced by the compilers, but also generated source files if needed.
- `share` : shared data files, resources, documentation
- `test` : output files from tests.

2.2.4.4 Dependencies management

One preliminary prerequisite of the build process is to fetch the needed external libraries. The needed external libraries list is defined using the USEMOD variable at application level.

The `module-extlib.mk` makefile from the ABS core packages defines the rules to download the ABS compatible packages from a distribution repository. The URL of the repository is defined by a variable. Its default value is the primary ABS repository. A repository list can be set. The `getdist.sh` script included in the ABS core package handles the download and extract. It use the repository list to search for the library to download by trying one by one each repository in the list order.

Once download and extracted int the `build/<arch>/extlib` directory, the makefile import.mk is included to enable the imported library to enrich the build process to for instance:

- fetch itself its own required external libraries.
- enrich the make variables, for instance add specific flags to the compiler command.
- provide new rules and even ABS extensions.

Then an ABS compatible library distribution shall include in its archive root directory a file named import.mk.

The application level makefile app.mk, include a rule to generate the import.mk file when building a distribution archive. Without specific directive, the generation is performed accordingly to the application configuration (name, version, and own dependencies from its own USELIB setting).

The `module-extlib.mk` makefile also perform some checks to detect conflicts between dependencies. To do so, it use the GNU make text macro to check each library is not present more than once. The steps of the checking is:

- get the complete sorted list of external library (containing direct and indirect imported libraries)
- remove the version numbers in the list.
- build a second list by sorting (the sort function removes duplicated words).
- if the two list are not identical, it means one entry has been removed in the last sort because it was present at least twice before the sort, revealing a dependency consistency.

This makefile also provide a `checkdep` target. It calls the `deptool.bash` script from the ABS core package. This script, once external libraries have been fetched, is able to extract, merge and translate project configuration and settings from the related `import.mk` files into a GraphViz dot script that is then rendered as an image (see example in section 4.4.5).

2.2.5 C/C++ support

C/C++ module support is brought by the core ABS module. A C/C++ module shall have only one "real" target: one library (shared object file) or one binary (standard executable file). The C/C++ features and related makefiles are included with the ABS core distribution, more specifically, the C/C++ features are provided by the following makefiles:

- modules-cheaders.mk: rules to publish headers. Related to the PUB_H module configuration parameters, introduced only to ease the integration of existing software as an ABS managed project.
- module-crules-var.mk: defines common variable for C/C++ handling.
- module.crules.mk: defines rules to go from source code to final binary.
- module-exe.mk: defines executable target for `exe` module type.
- module-library.mk: defines shared object target for `library` module type.
- module-test.mk: provides Unit test features based on cppunit.

For C/C++ the build process and related rules overview is the following:

- The list of object files is derived from the list of source files (files from module src directory having .c or .cpp extension).
- an object file is done from the corresponding C source file by running the C compiler.
- alternatively an object file is done from the corresponding C++ source file by running the C++ compiler
- the final target is made from all object file by running the linker command.

Some advanced rules are included to define dynamically dependencies to header files directly from source and ensure all needed source files are processed again when any header file is changed.

Additionally, there are some rule to generate intermediate source files, that will be themselves derived as object file and complete the list of objects to link in the final target:

- vinfo.c or vinfo.cpp: defines a static string containing meta data about the application and module, such as name, version, or build option. This string is used by the mastol logger to print a signature of each library when the software is started. It enables also to use the unix `ident` command to extract the information directly from the built binary file.
- resource C file: such file is done for each resource file found in the module's src/res directory. It defines a static buffer and set its value to the content of the corresponding resource file in src/res. This enable to define long static string from flat files, or any kind of resource (image, or any other) to be embed in the target library.

2.2.6 Java support

Java is handled slightly the same way than native compilation. Where native compilation is a two step process, building object file from source file, then linking all together in an executable or a library, the Java build is also a two step process: for each java source file, a .class file is build, then all .class files are compressed in a .jar file.

2.2.7 Python Support

Python being a programming platform relying first on an interpretor, the python build process is more a file packaging process. The python files are copied into the build directory tree trying to comply to what's python expects when loading libraries.

2.2.8 Unit test management

Unit test management is defined in language dedicated makefiles of the core ABS package:

- module-test.mk: C/C++ unit test handling using cppunit.
- module-junit.mk: java unit test handling using JUnit.
- module-testpython.mk: python unit test handling

Each language use its own test runner and unit test library but all should be compliant to the xUnit general principles (see [R2]). From the ABS perspective, all test procedure are, first eventually complied (C/C++/java), then run sequentially.

Of course, the test target has a dependency over the main module target to ensure the software to be checked is built before running tests. The test run progression is printed on the standard output and in a file for further post-analysis. On test run completion a report file is generated using the JUnit report XML format to enable processing of results with third party tools in general and within jenkins in particular. A summary is also printed on the standard output by XSLT processing of the XML report (see xunit2txt.xml in ABS core package).

2.2.9 Generic file set handling

The `fileset` module type is the ultimate generic module, but it finally performs almost no processing, it just copy files as-is from module's src directory to the application target directory. The only specific processing is to grant execution to files comming from module's src/bin if any. Since those files destination is the application's bin directory, we choose to apply the unix general rule to store there executable files only.

2.2.10 Document processing

Document processing feature is brought when using the `doc` module type. It is implemented by a the dedicated ABS-doc package. Its main components are:

- main.mk: document file processing rules
- html directory: resource for generated html documents.
- style.xhtml.xls: heml transformation to html format
- tex directory: latex templates and macros
- style.tex.xsl: heml transformation to tex format.

This package uses external components :

- heml processor: enables converting the document input format using XSLT style sheets. It is used by ABS-doc to generate both tex and html files from a single source heml document.
- plantuml: generates graphic UML diagrams from textual object specifications.
- dia: inline export of dia diagrams to document embeddable png files.
- doxygen: reference documentation generation from the commented source code.

2.2.11 Cross compilation

ABS enable cross copmilation where cross-compilers are available. The XARCH varaible is used to someway force the architecture identifier to the target architecture. It also includes also some standard cross compilation configration. Those configuration set specific value to many variable such as CC, CPPC, etc. Those variables define the command to use for many compilation tasks and of course needs specific value to use alternate compilers and linkers, that are in this case the cross compilers and linkers.

2.2.12 Dependencies management

2.2.12.1 Scope

ABS manage two kind of dependencies:

- internal dependencies between modules. As presented earlier, an ABS compatible project may be composed of several modules and each module may need some other modules to be built first.
- external library dependencies. To manage external libraries (that can be software components built with ABS), ABS provide fetching feature and packaging format to automate the integration of the required external libraries binaries into a project workspace.

2.2.12.2 Internal modules dependencies

A module dependency is define at module configuration level: a module requesting other modules to be built first just have to set the required modules names into the `USEMOD` variable. At generic module level rules (defined in `core/module.mk`),

a explicit dependency to all sources files to a dependency build stamp file is derived from the USEMOD definition. This enables to trigger automatically the build of missing required modules, before processing any source file of the currently build module. USEMOD definitions are also processed at application wide level (see core/app.mk , to let the make command compute the modules building order consistently with each internal dependency.

USEMOD definition is also processed by more language and compilers specific rules (see for instance core/module-crules-*.mk) to compute the needed command flags to let commands retrieve the files from the dependent modules (see for instance CFLAGS, LDFLAGS definitions).

2.2.12.3 External library dependencies

In a similar way, to make use of an external library, the application wide configuration file (app.mk) shall define the USELIB variable with the names (including version number) of the needed external libraries.

The handling of the USELIB variable is mostly defined in core/module-extlib.mk), this sub make file contains the rules to include a specific import.mk file that should be present in the precompiled software package to import (see here after for more information about this file). When this file is not founde, a rule is activated to download the package from a remote repository. The repository root URL is defined by the LIB_REPO variable. Its default value is the ABS repository URL. Once downloaded and included, the external dependency may itself have other dependencies. Using the same rules, all dependencies of dependencies are transitively fetched.

Then a ABS compatible library package organisation should be the following, inspired from standard Unix file organization:

- root directory named <libname>-<version>
  - import.mk : ABS integration file
  - bin : scripts and commands binaries.
  - lib : shared libraries and objects binaries.
  - etc : configuration files
  - share : additional shared files, documentation, ...

An ABS library package should be named according this scheme: <libname>-<version>.<arch>.tar.gz .

The import.mk file is included to the main Makefile when invoking make to perform a build. Then it should alter as needed the ABS make variables to complete flags and other parameters to enable all processing commands (mainly compilers) as well ase the USELIB variable itself to declare new dependencies. For project managed with ABS, import.mk file is automatically generated and inserted in the built binary package when using the dist target from application level (then defined in the core/app.mk). However, for external project, the import.mk is in most case easy to implement by a one line make macro call. For a library named AAA in version VA and using itself libraries BBB and CCC in version VB and VC, the import.mk file is:

```
1 $(eval $(call extlib_import_template,AAA,VA,BBB-VB CCC-VC))
```

2.2.13 Extensibility

ABS is extensible. A pattern matching rule is defined in module.mk to let ABS itself to download additional package (mostly set of makefiles) for some specific module types (defined in the module configuration file module.cfg using MODTYPE package). The MODULE_TYPE_MAP defines the assignment of module type over ABS packages. Its initial value is defines in core/module.mk and can be extended.

A project using ABS can itself provide its own ABS extension. ABS extension management is handled by core/module-absext.mk that provide many helpers rules to manage the integration of the extension in the distributable package of the application through its embedded import.mk.

2.2.14 In-line help

Master makefiles (app.mk and module.mk include a generic help target to display some short user documentation. It is just a grep command to display all lines starting with "## " (two sharp, one space) of all included makefiles. To let this system display readable help messages, try to apply the following rules while developing and maintaining all ABS' makefiles:

- Start with a header giving a clear section break and short description of the scope of the makefile.
- introduce overloadable variables definition section
- in-line variable comment tells the role and the default value of the variable
- introduce the targets section
- in-line target comment tells the role of the target. Remember to mention variables expected on make invocation command line that are specifically used by the target.
- limit line length to 80 characters max to keep readable even on text console.

Those rules are summarized by the following template:

```
## ------------------------------------------------------------------
## Documented makefile template
## Short introduction of the services provided by this makefile
## ------------------------------------------------------------------
##
## Overloadable variables :
##
##  - VAR1: variable 1 description, default value is value1
VAR1=value1
##  - VAR2: variable 2 description, default value is value2
VAR2=value2
##
## Targets:
##   - tgt1 A=arg: target 1 description, A argument is ...
tgt1:
    echo "A=$(A)"
##   - tgt2 [B=arg]: target 2 description, B optionnal argument is ...
tgt2:
    echo "B=[$(B)]"
```

Listing 1: Documented makefile template

Since only included makefiles are parsed for displaying help, the content displayed depends on the module configuration. It should then only print help about features available to the current configuration. For instance java build service help is shown only from java modules. When the inclusion of the makefile is dependant of the target set on make invocation, its help will not be printed when calling only help target, unless the help target itself is added to the include target selection. As an example, see the following extract showing conditional inclusion of test services sub-makefiles:

```
INCTESTS:=$(findstring test,$(MAKECMDGOALS))$(findstring check,$(MAKECMDGOALS))$(findstring help,$( ↩
    MAKECMDGOALS))
ifneq ($(INCTESTS),)
 ifneq ($(findstring library,$(MODTYPE))$(findstring exe,$(MODTYPE)),)
  $(info including module-test)
  include $(PRJROOT)/make/module-test.mk
 else ifeq ($(MODTYPE),jar)
  include $(PRJROOT)/make/module-junit.mk
 else ifeq ($(MODTYPE),python)
  include $(PRJROOT)/make/module-testpython.mk
 endif
endif
```

Listing 2: module.mk extract - test services conditional include section

2.3 External libraries and dependencies

Buildscripts need to have some tools available on the workstation for its own purpose.

Table 1: Dependencies list

Tool/lib	description	license
gcc	C compiler (default)	GPL
g++	C++ compiler (default)	GPL
g++	Linker (default)	GPL

Tool/lib	description	license
gdb	The GNU debugger	GPL
javac	Java compiler and development kit	JDK Supplier dependent, GPL for OpenJDK
xsltproc	libxml2's XSLT processor	MIT license
Doxygen	Document generator	GPL
Latex	Document processing	Latex project public license
Bourne shell	command interpreter	GPL (bash)
GNU make	make command	GPL
GNU ed	File editor	GPL
GraphViz	diagram generator	Eclipse Public License
curl	command line HTTP client	MIT/X derivate
wget	alternate HTTP client	GPL
ssh/scp	secure shell client	BSD derivate
subversion	revision control system	Apache/BSD
java	Java Runtime Environment	JRE supplier dependent, GPL for OpenJDK
GNU tar	GNU tape archive tool	GPL
valgrind	debugging and profiling tool	GPL

about the GPL and copyleft licenses

All GPL licensed product are used as is in development environment and are not needed to be deployed with any ABS managed project on operational target, unless such project itself makes direct and explicit use of such product for its own purpose. Then it is possible and allowed to develop full proprietary software with abs.

GNU make and a *not too old* Bourne shell (bash preferred) are the only mandatory prerequisites:

- other tools listed above are required only for the feature they support. For instance a java compiler is needed only when the project managed with ABS includes java code.
- for a given feature, only one tool supporting the feature is needed. For instance for C++ projects, one can choose gcc or clang. It is not necessary to have both available on your workstation. Any other C compiler can even be used as soon it supports the same command line flags. However some advanced feature may still need a particular implementation (See table below).
- no precise version information is provided since many Linux and posix like system are known to be compatible. Buildscripts is daily used with various RedHawk/CentOS Linux versions (starting from version 6.3), Debian Linux. Basic features are working Cygwin, Msys2 (GNU tools for Mingw64) despite quite slow. However some very particular features may be usable only from a few targets operating systems (for instance, obviously Linux kernel module support is operational only from a Linux host).

Features using specific tooling are:

Table 2: Per feature dependencies

Feature	related tools
External dependencies download	curl or wget
Release note generation	curl, subversion or git
PDF document generation	latex
Document generation from source code	Doxygen
UML class diagrams generation from source code	Doxygen, xsltproc, GraphViz
Distribution archive publication	ssh/scp
Diagram rendering	graphviz (through document processing features)
Distribution archive generation	GNU tar
Debugging execution	gdb
Profiling execution	valgrind
Automated configuration management	subversion, GNU ed

3 Validation plan

3.1 Test strategy

The dummy project named sampleprj is setup to check many ABS features:

- the module management is checked through the several modules created in sampleprj.
- C/C++ library build is checked using the ccplib module.
- C/C++ executable build is checked using the cppexe module.
- java package build is checked using the jmod module.
- python software packaging is checked using the psysubpckA and pysubpckB modules.
- modules include unit tests to check related test automation features.
- document edition and generation features are checked using the _doc module.

Some common references are used in the procedures described in the next section to identifiy file path and repository location:

- <wsroot> is your workspace root directory
- <installpath> is an installation path for installing built components during tests.

3.2 Procedures and analysis

3.2.1 Control procedure general - General functional test

Operation #1
Checkout the abs release to be tested, then go to the sampleprj sub-directory (replace X.Y.Z by the version to test):

```
$ cd <wsroot>
$ git clone https://github.com/seeduvax/AcrobatomaticBuildSystem.git
cd AcrobatomaticBuildSystem
git checkout abs-X.Y.Z
cd sampleprj
```

Operation #2
Build the sample project's install archive:

```
$ make distinstall
```

Assert #1
Project is built. At the end of the process the installation archive is available in dist directory as sampleprj-0.4.Xd.<arch>-install.bin

Operation #3
Install the package to the test install location <installpath> :

```
$ ./dist/sampleprj-0.4.Xd.<arch>-install.bin install <installpath>
```

Assert #2
The project C/C++ binaires are built and installed (from <installpath>):

- bin/sampleprj
- bin/sampleprj_cppexe
- lib/libsampleprj_cpplib.so
- lib/libsampleprj.so

Checked req.: abs.lang.1

Assert #3
The project java packages are built and installed (from <installpath>):

- lib/fr.example.sampleprj.jmod-0.4.Xe.jar

Checked req.: abs.lang.2

Assert #4
The project python scripts are integrated and installed (from <installpath>):

- bin/sampleprj_pysubpckA.sh
- bin/sampleprj_pysubpckB.sh
- lib/python/pysubpckA/moduleA.pyc
- lib/python/pysubpckB/moduleB.pyc
- lib/python/pysubpckB/pysubsub/subsub.pyc

Checked req.: abs.lang.3

Assert #5
The project's dependencies are integrated and installed (from <installpath>):

- bin/lua
- bin/luac
- lib/liblua.so

Checked req.: abs.dep.1

Operation #4
Extract configuration management information from the binaries:

```
1 $ ident <installpath>/lib/libsampleprj*
```


Assert #6
Configuration management data (product name, version, subversion revision, build context) is printed on screen like in the following sample:

```
<installpath>/libsampleprj_cpplib.so:
     $Attr: app.name=sampleprj $
     $Attr: app.version=0.4.2e $
     $Attr: app.revision=undef $
     $Attr: app.file=libsampleprj_cpplib.so $
     $Attr: company=eduvax $
     $Attr: copyright= $
     $Attr: build.mode=release $
     $Attr: build.opts= $
     $Attr: build.date=jeudi 2 novembre 2017, 09:59:37 (UTC+0100) $
     $Attr: build.host=tethys $
     $Attr: build.user=sdevaux $
     $Attr: build.id=null $

/home/net/m026258/temp/sampleprj-0.4/lib/libsampleprj.so:
     $Attr: app.name=sampleprj $
     $Attr: app.version=0.4.2e $
     $Attr: app.revision=undef $
     $Attr: app.file=libsampleprj.so $
     $Attr: company=eduvax $
     $Attr: copyright= $
     $Attr: build.mode=release $
     $Attr: build.opts= $
     $Attr: build.date=jeudi 2 novembre 2017, 09:59:37 (UTC+0100) $
     $Attr: build.host=tethys $
     $Attr: build.user=sdevaux $
     $Attr: build.id=null $
```

Version numbers may change regarding the `sampleprj` version you are using for the test. The build attributes (date, host, user) are related to the build operation done in first steps of the current test procedure.
Checked req.: abs.cm.1 abs.cm.2

3.2.2 Control procedure arch - multi architecture support

Checked requirements abs.arch.1

Operation #1
Run the check procedure `general` (see section **??**) from a Debian 8 x86/64 host.

Assert #1
All assertions from the procedure `general` are satisfied.

Assert #2
The install package file name is `sampleprj-0.4.Xd.Debian_8_x86_64-install.bin` .

Operation #2
Run the check procedure `general` (see section **??**) from a RedHawk 7 x86/64 host.

Assert #3
All assertions from the procedure `general` are satisfied.

Assert #4
The install package file name is `sampleprj-0.4.Xd.CentOS_7_x86_64-install.bin` .

3.2.3 Control procedure run - Running debugging and testing features test

Operation #1
Go to <wsroot> and checkout sampleprj-0.4, the go to the downloaded project's root directory:

```
$ cd <wsroot>
$ svn co <repo_url>/branches/sampleprj-0.4
cd sampleprj-0.4
```

Operation #2
Enter the cppexe module directory

```
$ cd <wsroot>/sampleprj-0.4/cppexe
```

Operation #3
Build and Run the sampleprj cppexe module:

```
$ make run
```

Assert #1
The software is built and run. It's dummy output is printed on the console.

```
Hello world from sampleprj::cpplib!
len=33
bufptr=b3820e60
[Text to be embedded as resource

]
```

Note: the bufptr value can be different, it is a non predictible pointer value.
Checked req.: abs.run.1

Operation #4
Start the debugger to run cppexe into:

```
$ make debug
```

Assert #2
gdb is started

Operation #5
Start cppexe from the debugger:

```
(gdb) runapp
```

Assert #3
cppexe is run, and returns to the debugger prompt when done.
Checked req.: abs.run.2

Operation #6 exit the debugger

Operation #7
Jump to cpplib module and run unit tests

```
$ cd ../cpplib
$ make test
```

Assert #4
Unit test are run and the test report is printed on the console. The sampleprj is configure to have one test OK, and one test failing to be able to check error detection and restitution in the report. The expected test summary is the following:

```
# ------------------------------------------------------------------------
# Successful tests
# ------------------------------------------------------------------------
- N4test11TestExampleE::testCaseSuccess 1.7083e-05 s (0.017083 ms).

# ------------------------------------------------------------------------
# Disabled tests
# ------------------------------------------------------------------------

# ------------------------------------------------------------------------
# Failed tests
# ------------------------------------------------------------------------
- N4test11TestExampleE::testCaseFail    9.1389e-05 s (0.091389 ms).
test/TestExample.cpp:32
equality assertion failed
- Expected: 0
- Actual  : 1

# ------------------------------------------------------------------------
# Statistics
# ------------------------------------------------------------------------
- Tests: 2
- Total Failures: 1
- Total Errors: 0
- Total Disabled: 0
- Total time: 0.000108472 s (0.108472 ms)
```

Of course, the exact timings may be different.
Checked req.: abs.run.3

Operation #8
Start the debugger to run the tests:

```
$ make debugtest
```

Assert #5
gdb is started

Operation #9
Run the tests from the debugger:

```
(gdb) runtests
```

Assert #6
Tests are run as previously (but the final summary is not displayed) and returns to the debugger prompt when done.
Checked req.: abs.run.4

Operation #10 Exit the debugger, the test is completed.

4 User guide

4.1 Project layout

4.1.1 Project level file hierarchy

Buildscripts expects one specific files and directories layout to be able to apply its generic rules and perform all needed actions to achieve to build according the current project state. The standard layout is:

- <project name>-X.Y/ : root directory for the project's X.Y branch workspace.
  - _doc/ : documentation module
  - <1st module name>/ : 1st module directory.
  - <2nd module name>/ : 2nd module directory.
  - ...
  - <Nth module name>/ : Nth module directory.
  - app.cfg : project configuration file
  - Makefile : project's root makefile, copy of the ABS bootstrap makefile[1] .
  - build/ : root directory for all generated artifacts during the build process. Shall not be inserted into the version control system.
  - local.cfg : optional workspace local configuration file. Shall not be inserted into the version control system.

4.1.2 Project configuration file app.cfg

This file hosts the application wide parameters. The mandatory definitions are:

- APPNAME=<application name> : name of the application. Caution: since this name is used to define some output files names, it shall not contain any space, the '-' character or any special characters that may not be supported in a file name.
- VERSION=X.Y.Z : version identifier. Must be initialized manually on project creation consistently with branch name set in the version control system. Then it is automatically maintained when branches are created from existing branch and when a branch is tagged.
- ABS_REPO:=<URL> : URL of the abs distribution repository.
- VABS:=X.Y : version of ABS to be used to build the application.

The optionals definitions are:

- COMPANY=<company name> : the name of the company or organization that owns the project.
- COPYRIGHT=<copyright declaration> : Copyright information. The standard format is the following: (c) <create date>[-<update date>] <copyright older name>
- EXPMOD=<mod name>* : list of modules to be exported. Any module that is a library to be used by another application should be exported.
- DOMAIN=<reversed domain name> : Java package name prefix.
- USELIB=<<libname>-X.Y.Z>* : list of external libraries directly used by the project.
- extra_import_defs=<any makefile content> : additional rules and macro definitions to be inserted into the import.mk file packaged with the product. This import.mk file is included in any user application that, through its own cnofiguration and USELIB value, makes reference to the product.

4.1.3 Module level file hierarchy

The internal file layout in modules is:

- <module name>
  - include/<project name>/<module name> : public C/C++ header files
  - src/ : implementation source files.
  - test/ : unit tests source code.
  - module.cfg : module's configuration file.
  - Makefile : module's local makefile, copy of the ABS bootstrap makefile[2] .

4.1.4 Module configuration file module.cfg

This file stores all the module specific configuration. The mandatory expected definition are:

- MODNAME=<module name> : the module's name. String without any special character is preferred, since this name will compose file names.
- MODTYPE=<module type> : the module's type. Use any of the following value
  - exe : "native" (C/C++) executable.
  - library : "native" (C/C++) library.

[1] http://www.eduvax.net/dist/noarch/abs-3.0-bootstrap.mk
[2] http://www.eduvax.net/dist/noarch/abs-3.0-bootstrap.mk

  - jar : java package
  - python : python package
  - linuxmodule : linux kernel module
  - doc : document set.

Additional definitions may be used:

- USEMOD=<module name>* : list of modules from the same project that are used by the current module.
- LINKLIB=<lib name>* : list of libraries to link with. Use short library name, that is, if abc is given, the expected shared object file name is libabc.so for Unix/Posix systems, or abc.dll for Windows systems.
- USEJAR=<pacakge name>* : list of java package needed by the current module (when module is itself a java package). Use only package/version name, that is if abc-1.2.10 is set, the file named abc-1.2.10.jar is included in the *CLASSPATH* .
- CFLAGS+=<CC argument>* : additional C/C++ compiler flags.
- LDFLAGS+=<LD argument>* : additional linker flags.
- MAINCLASS=<class name> : name of the class to be used as the main entry point class in a java package. This class must have a public static method named main and expecting one string array as parameter.

Anonymous module

When a module's name is equal to its project's name, it is considered as the default anonymous module. The only differences between the anonymous default module and any other module is the naming schemes:

Table 3: Naming schemes

Naming type	regular module	anonymous module
Executable target file (unix)	bin/<app>_<module>	bin/<app>
Executable target file (windows)	bin/<app>_<module>.exe	bin/<app>.exe
Library (unix)	lib/lib<app>_<module>.so	lib/lib<app>.so
Library (windows)	bin/<app>_<module>.dll	bin/<app>.dll
Include dir	include/<app>/<module>	include/<app>
java package name	<domain>.<app>.<module>	<domain>.<app>
C++ namespace	<app>::<module>	<app>

There is no obligation to create a anonymous module for a project. But a project can't have more than one anonymous module. Since all modules shall have distinct name, it is not possible to have two modules having the same name than the project.

4.1.5 linux kernel modules

Buildscripts can handle the compilation of linux kernel modules. To integrate an existing linux kernel module source code in an ABS managed project, apply following steps:

- create a new buidlscript module in the project using _lkm as name suffix:

```
make newmode M=<driver name>_lkm
```

- Edit module.cfg file to define it is a linux kernel module:

```
MODTYPE=linuxmodule
```

- Copy all source files (including root directory containing original makefile) int src dir.
- When some imported source file should not be compiled (sometimes, obsolete files are remaining in public source code distributions of kernel modules), filter out those files by defining the DISABLE_SRC variable (do not include src/ in listed file names).

```
DISABLE_SRC=file1.c file2.c ... fileN.c
```

- The header files to be included from user space applications or from other kernel modules will be published in include/<app>/<module> directory only when explicitely listed in module.cfg using the variable PUB_H . Specify file path from the src directory (do not include src/ itself.

```
PUB_H=header1.h header2.h ... headerN.h
```

The project's neighbour modules that need to include some of a kernel module header shall use USELKMOD variable instead of USEMOD because the dependency reference expansion to compiler and linker flags is not exactly the same for linux kernel modules and other modules.

4.2 ABS integration to a project

4.2.1 Integrate the make directory

Buildscripts is just a set of makefiles, scripts and style sheets. There is no binary to install, the only task is to include in the root directory, the ABS bootstrap makefile as Makefile , and define the project's configuration file.

4.2.2 Reference ABS versions

All packaged ABS version are available from the ABS distributions repository[3] . Some mirror may have been created in other location that can't reach this public primary repository, check people that have provided you this document that may know about such mirror. This document first public issue as written while integration ABS version 3.0.

4.2.3 Create the configuration files and bootstrap makefiles.

From the project root (where the make directory of ABS has been integrated):

- create app.cfg file and define at least the APPNAME , VERSION , VABS and ABS_REPO variables.
- extract the file core/bootstrap.mk from the ABS core distribution archive (or fetch the reference version such as abs-3.0-bootstrap.mk[4]) into the project root directory and rename it Makefile .
- when some modules are already existing, for each one go to its directory, then:
  - create module.cfg file en define at least the MODNAME and MODTYPE variables.
  - copy the bootstrap makefile already integrated to the root directory as the module's Makefile file.

After that you are ready to build your project with ABS by calling make , and even create new modules by calling make newmod M=<module name> . Next sections details all services available through make targets and variables that can be used to configure your project and enclosed modules.

4.3 Make targets

Buildscripts services and function are available as make targets. User can operate from project root to work with the whole project or at module level when focusing on a particular module.

4.3.1 At project level

The project level main targets are:

- make all : default target, builds all modules, without unit tests.
- make testbuild : builds all modules including unit tests but without running the tests.
- make clean : deletes all built artifacts (deletes the build and dist directories.
- make newmod M=<module name> : creates a new module. Create the directory structure, module.cfg and Makefile files with default minimal configuration (default module type is *library*).
- make branch : creates new branch from the current one. New branch identifier (X.Y) and comment including issue reference will be interactively requested during processing. Write access to the subversion repository is needed to proceed.
- make dist : builds distribuable binary "short" package. Such packaging is design to support publication to a build repository. The packages available on the build repository can be fetched by any project (see USELIB variable usage).

[3] http://www.eduvax.net/dist
[4] http://www.eduvax.net/dist/noarch/abs-3.0-bootstrap.mk

- make help : in-line help system. Show usage information about application level services.
- make doc : generate documentation from code. Doxygen is needed to process the source code and shall be available on your workstation to use this feature.
- make install [PREFIX=<install root>] : builds and installs the application. The default installation root is /opt/<app name>-<version> . All needed dependencies are also installed, so avoid using /usr or /usr/local as installation root to avoid any risk to overwrite some system libraries.
- make distinstall : generates an installation archive including the complete built application and its dependencies. The generated file is executable and handles, when later run, the installation process.
  - when project contains in its root directory a file named extradist.sh , this script is run during the processing of the dist target. It receive the target build path (default is dist/<app>-<version> as first argument. You can put in this script any commands that copy/create/update files into the dist directory that will then be included in the built final archive file.
  - using the extra dist feature, one can insert in the dist/<app>-<version>/bin directory a file named postinstall.sh . The in installation archive embedded script, at the end of the install process, looks at this file availability and if present run it with two arguments, the application name, and the application version. Caution: take care to set this postinstall.sh file the executable attribute in order to be invoked by the installation script.

Some utility and additional targets may be added according to ABS improvements. See in-line help for more complete reference.

4.3.2 At module level

The module level main targets are:

- make all : default target, builds the module. The build of other modules may be automatically triggered regarding the prerequisites definition (see USEMOD variable).
- make run [RUNARGS="[<arg>]*"] : builds and run the module. Applicable to executable modules or libraries found to include a lua_open function. In this second particular case, ABS attempts to launch the module from a lua shell and requires some feature from mastol. This target also ensure library search paths are consistent with dependencies definitions.
- make testbuild : builds the module and its unit tests but doesn't run the tests.
- make check [RUNARGS="+|-f <test name pattern>"] : builds the module and its unit tests, then run the tests.

Test naming

Cppunit that is used as the unit test engine for C/C++ modules, can use the symbol names as test identifier. To capture the symbol name, run once make check and see the right names to use to identify test on the test execution report.

```
$ make check
[...]
# ------------------------------------
# Successful tests
# ------------------------------------
- N4test11TestExampleE::testCaseSuccess 1.4995e-05 s (0.014995 ms).
[...]
$ make check RUNARGS="+f N4test11TestExampleE"
[...] only test from TestExample test class are done.
$ make check RUNARGS="-f testCaseSuccess"
[...] all test are done except methods named testCaseSuccess (whatever is the enclosing ↩
    test class)
```

- make debug : same as make run but inside a *gdb* session. Once gdb shell is ready, invoke runapp to start running the module in the debugger.
- make debugcheck : same as make check but inside a *gdb* session. Once gdb shell is ready, invoke runtests to start running the tests in the debugger.
- make test : see make check
- make debugtest : see make debugcheck
- make edebug : displays parameters required for debugging from *Eclipse* (applicable to executable modules only).
- make edebugtest : displays parameters required for debugging unit tests from *Eclipse* .

- `make newclass C=<class name>` : generate new empty class files (both implementation and headers for C++) complient to ABS rules.
- `make newtest T=<name of class to test>` : create source code file for a unit testing class.
- `make help` : in-line help system. Show usage information about per module available services.

Some utility targets may be added according to ABS improvements. See in-line help for more complete reference.

4.3.3 Options

the main build targets will use the following option (given as make command arguments) to specialize the build process:

- `MODE=<debug|release>` : compilation mode, with debug symbol and tracing activated or without debug symbol, tracing deactivated and some optimization flags. The default mode is *debug* .
- `DEFINES="<symbol1> <symbol2> ..."` : C/C++ preprocessor symbol definition. Implementation can provide alternate or special optional features and this argument enable the user to activate such. Any symbols provided in the list should match at least one `#ifdef` directive in C/C++ source code.

4.3.4 distribution archive and deployment

Official distributable packages should be built and published by the continuous integration system that includes many automated checks. However manual build of such package is possible through make targets (and the automated system use the same targets). The operations to be performed for the two step build and publish process are:

- use command `make dist` or `make distinstall` regarding the product is only a library or includes applications.
- copy the generated archive in `dist` to your web server hosting your binaries repository:

```
1 scp dist/<product>-<version>.<arch>.tar.gz <login>@<webserver>:<DocumentRoot>/dist/<arch>
```

Build and publish for every architecture

This process shall be repeated for every supported architecture, that is, build the package on every compilation host required to cover all the needed processor classes and Operating Systems.
To meet traceability requirements, only fresh checkout of tagged version of a product should be build and published to the repository. The continuous integration state is available through its web interface[a] .

[a] https://sdengrle.public.infrapub.fr.st.space.corp/jenkins/view/ASTRE/

Binaries identification.

Buildscripts includes in the C/C++ binaries an identification string. The `ident` command can parse the binary to retrieve that string and restore the build and configuration context.

```
1  $ ident mastol_luad
2  mastol_luad:
3       $Attr: app.name=mastol $
4       $Attr: app.version=1.1.10 $
5       $Attr: app.revision=5817. $
6       $Attr: app.file=mastol_luad $
7       $Attr: company=Airbus Defence and Space $
8       $Attr: copyright=(c) 2013-2017 Airbus Safran Launchers $
9       $Attr: build.mode=release $
10      $Attr: build.opts= $
11      $Attr: build.date=mer. mai 10 10:10:21 CEST 2017 $
12      $Attr: build.host=apollo $
13      $Attr: build.user=m026258 $
14      $Attr: build.id=null $
```

Some criteria to know the binary was produced through the validated process by continuous integration system are:

- version has no `d` suffix.
- revision has no `M` suffix.
- build mode is `release`
- build identifier is not null.

Only tagged software should deployed onto integration, validation or production target hosts.

4.3.5 Workspace local configuration

Aside the application configuration file app.cfg , some workspace local parameter definitions can be stored in a file named local.cfg . This file is dedicated to the handling or workaround of local workspace issues, such as using alternate binary repository. Please never add this file into the revision control system since it can overload the correct shared project configuration with values not applicable to every reference compilation hosts. Here are some examples of use:

```
1 LIB_REPO:=/the/local/path,$(LIB_REPO)
```

Listing 3: local.cfg - add local mirror to the binary repository search paths

```
1 ARCH=RedHatEnterpriseWorkstation_6.3_x86_64
```

Listing 4: local.cfg - force architecture name

4.4 Dependencies management

4.4.1 Sample projects

Let's have two applications:

- avionic inluding the mandatory module bus and the optional modules alpha and beta . Other dependencies are:
  - All modules need the external library space-0.2.5
  - beta needs alpha
  - and beta also uses another external library star-1.0.0
- spaceship is an application using version 1.1.0 of avionic and has itself two modules:
  - payload using only bus from avionic
  - booster using optional module beta from avionic .

4.4.2 Step 1 - define applications overall dependencies

To link an application to external libraries, define the USELIB macro int the application configuration file app.cfg

```
1 APPNAME=avionic
2 VERSION=1.1.0
3 USELIB=space-0.2.5 star-1.0.0
```

Listing 5: avionic/app.cfg

```
1 APPNAME=spaceship
2 VERSION=1.1.0
3 USELIB=avionic-1.0.0
```

Listing 6: spaceship/app.cfg

On version identifiers

Software components versions managed with ABS should be defined by three numbers and the software product full name format is <name>-X.Y.Z . Using this full specification allows to define dependencies without any ambiguity. But regarding the current project's maturity, such strict definition may be too disturbing since, during some period, versions can change frequently. Moreover, only tagged version are available through such definition and some recent needed updates may not be available through tags yet. To ease in development version definition, the two following kind of version reference can be used:

- <name>-X.Y : will bring the last tagged version of the component in branch X.Y. The downloaded content will then be different after each new tag in branch <name>-X.Y .
- <name>-X.Y.Zd : will bring the last build made by continuous integration system before tagging <name>-X.Y.Z .

Caution: the d suffix shall never be used in a product to be tagged. An application for which controlled state and configuration are expected can't refer to any "moving" dependency.

4.4.3 Step 2 - make reusable modules public

The module of an application to be reused by some other applications (typically the libraries) shall be explicitly declared to ensure the header files and other needed meta information are included in the published final package. Use the EXPMOD macro to list all the modules to be exported.

```
APPNAME=avionic
VERSION=1.1.0
USELIB=space-0.2.5 star-1.0.0
EXPMOD=bus alpha beta
```

Listing 7: avionic/app.cfg

4.4.4 Step 3 - explicit for each module the local dependencies

Locally at the module level, all dependencies shall be defined: USEMOD lists the other modules of the same application that are used by the current module. LINKLIB lists the few external shared objects brought by external dependencies that the current module directly use.
For the three avionic 's modules:

```
MODNAME=bus
MODTYPE=library
LINKLIB=space
```

Listing 8: avionic/bus/module.cfg

```
MODNAME=alpha
MODTYPE=library
LINKLIB=space
```

Listing 9: avionic/alpha/module.cfg

```
MODNAME=beta
MODTYPE=library
USEMOD=alpha
LINKLIB=space star
```

Listing 10: avionic/beta/module.cfg

Then for the two spaceship 's modules:

```
MODNAME=payload
MODTYPE=library
LINKLIB=avionic_bus
```

Listing 11: spaceship/payload/module.cfg

```
MODNAME=booster
MODTYPE=library
LINKLIB=avionic_beta
```

Listing 12: spaceship/booster/module.cfg

4.4.5 Dependencies check

However, despite any effort to follow the rules presented above, dependencie management can become far complex. Since a project defines only its direct dependencies, it can't ensure any used external library will not itself use a library also linked directly to the project but with a different version. To support the user and at least identify the library mix issues, some checks are performed on the full USELIB defintion recursively computed from include to include. A warning is displayed during the build when different versions of a same lib is found. Invoking make checkdep displays a complete depedency graph.

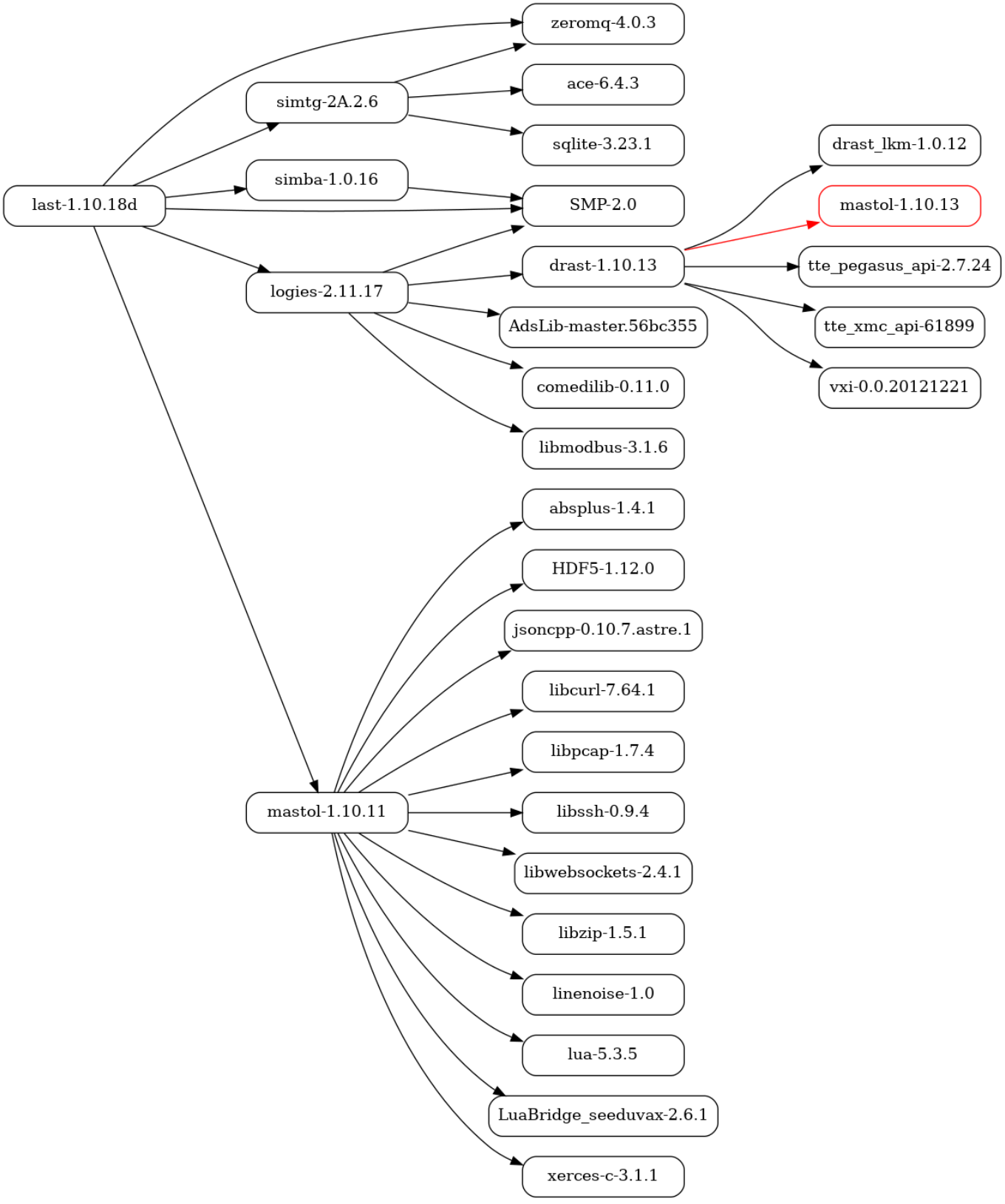

Figure 2: Real world example of a dependency graph with a version conflict

4.5 Documentation module

4.5.1 Extend documentation templates

ABS Doc module permit the use of additional templates to generate documentation files. This is done using an absext module (see section 2.2.13).

no need ABS_EXT_MAP variable in module.cfg , just include the main.mk file in app.mk to forward the inclusion in the generated import.mk file that is included in the distribution package:

```
extra_import_defs=include $$(_absext_index_<project name>_<module name>)/main.mk
```

In src/main.mk, add the path to the abs ext (_absext_index_<project name>_<module name>) in TEXINPUTS variable and add couple in HEMLTOTEX_MAP.
To use the new template, just import the generated project using USELIB variable in app.cfg . Use the DEFAULT_TEMPLATE variable to indicates the name of the template to use for the generation of the documentation.

4.5.1.1 Example of extension module

Module 'extmodexample' hierarchy in prjexample project:

- prjexample
  - app.cfg
  - Makefile
  - extmodexample
    - module.cfg
    - src
      - main.mk
  - styleExt.tex.xsl
  - styleExt.xhtml.xsl
  - styleExt.xml.xsl
  - module.cfg
    - Makefile

```
APPNAME=prjexample
extra_import_defs=\
include $$(_absext_index_prjexample_extmodexample)/main.mk # this permit to load the main.mk at the load of external libraries.
```

Listing 13: content of app.cfg

```
MODNAME=extmodexample
MODTYPE=absext
```

Listing 14: content of module.cfg

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:import href="style.tex.xsl"/> <!-- import the default style (in abs doc module) -->
</xsl:stylesheet>
```

Listing 15: minimal content of styleExt.*.xsl

```
TEXINPUTS+=$(_absext_index_prjexample_extmodexample)//
HEMLTOTEX_MAP+=<template name>:$(_absext_index_prjexample_extmodexample)/styleExt.tex.xsl
HEMLTOXHTML_MAP+=<template name>:$(_absext_index_prjexample_extmodexample)/styleExt.xhtml.xsl
HEMLTOXML_MAP+=<template name>:$(_absext_index_prjexample_extmodexample)/styleExt.xml.xsl
```

Listing 16: content of main.mk

4.5.1.2 Example of use in a _doc module

To use the extension, simply add the library in USELIB in app.cfg and add following varibles in the module.cfg.

Table 4: Variables in module.cfg

Variable	Description
DEFAULT_TEMPLATE	The name of the template to use for all the heml docs in module
TEMPLATE	Permit to specify a template for a specific heml file (the value is a list of <filename without extension>:<template name>)

4.6 Variables references

Any of the following variable referred and used by the various included makefiles can be overloaded (new definition using `=` or `:=` operators) into any configuration file: `app.cfg` , `local.cfg` or `module.cfg` , loaded in that order. Take care of the load order to ensure you get the expected value definition for each module in each workspace.
All variables can also be overloaded through the command line. The values set in command line take precedence over any other definition.

Table 5: Variable index and reference

Variable	Default value	Description, role and usage
ABS_REPO		URL of the distribution repository to use to fetch ABS.
APPNAME		Application name
ARCH	$(SYSNAME)_$(HWNAME)	Target's architecture name. Without other overloading, it shall reflect the real architecture of the current compilation host.
CC	gcc	C compiler command.
CFLAGS	-Iinclude -fPIC	C compilation options. Caution: strict overload may suppress all additions performed by the dependencies management from USELIB and USEMOD definitions.
COMPANY		Company name.
COPYRIGHT		Copyright info, usage is to follow this format: (c) <creation year>-[<publication year>] <company name>
CPPC	g++	C++ compiler command.
DEFINES		Extra C preprocessor symbols to define: each word contain by this variable is forwarded to the C/C++ compiler through CFLAGS as -D<symbol>
DOMAIN		Package name prefix pour java applications.
EXPMOD		List of public modules, that is the list of modules that are libraries with C/C++ header to be included in the distribution packages.
EXTLIBDIR	$(TRDIR)/extlib.$(ARCH)	Directory to host local copies of externals libraries.
JAR	jar	Java packager command
JAVA	java	Java run-time command
JAVAC	javac	Java compiler
JDB	jdb	Java debugger command
HWNAME	output of uname -m	CPU architecture name (such as x86_64, i686, ...)
LD	g++	Linker command.
LDFLAGS		Linker arguments. Caution: strict overload may suppress all additions performed by the dependencies management from USELIB, USEMOD and LINKLIB definitions.
LIB_REPO	$(ABS_REPO)	Pre-built package of external libs repository URL. May contain several location (',' is the separator character). Do not include file:// for local file systems locations.
MAINCLASS		Main class name for a java module.
MMARGS		Make command arguments to be forwarded from the application level global make to the modules' make. May be used to request multi-threaded compilation (see make documentation about -j option).

Variable	Default value	Description, role and usage
MODE	debug	Compilation mode, two values accepted: debug or release. Using first binaries are compiled without optimization and with debug symbols, while second requests application of some optimisation and no debug symbols.
MODNAME		Module's name
MODULES	<computed modules list>	List of module to build from the application level make. Defaultly this variable is set automatically by searching available makefiles in the project's subdirectories.
MODTYPE		Type of module, one of the following: jar (java package), library (C/C++ shared object), exe (C/C++ command), linuxmodule (linux kernel module), doc (document set), python (python library).
OBJDIR	$(TRDIR)/obj/$(MODNAME)	Directory to store intermediate files during the build process (includes object files, generated source code).
PREFIX	/opt/$(APPNAME)_$(VERSION)	Default installation dir prefix for the install make target.
PUB_H		List of C/C++ header files stored in src directory to be published. Note: ABS standard layout request to store such files in include directory aside src directory. This feature was introduced only to ease the integration of legacy non compliant code into an ABS project.
RMODDEP	1	Maximum recursion level for deep inter module dependencies checking.
RUNARGS		Arguments to be forwarded to the application execution when invoking make run, make debug, make check or make debugcheck.
SYSNAME	output of make/OS.sh	Target operating system. May include a version number (ex: Debian_8, CentOS_7)
TRDIR	$(PRJROOT)/build/$(MODE).$(ARCH)	Root directory for any built artifact storage.
USEJAR		List of external dependencies for a java package. Similar to USELIB but dedicated to java. For each entry, the package is downloaded from $(LIB_REPO)/noarch/<entry name>.jar
USEJMOD		List of internal dependencies for a java package. Similar to USEMOD but dedicated to java.
USELIB		List of external dependencies. For each entry, the package is downloaded from $(LIB_REPO)/$(ARCH)/<entry name>.tar.gz
USELKMOD		List of internal dependencies towards linux kernel modules.
USEMOD		List of internal dependencies.
VABS		ABS version to use.
VERSION		Application version. Version number defines the current development increment. The value is finally the next tag name for the current branch.
VISSUE		Number of the Jira issue used to track the current branch activity. Used for release note processing.
VPARENT		Parent version number. The version used to create the branch of the current version, or the previously released version in the same branch. Used for release note processing.

5 Requirement references

Requirement	Referenced by
abs.lang.1	assert 3.2.1.2
abs.lang.2	assert 3.2.1.3
abs.lang.3	assert 3.2.1.4
abs.dep.1	assert 3.2.1.5

Requirement	Referenced by
abs.run.1	assert 3.2.3.1
abs.run.2	assert 3.2.3.3
abs.run.3	assert 3.2.3.4
abs.run.4	assert 3.2.3.6
abs.cm.1	assert 3.2.1.6
abs.cm.2	assert 3.2.1.6
abs.cm.3	No reference.
abs.cm.4	§2.2.3
abs.arch.1	check 3.2.2