

Requirements for Software Product Building and Management

Revision 2.1

Contents

1	Introduction	4
1.1	Purpose	4
1.2	Overview	4
1.3	Scope	5
1.4	Terminology	5
1.5	How to read the requirements	7
2	Stakeholders	7
3	Roles	7
4	General requirements	8
4.1	Behavioral requirements	8
4.1.1	Ability to override certain products from a known release	8
4.2	Constraints	8
4.2.1	Supported operating systems	8
4.2.2	Shell support	8
5	Requirements for product management	9
5.1	Description	9
5.2	Behavioral requirements	9
5.2.1	Multiple versions	9
5.2.2	Multiple ABI variants	9
5.2.3	Multiple sets of optional components	9
5.2.4	Exclusivity of active products	10
5.2.5	Consistency of active products	10
5.2.6	Products must be installable without system privileges	10
5.2.7	Installation must not specify a specific mount point	10
5.2.8	Management of product dependencies	10
5.2.9	Ability to use products installed privately, simultaneously with other products	11
5.2.10	Reporting on active or available products	11
5.2.11	Umbrella products	11
5.2.12	Registration of non-relocatable products	11
5.2.13	Removal of products	11
5.2.14	Automatic platform selection	11
5.3	Performance requirements	11
5.3.1	Ability to install pre-built products	11
5.3.2	Non-duplication of already-installed products	12
5.3.3	Speed of setup	12

5.3.4	Shell responsiveness	12
5.4	Constraints	12
5.4.1	Simultaneous active products	12
6	Requirements for product development	12
6.1	Description	12
6.2	Behavioral requirements	13
6.2.1	Support for out-of-source builds	13
6.2.2	Support for all standard build targets	13
6.2.3	Support for the <i>default</i> build target	13
6.2.4	Support for the <i>test</i> build target	13
6.2.5	Test output must be brief and clear	14
6.2.6	Access to full test output	14
6.2.7	Support for the <i>install</i> build target	14
6.2.8	Support for the <i>package</i> build target	14
6.2.9	Support for the <i>clean</i> build target	14
6.2.10	Support for the <i>help</i> build target	14
6.2.11	Specification of non-system compilers	14
6.2.12	Build verbosity	14
6.2.13	Specification of configuration for ABI variants	15
6.2.14	Single-location specification of options for ABI variants	15
6.2.15	Ability to add new supported languages	15
6.2.16	Ability to add new supported code generators	15
6.2.17	Configuration of particular build operations	15
6.2.18	Automatic rebuild	15
6.2.19	Test dependencies	15
6.2.20	External dependencies	16
6.2.21	Ease of use of external dependencies	16
6.2.22	External dependent products	16
6.2.23	Specify different dependent product versions by variant	16
6.2.24	Ease of specification of targets	16
6.2.25	Programmatic identification of library versions	16
6.2.26	Identification of unofficially-built code	17
6.2.27	Partial builds	17
6.2.28	Cross-compilation	17
6.3	Performance requirements	17
6.3.1	Parallel build capability	17
6.3.2	Parallel test capability	17
6.4	Constraints	17
6.4.1	Integration with product management system	17
6.4.2	<i>art</i> suite support	18
6.4.3	Language support	18
6.4.4	Code generator support	18
6.4.5	Product building capability	19
6.4.6	Automatic dependency determination	19
6.4.7	Linking dynamic libraries	19
7	Requirements for product integration	19
7.1	Description	19
7.2	Behavioral requirements	19

7.2.1	Build multiple products	19
7.2.2	Build integrity	20
7.2.3	Check for missing products	20
7.2.4	Identify dependent products	20
7.2.5	Check out products from SCM systems	20
7.2.6	Add support for new SCM systems	21
7.2.7	Dependency version management	21
7.2.8	New product skeleton	21
7.2.9	Multiple integration builds using single product source area	21
7.2.10	Identification of product build area	21
7.2.11	Identification of product sources used	21
7.2.12	One-step invocation for common tasks.	21
7.3	Performance requirements	22
7.3.1	Parallel operation across products	22
7.4	Constraints	22
7.4.1	Integration with product development system	22
7.4.2	Integration with product management system	22
7.4.3	Support checkout from named SCM systems	22
7.4.4	No constraints on origin of product source	22
7.4.5	No constraints on use of product development system	22
7.4.6	No constraints on product variant selection	22
8	Requirements for build tool for release managers	23
8.1	Description	23
8.2	Behavioral requirements	23
8.2.1	Product build instructions must be self-contained	23
8.2.2	No constraints on product build system	23
8.2.3	Products specify only direct dependencies	23
8.2.4	Bulk building	23
8.2.5	Umbrella products	23
8.2.6	Forced rebuilds	23
8.2.7	Automatic rebuild of dependent products	24
8.2.8	Updating versions of dependent products	24
8.2.9	Multiple ABI variants	24
8.2.10	Flexibility between ABI variants and optional component sets	25
8.2.11	Single authoritative specification of dependencies	25
8.2.12	Production of distribution manifests	25
8.3	Performance requirements	25
8.3.1	Partial rebuilding	25
8.3.2	Efficient use of multi-core build machines	25
8.4	Constraints	25
8.4.1	Build for all officially-supported platforms	25
8.4.2	Integration with product management system	26

1 Introduction

1.1 Purpose

The Fermilab Scientific Computing Division's Scientific Software Development Group has organizational goals of developing and supporting the software infrastructure relied upon by current and planned Intensity Frontier experiments, as well as Cosmology experiments and projects. The group has a working set of tools for *product management*, *product development*, *integration of multiple products*, and *building of software for release*. The requirements contained here summarize essential functionality that is currently used or known to be needed by supported experiments and other users.

1.2 Overview

The *Product management system* (section 5) makes software available to end-users. We call the atomic unit by which such software is versioned and delivered a *product*. The product management tools allow users to select which versions of which products are used together, and help prevent the simultaneous use of incompatible products.

The *Product development system* (section 6) is used by the developer of a product; in particular, it is used to build, test, and package the code for use by others as an external product.

The *Product integration system* (section 7) is used by developers of a suite of related products, which must be built in concert as a consistent whole, avoiding inconsistencies, and without undue inconvenience.

The *Software release build system* (section 8) is used by release managers to create coherent sets of products of known version that can be managed by the product management tools. The source and the build system of each product may or may not be under the control of the people doing the packaging. The software release build system must build the product using its own build system in a manner compatible with the behavior of the product management tools (section 5).

While the development and integration systems described above may do part of the task of the software release build system, the focus is different and the features of each system reflect that. The development and integration systems provide fast cycle-time, highly flexible targeted builds with careful management of both inter- and intra-product dependencies for interactive development of code. The software release build tool, however handles only inter-product dependencies and produces coherent suites of related software from known versions regardless of the source of that software or the build system it uses.

For example, a *LArSoft* release will include *GCC*, *Boost*, *SQLite* and other products whose source and build systems are not under the control of the release manager or local developers (for which the product management and release build systems are relevant). However, it will also include a particular version of the *art* suite and, of course, the locally developed and controlled *LArSoft* products. A *LArSoft* developer may have a software

development task which encompasses only one package (say, *larcove*) and likely will not wish to have local copies of everything else; the product management and development systems are applicable here. If the developer's task encompasses multiple packages (say, he wishes to integrate his *larcove* changes into the products that use it), then the product integration system also comes into the equation.

1.3 Scope

The scope of the requirements described in this document is the software developed by and for the stakeholders listed in section 2, and the products relied upon by that software.

This document does not cover the topic of product deployment.

This document does not address the issue of naming conflicts between libraries, executables and possible other entities in different products.

1.4 Terminology

ABI: An *application binary interface* (ABI) is the low-level interface between two program modules, and determines such details as how functions are called and in which binary format information should be passed from one program component to the next, relating to how a binary product was built. For C++ and Fortran software, the compiler version and the operating system version are critical for ABI compatibility. For Python software, byte-code compiled libraries are compatible across operating systems, but not generally across Python interpreter versions. Often, ABI mismatches can lead to linking failures. Sometimes they lead to more subtle errors, and are very difficult to diagnose.

API: An *application programming interface* (API) is the source-code level interface between two program modules. It is related to source code version. API mis-matches typically lead to compilation failures.

active product: An *active product* is one that is currently being used. For executables, this means they are found on the `PATH`; for libraries, that they will be loaded when needed, *etc.* Only one variant of a given product can be active at one time.

available product: An *available product* is one that can be made active using the product management tool's command(s) to activate the product. Many variants of a given product may be available at the same time.

closed link: A *closed link* is the linking of a dynamic library leaving no unresolved symbols.

evaluation of a test: *Evaluation of a test* means determining whether the given test succeeds or fails.

external product: An *external product* is one for which the builder of the product is not in control of the software or its build system.

integration build area: An *integration build area* is a collection of *build areas*, managed in coherent fashion.

locally-developed product: A *locally-developed product* is one whose source is under the control of the developer or release manager.

package (v.): To make a product ready for distribution.

product: See *software product*.

product build area: A *build area* is a directory tree into which is put the files generated by the building of *standard build targets* for a single *software product*.

product source area: A *product source area* is a directory tree containing the source code for a single software product. Such a product source area should always be under the control of a *source code management system*.

release: A particular coherent set of products of known version.

setup (v.): Produce an environment in which the software can be built or in which a consistent group of products can be used.

software product: A *software product* (also shortened to *product*) is an identifiable, separately packaged body of software. It can include such software entities as libraries, executables, C and C++ headers, documentation and executables. A product is built and delivered as a unit, and is the smallest unit of versioning.

source code management system: A *source code management (SCM) system* is a system used for version control of source code. These systems are also called *source code repositories*.

standard build targets: A build system typically supports the following list of high-level targets, which we call *standard build targets*. These include:

default: The execution of targets that produce files. These include executables, libraries, and test programs. Also known as the *build stage* or (for historical reasons), the *all* target.

test: Runs user-provided integration and unit tests.

install: Installation of executables, libraries, headers, *etc*, in the appropriate location(s).

package: The production of an “installation kit,” suitable for distribution of the build products, to be used without requiring the ability to build the product.

clean: The removal of produced files.

help: List available individual component targets, such as libraries, executables and object files.

target: A *target* is an identifiable sub-component of a development build procedure that usually (but not always) corresponds to a generated file. Examples would be a particular executable, library or test.

umbrella product: An *umbrella product* is a software product that contains only dependencies on other products, and has no additional code, libraries, data files, or executables of its own.

user: The generic term *user* refers to a person fulfilling any of the roles defined in section 3.

variant: A *variant* of a product is a particular version and built configuration of a product. Here, “built configuration” could include such attributes as the platform, compiler used, debug or optimization level, or optional feature set.

version: The *version* of software specifies the source code text of the software. It determines the API to which users of a product program, *e.g.* the number and types of the arguments for a specific function.

1.5 How to read the requirements

Requirements are grouped by system (as specified in section 1.2) and type. The types are *behavioral*, *performance*, and *constraint*. Each subsection within a type typically contains one requirement. The name of the subsection is the name of the requirement. The first paragraph of each of these subsections describes the requirement. The following paragraphs, if any, provide examples and further supporting information about the requirements. An example of this format may be seen clearly in requirement 5.2.1.

For performance requirements, it is assumed that the average load per core on the test machine is not greater than 1.0. Furthermore, it is assumed that when network filesystems or any network attached storage is used for product management, the underlying network resources are not oversubscribed or under-engineered for normal interactive or batch system use during evaluation or testing of systems claiming to meet the contained requirements. Quantitative numbers for network resources needs can be provided if necessary.

2 Stakeholders

Stakeholders include:

1. The SSD Group, who deliver software used by many independent experiments and projects.
2. The experiments and project groups using the software delivered by the SSD Group. These include:
 - Experiments from the Intensity Frontier:
 - **NO ν A**
 - **Mu2e**
 - **Muon g-2**
 - **μ BooNE**
 - **ArgoNeuT**
 - Other experiments and projects:
 - **Darkside/ DS50**
 - **CosmoSIS**
 - Developers of toolkits and products used by multiple experiments:
 - *Nutools*
 - *LArSoft*
 - *artdaq*
 - *IFDH*
 - *artG4*
 - *ifbeam*

3 Roles

The roles interacting with all systems described herein include:

1. *Scientist-developers* (usually shortened to *developers*), who develop and use code from their experiments. They also rely on software products delivered through the product management tools.
2. *Product suppliers*, who make external software products available through the product management tools.
3. *Release managers*, who produce official releases of experiment software that relies upon products available through the product management tools.
4. *Release installers*, who are responsible for installation and maintenance of a set of installed products for a given set of computers.

4 General requirements

All systems specified in this document shall satisfy the following requirements.

4.1 Behavioral requirements

4.1.1 Ability to override certain products from a known release

The system shall provide the ability to override by local compilation a number of products from installation areas containing consistent releases, while utilizing other pre-compiled products from elsewhere, subject to binary compatibility constraints. See specific requirements [5.2.9](#), [5.3.1](#), [7.2.1](#) and [7.2.2](#).

4.2 Constraints

4.2.1 Supported operating systems

Platform support for specified systems shall include:

1. Scientific Linux 6, and compatibles (*e.g.* SLF6, RHEL6, CentOS6).
2. Scientific Linux 5, and compatibles (*e.g.* SLF5, RHEL5, CentOS5).
3. Mac OS X “Mountain Lion” (10.8, Darwin 12).
4. OS X “Mavericks” (10.9, Darwin 13).

Future platforms are expected to include SLF7 (and related RHEL derivatives), Ubuntu 12 and related derivatives (*e.g.* Mint), and newer versions of OS X.

4.2.2 Shell support

The integration system shall operate with full functionality when invoked from the following shells:

- *bash*
- *tcs*
- *zsh*

The minimum version of each shell that shall be supported shall correspond to that found on RHEL5 and associated systems.

5 Requirements for product management

5.1 Description

A *product management system* makes products available for use and manages which products are active at any time. The use described here includes running executable programs and providing libraries, headers, and modules necessary for the development of other tools and executable programs. A key feature of this system is the ability to manage a versioned collection of products, in which each product has a specific version. The system must allow all the products in the collection to be made active using the name, version, and (possibly null) variant names assigned to the collection.

5.2 Behavioral requirements

5.2.1 Multiple versions

It must be possible to have available multiple *versions* of the same product. The version of a product specifies the source code text used to build the product; a different source code text will get a different version number.

When a developer is working on adapting his own code to deal with a new version of an underlying product, it is important to be able to allow the use, on the same machine, of both the “old” and “new” versions of the underlying product.

When a product supplier is building and testing a new version of a product, it is important to conveniently switch from one version of the product to another.

5.2.2 Multiple ABI variants

It must be possible to have available multiple *ABI variants* of the same version of a product. An ABI variant specifies how a product was built in any and all ways that determine binary compatibility.

Developers do not want to waste time (their own, or that of the product suppliers that support them) debugging issues related to binary incompatibility. Such issues are especially subtle for C++ and Fortran code.

Developers want to be able to move easily from using a “release quality” build to a “debug quality” build of underlying software.

A release installer must be able to support multiple experiments or projects or both, and so to make available the ABI variants required by each supported experiment or project.

5.2.3 Multiple sets of optional components

It must be possible to have available multiple sets of *optional components* for the same version and ABI variant of a product. Examples include whether or not *RooFit* is included

in the build of *ROOT*, and whether the MPI library used in an application is *MPICH* or *MVAPICH*. Note that in the latter case, the different choices of optional component are mutually exclusive.

Release managers want to have the ability to decide, for their experiments, which set of dependencies will be used, *e.g.* what MPI implementation shall be relied upon, or whether their *ROOT* version will depend on *Geant4*.

5.2.4 Exclusivity of active products

The product management system shall ensure that only one version and variant of a product shall be active at any time in a particular environment.

5.2.5 Consistency of active products

The product management system must help assure that only consistent sets of products are made active, and that an attempt to make active an inconsistent set of products is stopped, with the user informed of the reason for the error.

Developers do not want to waste time (their own, or that of product suppliers that support them) debugging issues (either build issues or software behavior issues) resulting from the use of incompatible products.

A release manager responsible for physics verification and validation of a given release must be certain that the expected set of products is being used.

5.2.6 Products must be installable without system privileges

The product management system must not require system privileges for the installation of products. Not all users (especially not all scientist developers) will have system privileges on the machines on which they work. For some university groups, especially those using shared university resources, the release installer may not have system privileges.

5.2.7 Installation must not specify a specific mount point

The product management system must not require that the software be installed at a particular mount point. Requirement of a specific mount point may clash with an already-established mount point, and may not be possible without system privileges.

5.2.8 Management of product dependencies

The command to make a product active must make active all necessary supporting products. It must make active the correct version, ABI variant, and set of optional components.

The user of a product should not be required manually to activate the full set of dependencies of a product in order to use it.

5.2.9 Ability to use products installed privately, simultaneously with other products

It must be possible for the user to replace a product with a different but compatible build of the same product, visible to no-one else, while still using other products, not privately installed.

The developer must be able to explore the use of new software, including new versions or new builds of existing products, without disrupting the work of others.

5.2.10 Reporting on active or available products

It must be possible for a user to ask for the list of currently active products. It must be possible for a user to ask for the list of currently available products. It must be possible for a user to request the list of products that a product depends upon.

5.2.11 Umbrella products

The user shall have the ability to define a product that consists only of dependencies. When this product is made active, it causes other products to be made active.

5.2.12 Registration of non-relocatable products

On the rare occasion that it might be necessary, a user shall be able to make available a product which cannot be moved from its original installation location.

An example would be a proprietary driver that may be non-redistributable due to license or non-relocatable due to limitations in the code or build system. Some external software products, for example, have their installation location compiled in to allow them to access support files (`/usr/share` or equivalent, say).

5.2.13 Removal of products

A user of the product management system with appropriate privileges shall be able easily to remove a product from the filesystem without disturbing other products or other versions or variants of the same product. The user shall not have to specify individual files.

5.2.14 Automatic platform selection

The product management system, when making a particular product active, shall not require the user to specify the platform: the system should select automatically the product matching the current platform unless specified otherwise.

5.3 Performance requirements

5.3.1 Ability to install pre-built products

The system must not require the in-place building of all products. Many individual products take a considerable amount of time to build; *e.g.* it takes much longer to build GCC than to install from a pre-built binary distribution.

All stakeholders currently have the ability to use pre-built distributions, and most want to retain it.

5.3.2 Non-duplication of already-installed products

It must be possible for a user that has already installed a given ABI variant of a given product to use that already-installed software, rather than requiring the installation of a (redundant) product in another location.

Several institutions in the stakeholder communities have disk space issues due to shared resources. Experiments also wish to have multiple releases on hand simultaneously but do not wish to have separate copies of the same possibly large products in each release. One-point maintenance for a particular product variant is also desirable.

5.3.3 Speed of setup

For an interactive system, when used with a product tree of 100 products, it must take no more than 10 seconds to make the specified set of products active. Under batch processing scenarios, the setup time may not take a majority of the initialization time.

Products are known to be available through the following resources:

1. AFS,
2. CVMFS,
3. NFS, and
4. locally installed products.

For each of the file access facilities above that requires network transfers (all except local disk), the available systems must be able to operate at sufficiently high bandwidth, such that the network does not take a majority of the setup time.

5.3.4 Shell responsiveness

For an interactive system, when used with a collection of 100 active products, the system must not make executing common commands (*e.g.* 1s) more than one second slower.

5.4 Constraints

5.4.1 Simultaneous active products

The product management system shall support at least 100 simultaneous active products without exhausting shell resources with supported shells on supported platforms.

6 Requirements for product development

6.1 Description

A *product development system* is used to build, test, install, and package software. *Building* in this case typically means compilation, producing executable programs or

dynamic libraries, or both. For Python, building includes byte-code compilation of modules.

6.2 Behavioral requirements

6.2.1 Support for out-of-source builds

The development build system shall be configurable to put files generated by the build procedure into a directory tree which is distinct from the product's source directory tree.

This simplifies the job of source control for developers. It is easier also to operate on systems with disk quotas on backed-up filesystems. Also, if the source is on a slower, likely remote system, build times will be improved by out-of-source builds.

This requirement does not specify that the user should configure or execute the build from any particular directory.

6.2.2 Support for all standard build targets

The development build system shall support all *standard build targets* as defined in section 1.4).

6.2.3 Support for the *default* build target

The development build system shall provide support for the following build operations:

1. Running of the preprocessor, and saving of the output, for all languages that use a preprocessor.
2. Run code generators to generate files to be used by other targets.
3. Perform template-based text substitution using information known at build time. An example of such information is whether the build being done is “debug” or “optimized”.
4. Compilation of source code to object code for all supported languages.
5. Production of shared libraries from user-specified lists of object code.
6. Production of Fortran module files.
7. Production of executables from source code in supported languages, object code and/or shared libraries.

6.2.4 Support for the *test* build target

The development build system shall support the specification, execution and evaluation of tests. This functionality shall include:

1. The running of interpreted scripts, built executables or external programs with specified arguments.
2. The evaluation of a test based on its exit code.
3. The evaluation of a test based on presence of an expression in the output.
4. The ability to chain several tests, with one test's output forming another's input.

6.2.5 Test output must be brief and clear

The output written to the terminal by the *test* target must be brief, so that failed tests are obvious.

6.2.6 Access to full test output

Notwithstanding requirement 6.2.5, the development build system shall save and provide access to the full output for all tests in order to avoid having to re-run them to examine details.

Intermittent failure of tests is not uncommon, either due to mis-specification of dependencies between tests, or other factors such as load-dependent timeout failure. If the details of the original failure are not preserved, it could take some time to reproduce the problem.

6.2.7 Support for the *install* build target

The system shall support the installation of a product into a developer-specified area distinct from the defined build area consistent with the product management facilities described in section 5.

6.2.8 Support for the *package* build target

The system shall support the production of a product installation entity consistent with the product management facilities described in section 5.

For example: if the product management system is *UPS*, then this would be a product installation tar file.

6.2.9 Support for the *clean* build target

The build development system shall on demand remove files produced by the *default* and *test* targets.

6.2.10 Support for the *help* build target

The build development system shall on demand list the targets available in the current context.

6.2.11 Specification of non-system compilers

The system shall enable the building of the product with a compiler other than the system compiler.

6.2.12 Build verbosity

By default, the system shall not display the individual commands used to build each target. However, the developer shall be able to specify that the system show such detailed information.

Under normal circumstances, this level of detail is unnecessary and detracts from the developer's ability to spot problems building targets. However, being able to see exactly which commands are executed becomes important once problems are encountered.

6.2.13 Specification of configuration for ABI variants

The system shall support the command-line selection of one of a set of configurations of ABI variants, (*e.g.* debug or release-quality), upon establishment of the build area. See also requirement [5.2.2](#).

6.2.14 Single-location specification of options for ABI variants

The system shall enable the specification of configuration options by ABI variant for each build operation for all operations of that type.

For instance, one should not have to specify `-g -O0` for each and every debug compilation of a C or C++ source file with *GCC*.

6.2.15 Ability to add new supported languages

The system shall provide the ability to add support for a new compiled language without modification to the core build system.

6.2.16 Ability to add new supported code generators

The system shall provide the ability to add support for a new code generator without modification to the core build system, integrating its products into the build procedure.

6.2.17 Configuration of particular build operations

Developers must have the option of overriding the default set of options for the building of specific build targets.

For example, some source code units fail to compile with `-O3` optimization; the developer must be able to specify the use of some other optimization flag for that compilation unit.

A second example is the need to specify a preprocessor defined symbol for a particular compilation unit.

6.2.18 Automatic rebuild

The development build system shall automatically determine which build targets are out of date with respect to their dependencies and rebuild them as appropriate.

This includes implicit dependencies, such as recursively included headers in C and C++. This is the sort of thing typically done by *makedepend*. It also includes explicit dependencies such as libraries.

6.2.19 Test dependencies

The developer shall be able to specify that one test depends upon the result of another, and must therefore be executed first.

6.2.20 External dependencies

The developer shall be able to specify external dependencies (*e.g.* libraries, executables) from other products than the one being built that the development build system shall ensure are satisfied in order to build the product.

6.2.21 Ease of use of external dependencies

The system shall provide a way for a developer to use the code in other products without having to specify the compiler options explicitly for that code and its dependencies, if any.

In order to use code from *ROOT*, which may in turn use *Geant4* (among other things) a developer should not have to specify all the `-I`, `-L`, `-l` (and possibly `-D...`) required by *ROOT*, *Geant4* and the rest of their dependencies. If the development system were to leverage *CMake*, this functionality would be provided by leveraging the `config.cmake` files and the `find_package` facility.

6.2.22 External dependent products

The developer shall be able to specify dependencies for a product by product, version and variant.

For example: the release-quality build of *art* using *GCC* 4.8.2 to the C++2011 standard requires as a dependent product *ROOT* 5.34.18 built the same way. Similarly, the debug-quality build of *art* will require the debug-quality build of *ROOT*. See also requirements [6.2.23](#) and [6.4.1](#).

6.2.23 Specify different dependent product versions by variant

The developer shall be able to specify different versions or variants of a dependency differently according to the particular variant of the product being built. This shall include being able to specify a dependent product for one variant, but not at all for another variant.

For example: the *art* developer may need to provide a variant of version 1.10.03 built with *GCC* 4.8.2 and additionally a variant built with *GCC* 4.9.1.

See also requirements [6.2.22](#) and [6.4.1](#).

6.2.24 Ease of specification of targets

It shall be easy for the developer to specify targets, both of types built into the development build system, and those added by the developer or his organization. It is a specific requirement that targets with multiple sources shall be specifiable without naming every individual source (*e.g.* file wildcards for library constituents).

6.2.25 Programmatic identification of library versions

The development build system shall mark each dynamic library with its product version in such a way that the version may be obtained programmatically.

A developer will find this information useful while debugging new developments to one or more products that may be part of a coherent whole. Implementation of this requirement is also a prerequisite for implementation of requirements [6.2.26](#) and [7.2.2](#).

6.2.26 Identification of unofficially-built code

The development build system shall provide the means to distinguish programmatically between libraries built unofficially from possibly uncontrolled sources and those packaged for release as a tagged version of the product.

Experiments need to be assured that a production release is built from only sanctioned sources.

6.2.27 Partial builds

The developer shall be able to build of one or more targets of the product as specified on the command line, with associated dependent targets being built automatically, without having to configure or build the whole system.

Examples would be a particular executable, library, generated code item, preprocessed source or the execution of a particular test.

6.2.28 Cross-compilation

The development build system shall support the use of cross compilers to produce a product suitable for use on an architecture different from that used to produce the compiled product.

6.3 Performance requirements

6.3.1 Parallel build capability

The development build system shall reliably execute as many build tasks in parallel as are compatible with the requested parallelism and the interdependencies of the individual targets.

6.3.2 Parallel test capability

The development build system shall reliably execute as many tests in parallel as are compatible with the requested parallelism and the interdependencies of the individual targets.

6.4 Constraints

6.4.1 Integration with product management system

The development build system shall utilize the product management system for the purposes of making particular versions and variants of dependent products active for a build.

See requirements [6.2.22](#) and [6.2.23](#).

6.4.2 *art* suite support

The development build system shall build items known to be required by *art* and other products according to defined naming conventions. These items are:

- *ROOT* dictionaries (identified by `_dict` name suffix),
- *art* and *artdaq* plugins (identified by `_module`, `_service`, `_source` and `_generator` name suffixes),
- *SMC* state charts (file extension `.smc`).

Currently several types of files requiring special compilation rules or code generators are identified through naming conventions. The *art* framework allows the user's data model to be extended using the *ROOT* dictionary generators. There are several different plugin types within *art* and *artdaq*: modules, services, input sources and generators. Information is added to the resulting shared libraries to locate them at runtime and also to discover their entry points.

6.4.3 Language support

The development build system shall support the compilation of source code for the following languages:

- C.
- C++.
- Fortran.
- CUDA.
- Python: interact with the official `distutils` system to build and install correctly constituted python packages, modules and scripts, including those which provide python language extensions and interface with non-Python code and libraries.

6.4.4 Code generator support

The development build system shall support use of the following code generators, integrating their products into the build procedure:

- *flex/lex*.
- *bison/yacc*.
- *SMC*¹.
- *moc*².
- *rootcint*, *genreflex/GCC-XML* and *cling* (*ROOT* dictionaries).
- *swig*³.

¹<http://smc.sourceforge.net>.

²The Qt meta-object compiler. See <http://qt-project.org/doc/> for details

³<http://www.swig.org>.

6.4.5 Product building capability

Products built by the build system shall be consistent with the product management facilities described in section 5.

Since the development group is a user of the development environment, it is critical that the development environment support the building of the products that we deliver through the product management system.

For instance: the requirements that products be relocatable (5.2.6, 6.2.3, 5.3.1) has implications for the build system that must be taken into account.

6.4.6 Automatic dependency determination

The development build system shall be able to automatically determine the source code dependencies for the following languages:

- C
- C++
- Fortran

6.4.7 Linking dynamic libraries

The development build system shall support closed links. The default build of a dynamic library shall be closed.

Failure to create closed links delays the time until some types of build error are discovered. This makes fixing these errors more difficult.

7 Requirements for product integration

7.1 Description

A *product integration system* coordinates the simultaneous development and building of multiple products, performing software development tasks involving one or more of the coordinated products. This system references a collection of *product source areas* and permits building these products into a *product build area*. Typically the collection of products will form a chain of dependencies. This system ensures that changes in one product will automatically cause targeted rebuilding in dependent products, maintaining a consistent whole. A developer will typically go through a development and test cycle for one or more software features across a set of related products operating within one build area.

7.2 Behavioral requirements

7.2.1 Build multiple products

The integration system shall be capable of orchestrating the build and installation of one or more products. This includes correct dependency handling between products. This ability must extend through layers of dependencies.

Within the context of *art*, a change in *cetlib* that affects *fhicl-cpp*, and therefore affects *art* must cause all affected files to be rebuilt in all affected products. For example: within the *art* suite, *art* depends on *fhicl-cpp* both explicitly and via *messagefacility*. A developer may wish to make changes in *fhicl-cpp* and see the effects in the *art* framework. It is the job of the integration system to calculate the pieces of *messagefacility* and *art* that are affected by the *fhicl-cpp* changes and automatically rebuild them.

7.2.2 Build integrity

The integration system shall fail to setup an inconsistent set of products. This failure shall include a diagnostic useful in aiding the developer to isolate and resolve the source of the inconsistency.

It is necessary to ensure the integrity of the build against inconsistent use of headers and libraries from products built within and without the integration system, as inconsistent build are **always** wrong, even if the undefined behavior so produced occasionally matches the desired behavior. Difficult to trace memory errors and bad results are the far more common results of such inconsistencies. Experimenters and *art* experts have wasted time and experiments have wasted data due to inconsistent builds.

As an example: *art* depends on *messagefacility*, which depends on *fhicl-cpp*. An attempt to build *art* and *fhicl-cpp* without also building *messagefacility* **must fail**.

The development system requirements [6.2.25](#) and [6.2.26](#) are relevant to implementation of this requirement.

7.2.3 Check for missing products

The integration system shall provide a tool for a user to identify which products from a named release must be added to those orchestrated currently in order to guarantee a consistent build.

This requirement implies a user-invocable tool which provides the functionality which must be present to implement requirement [7.2.2](#).

7.2.4 Identify dependent products

The integration system shall provide a tool for a user to identify which products from a named release not being orchestrated currently by the integration system may be obsoleted by changes to the products currently being orchestrated.

A developer making local changes to products might want to be aware of which (if any) products not currently being orchestrated would need to be updated in order to take account of those changes.

7.2.5 Check out products from SCM systems

The integration system shall provide the facility to check out a particular version (or branch, if applicable) of a product's source from SCM systems using a single SCM system-agnostic command.

7.2.6 Add support for new SCM systems

The integration system shall support the addition of new SCM systems without having to alter the core system.

7.2.7 Dependency version management

The integration system shall enable the user to update the version number of a product, and update the use of that product in products requiring it as a dependency while ensuring consistency per requirement [7.2.2](#).

The manual editing of multiple version specifications is error-prone; the system must provide the facilities to reduce the incidence of such errors. Note that the version number of a product is specific to that product, and does not necessarily bare any relation to the versions attached any releases of which it may be a part.

7.2.8 New product skeleton

The integration system shall create a new product directory structure on demand consistent with the structure required by the product development system.

This functionality should provide for the simple insertion of the new product into an existing empty remote repository for any supported SCM system.

7.2.9 Multiple integration builds using single product source area

The system shall allow a single product source area to be used simultaneously by two or more distinct integration builds.

For example, it is useful to be able to make both release- and debug-quality builds of the same set of product sources. Using exactly those same sources rather than copies thereof ensures synchronization between the different qualities of build.

7.2.10 Identification of product build area

The system shall provide the means to identify the particular build area used for a given integration build.

7.2.11 Identification of product sources used

The system shall provide the means to identify the product sources used for a given integration build.

7.2.12 One-step invocation for common tasks.

The product integration system shall provide one-step invocation of tools to execute the following common tasks:

1. Setting up for development.
The integration system shall initialize for a new build with a single command.
2. Checking out products from known locations without specifying full URLs.
See also requirement [7.2.5](#).

3. Executing standard build targets.
See also requirements [7.4.1](#), [7.4.5](#) and [6.2.2](#).
4. Packaging one or more products.
See also requirement [7.4.2](#).

7.3 Performance requirements

7.3.1 Parallel operation across products

The product integration system shall, when appropriate, execute build operations in parallel across products.

7.4 Constraints

7.4.1 Integration with product development system

The product integration system shall work with products utilizing the product development system (section [6](#)). It shall not be required to work with any other build system.

7.4.2 Integration with product management system

The integration system shall utilize the product management system to ensure the availability and consistency of external dependencies of all products being orchestrated currently by the integration system.

7.4.3 Support checkout from named SCM systems

The implementation of requirement [7.2.5](#) shall support both *Git* and *Subversion* systems, and conveniently support the common case where these repositories are hosted by FNAL's *Redmine* system.

7.4.4 No constraints on origin of product source

The integration system shall place no restrictions on the origin of any product sources currently orchestrated: any mix of sources from any origin is permissible, including any SCM, no SCM or symbolic links.

7.4.5 No constraints on use of product development system

The user of the integration system shall continue to be able to make full and unrestricted use of the product development system with respect to any individual product being orchestrated currently by the integration system. This shall include, but is not limited to being able to compile, build or test individual components of such a product.

7.4.6 No constraints on product variant selection

The integration system shall not restrict the selected variants of the products being built except to ensure that they are consistent.

The subject of selected variants is discussed in requirements [5.2.1](#), [5.2.2](#), [5.2.3](#), and [8.2.9](#).

8 Requirements for build tool for release managers

8.1 Description

The *build tool for release managers* is used for the building, installation, and packaging of specified versions of multiple products. These may be external products or locally developed products.

8.2 Behavioral requirements

8.2.1 Product build instructions must be self-contained

The user must be able to describe, as a self-contained entity, the instructions for building a particular product; the system must be responsible for locating and executing the build instructions for necessary dependencies.

8.2.2 No constraints on product build system

The system for coordinating and controlling the release build of software products shall be agnostic with respect to the build system utilized by any one product, and not restrict the choice thereof.

8.2.3 Products specify only direct dependencies

The configuration for a product shall require specification of only direct dependencies.

The user should not have to further specify the full list of implied dependencies, otherwise there is scope for inconsistency and error.

8.2.4 Bulk building

The user must be able to specify a group of not-necessarily-related products to build as a set, and to build a series of multiple products in the correct order as specified by their dependencies.

8.2.5 Umbrella products

A user must be able to cause the system to construct and package an *umbrella product*, as understood by the product management system (see requirement [5.2.11](#)).

8.2.6 Forced rebuilds

The system must be capable of rebuilding a particular product, checking for the presence of dependencies and (at the user's option) either recursively building those dependencies or failing due to missing dependencies.

8.2.7 Automatic rebuild of dependent products

The system shall provide the option of automatically rebuilding (or not) products which depend on a product which has been rebuilt.

A release manager may wish to rebuild a product upon which other products may depend. This may or may not imply that products depending upon it should be rebuilt, depending on the particular reason for the rebuild. Improving the robustness of the build procedure for a particular product would not require rebuilding dependent products, for example; adding a necessary patch would.

8.2.8 Updating versions of dependent products

The user must be able to easily update version numbers of dependencies of an updated product, within a defined, coherent set, when required.

For example, when the version of SQLite is updated, it must be easy for the user to update the versions of all products that depend upon SQLite (Python, and everything that depends upon Python).

The more manual steps involved in this process, the more there is scope for inconsistency and error.

8.2.9 Multiple ABI variants

The system must be capable of building multiple named *variants* of the same product and version (compiler, feature set, *etc.*), with dependencies specified including version and variant. It should not be assumed that binary-compatible products will have identical variant names.

Note that this requirement implies the ability to control what compiler and what version of that compiler is used, including the use of cross-compilers.

Products, depending on their source language, may have different dependencies on a particular compiler or set of options (C++2011, for example). A given set of products (*e.g.* for **Mu2e**) may have different variants (*e.g.* debug *vs.* prof, GCC 4.8.1. *vs.* GCC 4.8.2 *vs.* ICC) that should be propagated down through dependencies.

As an example of consistency and compatibility of products, a C++-based product with a variant label e5 may rely upon a build of a C-based product with a null variant identifier.

As a further example, the *CodeSynthesis XSD* product requires access at build-time to *Boost* and *xerces-c* products which have been built using a particular version of *GCC* to the C++2003 standard. Other products in the release require (both at build and use-time) those same products built to the binary-incompatible C++2011 standard. There is no inconsistency here as *CodeSynthesis XSD* consists only of programs, not libraries and therefore with static linking, the C++2011 versions of *Boost* and *xerces-c* can be used simultaneously with *CodeSynthesis XSD* without causing inconsistencies.

8.2.10 Flexibility between ABI variants and optional component sets

The user must be able to describe dependencies between products that differ for different versions, ABI variants, sets of optional components or platform (see requirements 5.2.1 and 5.2.2).

8.2.11 Single authoritative specification of dependencies

The specification of dependencies must be done in only one place. The specification of what is built and installed should determine what entries are put into the product description.

If a product installs headers, the appropriate means for users to establish compiler include-flags correctly must be made.⁴

8.2.12 Production of distribution manifests

The system must be able to produce a list of the distribution package files corresponding to a specified group of products.

8.3 Performance requirements

8.3.1 Partial rebuilding

The system must be capable of skipping already-built products, and of forcing a rebuild starting at a particular point in the sequence.

8.3.2 Efficient use of multi-core build machines

The system must be able to run, in parallel, builds of non-related products. This is so that we can make efficient use of multi-core build machines.

The system must be able to take advantage of parallelism within the build of individual products, when available (*e.g.* `make -j <ncores>`). This is so that we can make efficient use of multi-core build machines.

8.4 Constraints

8.4.1 Build for all officially-supported platforms

The build system must be able to operate on all the platforms and architectures we support. This does not include running on platforms for which we will cross-compile code, but does include the ability to run on the host system from which we will do the cross-compilation.

This does *not* require the system to be able to build a product on a platform or architecture not supported by that product.

⁴For example, in UPS this is done by specifying how the `_INC` environment variable should be set.

8.4.2 Integration with product management system

The build system must be able to integrate with the product management system without undue work for the packager of a particular product.

The user should not have to specify *e.g.* product dependency in multiple places in order to inform both the build system and the product management system (section 5) of dependencies.