A Build System for Software Development in Robotic Academic Collaborative Environments

Daniele E. Domenichelli*, Silvio Traversaro*, Luca Muratore†, Alessio Rocchi†, Francesco Nori*, Lorenzo Natale*
*iCub Facility, Istituto Italiano di Tecnologia (IIT), Via Morego 30, 16163 Genoa, Italy
†Department of Advanced Robotics, Istituto Italiano di Tecnologia (IIT), Via Morego 30, 16163 Genoa, Italy
{daniele.domenichelli, silvio.traversaro, alessio.rocchi, luca.muratore, francesco.nori, lorenzo.natale}@iit.it

***Abstract*—The software development cycle in the robotic research environment is hectic and heavily driven by project or paper deadlines. Developers have only little time available for packaging the C/C++ code they write, develop and maintain the build system and continuous integration tools.**

Research projects are joint efforts of different groups working remotely and asynchronously. The typical solution is to rely on binary distributions and/or large repositories that compile all software and dependencies. This approach hinders code sharing and reuse and often leads to repositories whose inter-dependencies are difficult to manage.

Following many years of experience leading software integration is research projects we developed YCM, a tool that supports our best practices addressing these issues. YCM is a set of CMake scripts that provides (1) build system support: to develop and package software libraries and components, and (2) superbuild deployment: to prepare and distribute sets of packages in source form as a single meta build. In this paper we describe YCM and report on our experience adopting it as a tool for managing software repositories in large research projects.

***Index Terms*—Build system; software development cycle; software reuse; superbuild;**

I. Introduction

In robotic research environments the software development cycle is quite hectic and often heavily driven by project or paper deadlines. The "publish or perish" model that drives the research has a sensible impact on the re-usability of the code developed for research purpose on two different levels: on a micro scale, the research products are small tools that are not sufficiently engineered to be reused in large projects; on a larger scale, the products are large conglomerates of software with strong coupling (i.e. lots of references between files in different modules) and weak cohesion. In both cases reusing the code is not easy.

This situation is worsened by the fact that most robotics software is either developed in C/C++ or has extensive dependencies written in this languages, for which no standard cross-platform tools for packaging and distribution is available, as opposed to higher level languages.

The goal of our research is to foster re-usability and the overall quality and maintainability of software developed for academic research purposes and its build system. It is motivated by our experience in managing large C/C++ software projects in collaborative robotic research environments (consisting of several research groups based on different countries), and in particular the iCub software project[1] [1]. This article is organized as follows:

- We analyze the academic software development cycle and will point the issues that arise in this environment (Section II), emphasizing the critical role of the build system (Section III), and describing the concept of "*superbuild*" explaining their importance (Section IV).
- We will list some of the tools commonly used in software development to handle dependencies (Section V), and we will state our objectives and explain the reasons why we decided not to introduce a new tool in our workflow (Section VI).
- We will introduce YCM the set of CMake modules that we developed, and we will show how YCM superbuilds are currently being used in four different European-funded research projects by different teams, and successfully promote reuse of code and collaboration between researchers from different groups (Section VII).

II. Academic Software Development Cycle

It is an indisputable fact that lots of software created for research is wasted. Academic research products are scientific papers. Data produced using the software is more important than software itself. Researchers may know how to write code, but they are not software engineers [2] and they lack experience in coding using established best practices. They work individually most of the time and have to be at the same time developers, designers, testers, release managers and integration managers. Moreover they have academic duties such as teaching, tutoring and taking classes [3].

The continuous renewal of Ph.D. and Grad Students without experience in software development weighs heavily on the software development process. A review process, besides improving the overall quality of the code [4], could contribute to improve the developers' software skills. Unfortunately in most of the cases, this kind of process does not exist for academic software.

All these factors introduce several different effects in the software products developed for research purposes.

[1]The iCub (http://www.icub.org/) is an open source humanoid robot adopted in more than 30 research groups world-wide, whose software is the result of the work of a large community (1.8M LOC, 170 contributors and 103 developers, source: OpenHub, https://www.openhub.net/p/robotology)

On a micro scale, the research driven software development cycle heavily impacts the self contained tools, algorithms, and libraries released. The goal of the development phase is often to produce a working prototype as fast as possible, with the intention to clean up the code later. Unfortunately experience shows that the Agile software development principle known as LeBlanc's Law "Later equals never" [5], [6] is almost always true for academic software, unless some commercial interests intervene.

While the code related to the subject of the study itself might be good quality, the code surrounding it is often low quality, written in a hurry, and good practices are often neglected.

Researchers have only little time available for packaging the code they write and the build system maintenance is something that goes often beyond their skills and interests. As a result, the product might perform "well enough" for writing an article, but when this is published – due to the tight research schedule driven by projects, Ph.D. programs or paper deadlines – the author is likely to abandon the code and start working on something new. The code is never properly re-engineered, and it is hardly re-usable as a component of a larger system.

On a macro scale, the research driven software development cycle affects the whole software ecosystem around funded research projects where different researchers, teams, and institutions work remotely and asynchronously on different parts of a big project.

Trying to achieve integration and interoperability between software developed by different groups is definitely a push towards better practices, but that's not enough.

When a funded project starts, one of the first decision to take is where to store the source code, which tools will be used for development, and which operating systems, IDEs, and compilers will be supported.

Users have very different sets of skills, they work in different fields and have different roles in the organizations: students, researchers, developers, team leaders. Each of them will use the software with different goals: to produce some data for a thesis or for an article, to implement some new algorithm, to show some demo to the projects reviewers.

Research groups are usually interested in enforcing their group identity, while coordinators need to show an integrated view of the results of a project. This has an important role when choosing where people will commit their source code, data, and artifacts.

One possible approach to solve this conflict is to use a central repository, usually not "owned" by any of the research group but by the consortium behind the project. This has the important effect to give visibility to the funded project. On the other side, this imposes to take early decisions over matters such as build tools and software configuration management (SCM) to be used for the whole development process. This approach usually leads to strong coupling and circular dependencies between the components of the system, and potentially reusable code cannot easily be made available for reuse. Moreover adding or removing functionality becomes hard, and the build system is thus easily broken [7].

The pitfall in this approach is that when the funded project is over, the parts of the code produced that should be reusable are supposed to be re-engineered and moved out of the repository. Unfortunately this rarely happens, due to lack of resources and to the need to move on to new projects. If some code is required for a later project, even if just a small part, "re-usability" is achieved by depending on the whole software collection, or by duplicating the code in the new repository.

The opposite approach is to let each team use their favorite tools and to store their code using their favorite SCM and host. This approach apparently has the advantage that every group can continue its work as they did before without spending time to learn how the other groups work. Unfortunately when it is time to integrate the efforts of different groups, the effect is exactly the opposite. Setup of all the software required becomes complicated and time consuming, due to the variety of tools required, to the lack of testing and documentation [8], and to the fact that build instructions are not always complete or up to date. Similarly, updates of the whole stack are painful, sometimes dependencies are added, and the other developers are not notified, so they have to figure it out by themselves. The learning curve for newcomers is steep and it could take days before they can be operational.

Another defect of this kind of approach is that it makes harder to give visibility to the software developed for a funded project, since code and documentation are scattered around different repositories and hosts.

In this paper we suggest a hybrid approach that uses a meta-repository whose only job is to fetch external repositories, where the actual source code is stored, and build it eventually with the assistance of an external tool to handle dependencies (we called this approach "*superbuild*").

III. Build Systems in Academic Software

The build system of a software project converts the source code into libraries and executables. It has an important role in any software project [9] and it needs to evolve together with the source code. A build system that grows uncontrolled or that becomes too hard to understand and to maintain can slow down the development process [10] or even stop the developers from making any progress [6].

In industry, maintaining the build system can be a time consuming task and can take a large part of the development effort [11]. In several software projects, the task of creating and maintaining the build system is held by build and release managers, that are software engineers specialized in this role [12].

In academic software, the importance of the build system is often disregarded and the resources required to develop and maintain it are not allocated. Build systems are often blindly "copy-and-pasted" from other software, from templates or even from previous projects, that in turn might be copied from somewhere else, or generated by some tools.

This causes code duplication, and due to the lack of maintenance power, the code will often go out of sync with the origin as soon as some change is made. Also for this reason

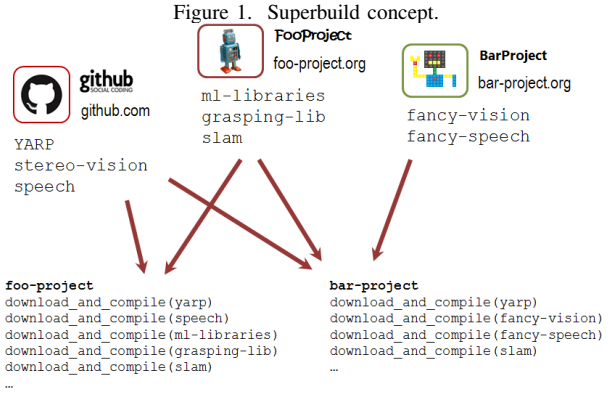
researchers ignore what their build system is doing, and do not know where to intervene when a problem arises. Finally, bad practices and lack of specific skills lead to build systems that are not optimized, and therefore slower than what they should be, wasting lots of researchers' time.

Build systems for large academic software are often complex and delicate, therefore researchers try to modify it as little as possible. When some issue arises, changes that could have been one liners, have consequences in several different places, and are likely to cause some other issues.

IV. Superbuilds in Academic Software

A superbuild is a collection of packages (sub-projects) that reside in remote repositories and are managed and built independently. The superbuild downloads and compiles all individual sub-projects using the appropriate SCM tool (git, subversion, etc) therefore the user gets working repositories in which it can develop (getting updates, making and committing changes). Superbuilds also have the role of handling dependencies and ensure that the software stack is compiled in the right order.

Figure 1. Superbuild concept.



The concept of superbuild is summarized in Figure 1. Here `FooProject` owns a repository which contains a set of packages (e.g. `ml-libraries`, `gasping-lib` and `slam`), and the `BarProject` developers host their own set of packages (e.g. `fancy-vision`, `fancy-speech`) on their own repository. Some other projects are developed by third parties and hosted somewhere else, or are already installed on the system. The `foo-project` and `bar-project` superbuilds download and compile the source code from all the required repositories. Since the superbuilds keep them under revision control, developers of both projects can easily contribute code to all the packages they use, without concerns on the repositories in which these are maintained.

We divide superbuilds in three categories.

- "Tool based superbuilds" make extensive use of external tools or scripts. An extra tool is added to the workflow and users have to learn how to use it in order to build the system.
- "SCM based superbuilds" manage the external repositories with the version control system (for example git submodules, svn externals, and mercurial sub-repositories), eventually with the aid of scripts or of the build system. These kind of superbuild have two important limitations: all the repositories must be using the same SCM, and they do not try to locate the packages on the system before downloading and building them.
- "Build system based superbuilds" manage everything using the build command, or the IDE. These superbuild use the targets of the build tool (e.g. `make`) to handle the sub-projects.

As explained in section II, several issues in the academic software development process can be solved by using superbuilds. In particular superbuilds:

- promote modularization and reuse of code;
- enforce identity of small research groups;
- give the due visibility to the collection of software for large projects;
- favor sharing of packages in code form, using SCM tools (as opposed to tar files or binaries);
- support development of unit tests for individual projects as well as the integrated solution.

All these in turn facilitate collaboration between different research groups, favoring good practices like code reviews and issue tracking.

Unfortunately setting up a superbuild is not an easy task and requires an experienced build manager and in the academic environment, the role of the build manager is often held by the researchers, that do not have the skills and the experience required for the task.

V. Related Tools

Dependency managers are widely adopted in industrial and open source software development. These can be divided in tools that download and build the source code and tools that download pre-built binary packages.

An examples of large open source projects that use an external tool to handle the build is KDE with kdesrc-build [13]. However their tools is KDE-specific, not suitable for general use.

Package managers that create packages directly from source code such as Portage [14] used by Gentoo Linux, Homebrew [15] on macOS, and recently vcpkg [16] on Windows have a similar role to these tools. Other package managers like the apt-get, yum, and pacman, on Linux distributions, and NuGet [17] and Chocolatey [18] on Windows handle binary packages. All this tools are typically platform dependent.

There are several package managers that focus on one specific programming language, and whose main purpose is to help programmers to locate libraries and programs, for example pip [19] for python, RubyGems [20] for Ruby, npm [21] for Javascript, PEAR [22] for PHP, CPAN [23] for Perl, Gradle [24] for Java, and Biicode [25] and Conan [26] for C/C++. All this tools are typically language specific.

In robotics, the issue of creating modular software and handling dependencies is particularly widespread, and consequently several tools were created by different teams, each

one with strong and weak points. Examples are qibuild [27], rosmake [28], catkin [29], ament [30] and robotpkg [31].

There are also a few attempts to create package managers using CMake [32] only, with no additional tools, for example cpm [33], cmakepp [34], and Hunter [35]. However as of late 2017, there have been limited adoption of these CMake-based package managers.

Superbuilds are used in several software projects. We examined a few superbuilds created for open source projects.

The Qt5 [36] main repository is an example of "SCM based superbuild" and uses git submodules and some script to build all the components of the library.

Most of the "build system based superbuild" that we examined are written using CMake, and are usually based on the `ExternalProject` CMake module. OpenChemistry [37], Paraview [38] and Drake [39] are examples of projects that use this approach. However, most of the superbuild infrastructures used by these projects are complex build systems, specifically tailored to this projects.

VI. Objectives and Approach

As stated before, the goal of our research is to foster reusability and the overall quality of software developed for academic research purposes and in particular in the field of humanoid robotics, by improving the quality of the build system. Compared to software development in industrial environment, the academic environment imposes some additional constraints that cannot be ignored.

We assume that the role of build manager will be held by the researchers that write the source code, and therefore they will have just a basic knowledge of the build tools and will not have time to perfect their knowledge. As a consequence, we aim at simplifying as much as possible the creation of a maintainable build system, but without hiding things or performing too much fundamental operations ("magic") behind the scenes as we believe researchers should understand how the build system works in order to be able to modify it and eventually to fix bugs.

Since people with very diverse backgrounds work in robotics, we tend see a lot of diversity also in the development environments with which people are comfortable. Researchers should be allowed to keep using their preferred workflow, and to choose the operating system and the build tools that they want to use.

Similarly, we do not want to propose practices or recommendations, firstly since enough has been already written on the matter (for example [40]), and secondly, even if we agree with most of the suggested good practices, because we believe that these approaches do not take into account that they have to deal with researchers and students who spend only a part of their time developing code, and that, with respect to full time software engineers, will receive just a basic training in software development.

The workflow should stay as simple as possible, since the introduction of a new tool or a new dependency can be a big perturbation in the way researchers develop code, and may have a bad impact on the time required to newcomers for setting up their computers and for learning how to use all the tools. Therefore we chose to keep the number of dependencies as small as possible.

For the projects related to the iCub robot, our research team has been using CMake for almost ten years [41]. CMake allows one to configure, build, and deploy complex software across many different platforms [32], and supports several build system and IDEs. Moreover, the `ExternalProject` CMake module is powerful and is able to download, update, patch, configure, build, install and test source code taken from external repositories or archives, even if the build system is not written in CMake. Therefore basing our work on CMake was a natural choice. After examining the existing tools and taking into account pros and cons of adopting a new tool, or adding it to the workflow, we decided that there is no need for additional tools.

During the development phase, we also decided to avoid as much as possible local changes or workarounds, and instead to work upstream with CMake developers as much as possible proposing patches that fix the bugs that we found and that add the features that we required. Even though contributing upstream is a time demanding task, it reduces the effort to maintain the code and minimizes the duplication of efforts. At the same time we wanted to be able to rely on our patches immediately, without having to wait for the next CMake release.

To better support our users we decided to create a repository that works as incubator and testing ground for CMake modules and patches that can be later ported upstream. This repository allows us to write patches, and make them available to our users as soon as they are ready, while we work to get them accepted in upstream CMake. This repository is described in depth in the next section.

Moreover, it imports some of the modules that are new or updated in recent CMake revisions, (thus decreasing the difficulty of writing a build system that works with older CMake versions and in consequence increasing support for older Linux distributions), and a few 3rd party CMake modules.

VII. Results

Using CMake as our only dependency and scripting language, we created YCM[2] [42] (formerly "YARP CMake Modules"), a collection of CMake modules that simplify the creation of a build system and allow us to:

- keep using the tools we are used to, barely changing the workflow;
- write modular code in order to promote reuse;
- write build systems that minimize code duplication.

Most of the modules in YCM are dedicated to solving the issue of writing a build system for a library or for a module as fast as possible (Generic, Packaging Helper, and Find Package Modules in Table I). These modules add install targets for

[2]https://robotology.github.io/ycm/

Table I
LIST OF CMAKE MODULES CURRENTLY INCLUDED IN YCM.

Generic Modules	ExtractVersion GetAllCMakeProperties GitInfo IncludeUrl ReplaceImportedTargets StandardFindModule
Packaging Helper Modules	InstallBasicPackageFiles AddUninstallTarget AddInstallRPATHSupport
Find Package Modules	*40 Modules*
Superbuild Helper Modules	FindOrBuildPackage YCMEPHelper
Bootstrap Modules	YCMBootstrap
Build Package Modules	*11 Modules*
Style Modules	*8 Modules*

binaries and header files and produce CMake code for sharing the libraries that include version information (i.e. CMake's config files required to locate the library). They spare the user the task of writing, debugging and maintaining equivalent functions from native CMake functionalities (which requires considerable knowledge of CMake, is tedious and error-prone).

The remaining modules (Superbuild Helper, Bootstrap, Build Package, and Style Modules in Table I) allow us to create build system based superbuilds quickly, integrating external libraries and code written by other research teams and stored on different hosts (eventually also private), using different SCM and build systems.

The `FindOrBuildPackage` module is one of the main components of any YCM superbuild and it supplies the `find_or_build_package` command, that has a syntax similar to the `find_package` command from CMake. Both commands perform a check trying to locate the package on the system, but the former, instead of reporting the failure when a package is not found, includes a `Build<Package>.cmake` file containing the "instructions" (or recipe) to build the package. This file can be written in CMake language using `ExternalProject`, or even specifying manually the commands to execute for more complex tasks. To simplify writing these recipe files we supply and recommend to use the `YCMEPHelper` module, a wrapper around `ExternalProject` that adds several useful features and build targets.

Since the superbuild tries to locate the sub-projects early on and builds only the ones that cannot be found, the build time is reduced if the user installs one or more of the sub-projects from binary packages. Moreover, if multiple cores are available on the build machine, further speedup can be achieved by using parallel builds, that are fully supported as long as the dependencies between the modules of the superbuild are declared correctly. Finally the user does not need to build the whole superbuild, but can choose to build only the modules he is interested in. This contributes to decrease the time required to compile the whole superbuild from sources (thus mitigating a potential problem of our approach).

In order to avoid hard dependencies on YCM itself, superbuilds can bootstrap YCM and manage it exactly like any other sub-project. This means that a user can download just the superbuild repository, and build it like any other CMake project. If YCM is not installed on the system, it will be downloaded and built in the configure phase, so that its modules will be available later in the CMake project.

The `ExternalProject` CMake module allows us to save "snapshots" of all the repositories in a certain state, in order to freeze the state of the development at different points during the evolution of the project (for example whenever a new demonstration is available) by selecting the tag or the commit for each sub-project.

Normally users will not develop code for all the repositories, but only for a limited set of them. Therefore for each sub-project we provide a *user mode* and a *developer mode*. The latter adds, where applicable, a few extra targets to check the status of the repository, to build dependencies and dependees of the project, and to edit CMake cache. Moreover in developer mode, updates on the repository are disabled, in order to ensure that the status of the repository is always preserved as it was left by the developer (Table II).

We provide also a maintainer mode that enables all the targets for all the sub-projects.

The integration with CDash allows us to run continuous integration on superbuilds and receive detailed reports with successes and failures for each sub-project, thus checking the compilation of the whole system and not only individual components independently. Documentation for the whole superbuild can be generated documenting the code in the sub-projects using doxygen[3]. Dependency graphs showing the relationship among the sub-projects and the components of the

[3]http://www.doxygen.nl/

Table II
LIST OF BUILD TARGETS IN A YCM SUPERBUILD.

Global Targets		all test update-all fetch-all status-all clean-all print-directories-all
Component Targets		<COMPONENT> <COMPONENT>-update
Project Targets	Common	<PROJECT> <PROJECT>-test
	Basic Mode Only	<PROJECT>-update
	Development Mode Only	<PROJECT>-configure <PROJECT>-fetch <PROJECT>-status <PROJECT>-clean <PROJECT>-edit-cache <PROJECT>-open <PROJECT>-print-directories <PROJECT>-dependees <PROJECT>-dependees-update <PROJECT>-dependers <PROJECT>-dependers-update
	Special components	documentation examples templates

superbuild are generated using dot (Figure 2.) and can be used to monitor the status and evolution of the entire project.

Compared to a catkin/ament based superbuild (widely adopted in robotics due to the large user base of ROS), using a YCM superbuild has several advantages in our use cases:

- The YCM superbuild can automatically choose to build a necessary dependency only if it is not found (or not found with the necessary version and compilation options) in the system.
- The YCM superbuild integrates natively with all build systems supported by CMake, while the ament one depends on two different tools (vcstool and ament) and only support these tools for interacting with the superbuild. This means, for example, that on Windows with the YCM superbuild you can use just Visual Studio to update and compile the superbuild, while you need to switch to the terminal for the ament-based superbuild.
- catkin and ament require some specific metadata (the `package.xml` file) in the repository, or to use custom (typically non-portable) workarounds in order to inject the metadata in the source repository. In YCM superbuilds, all the information necessary to build the system are contained in the `BuildPackage.cmake` file, that is stored with YCM or with the superbuilds and does not need to inject anything in the source repository.
- catkin and ament add a dependency on python. This is not a big problem, but it can complicate the deployment on Windows systems.

In conclusion YCM superbuilds allow us to work on any platform, using different IDEs, and setting up machines quickly. They have a gentle learning curve for new members of the team and can be used at different levels, not just by developers that will modify the source code of some subproject, but also by team leaders, that for example wants to build the whole software quickly in order to show a demo. Our approach favors distribution of the source code to foster collaboration and contributions between developers.

We are currently using YCM to maintain four superbuilds for the following projects funded by the EU:

- The WALK-MAN[4] (Whole-body Adaptive Locomotion and Manipulation) project, which has the goal to develop a robotic platform (of an anthropomorphic form) which can operate outside the laboratory space in unstructured environments and work spaces as a result of natural and man-made disasters.
- The CoDyCo[5] (Whole-body Compliant Dynamical Contacts in Cognitive Humanoids) project, which has the goal to advance the current control and cognitive understanding about robust, goal-directed whole-body motion interaction with multiple contacts.
- The CENTAURO[6] project, which aims at developing a human-robot symbiotic system where a human operator is telepresent with its whole body in a Centaur-like robot capable of robust locomotion and dexterous manipulation in disaster scenarios.
- The CogIMon[7] (Cognitive Interaction in Motion) project, which aims at a step-change in human-robot interaction toward the systemic integration of robust, dependable interaction capabilities for teams of humans and compliant robots, in particular the compliant humanoid COMAN [43].

Figure 2 and Table III show how YCM superbuild can manage projects composed by just a few repositories (i.e. CoDyCo) and projects building a large number of sub-projects (i.e. WALK-MAN).

Source code from both projects and their dependencies is hosted on different SCM (mainly git, but also subversion and mercurial), on both public and private repositories. Most build systems are written using CMake, but there are a few exceptions of projects handled by the superbuilds that use other build systems.

The use of superbuilds contributed in the process of sharing code between the projects, and favored the collaboration between different groups. For example:

- The `iDynTree` and `paramHelp` libraries were developed for the CoDyCo project, but are currently being used also for the WALK-MAN project, whose developers have also contributed a few patches.
- The `GazeboYARPPlugins` module was developed in collaboration between the developers of the CoDyCo and WALK-MAN projects, and is now adopted by several developers using `YARP`, but not involved in these projects.
- The `YARP` module, developed in another context and imported by all the superbuilds as "external", received valuable contributions by all projects.

Software design and implementation in the WALK-MAN project was heavily driven by the participation to the DRC[8] (DARPA Robotic Challenge), which pushed a fast development cycle while ensuring at the same time the quality of the whole system architecture.

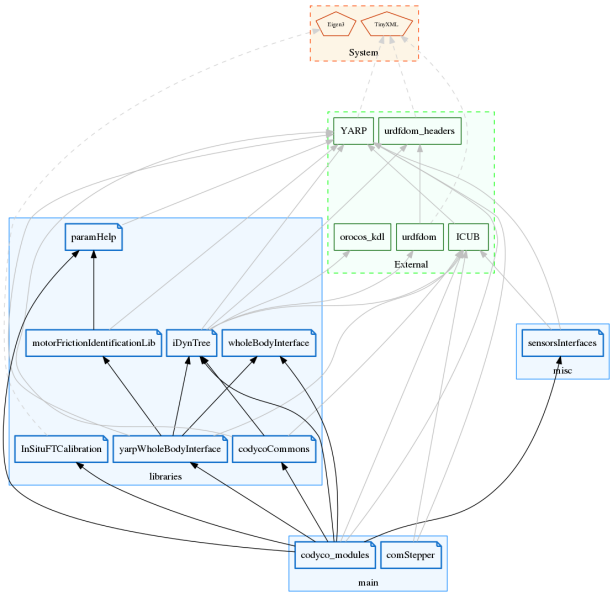
Table III
EVOLUTION OF THE WALK-MAN PROJECT SUPERBUILD.

Month	External Modules	WALK-MAN Modules
0	–	–
5	2	0
6	10	11
9	15	14
12	14	23
18	13	39
36	20	59

Table III shows that, even without a build manager, WALK-MAN developers were able to integrate a large number of projects in the superbuild. The WALK-MAN team was split in different sub-groups (e.g. locomotion group, valve turning task group, driving task group, etc.): each of them was able

[4]http://walk-man.eu/
[5]https://www.codyco.eu/
[6]https://www.centauro-project.eu/
[7]https://cogimon.eu/
[8]http://archive.darpa.mil/roboticschallenge/

Figure 2. Auto-generated graph showing the relationship among the sub-projects and the component of the CoDyCo project superbuild.



to work simultaneously thanks to the superbuild, which also ensured an efficient integration and code sharing.

A set of workflows has been adopted by the developers:

- installation, external development: the typical use case is a single developer, or small group, which can use the superbuild as a means of installing a set of libraries that can be of use to the development of his code. This allows one to keep the project "private" for a period of time, before finally adding it to the list of projects included in the superbuild for sharing.
- framework development: the typical use case is that of a developer that has to work on a large set of libraries at the same time, which typically are interdependent, and thus need to modify, test, commit changes and push in all of them.

In all the EU projects in which it was adopted, YCM allowed us to effectively manage the shared repositories with an increase in productivity and code sharing across projects. Table IV provide statistics to show the size of all four superbuilds (in terms of number of modules) and the number of modules shared with other superbuilds. These numbers demonstrates that the YCM allows us to handle repositories with large number of modules, with considerable re-use among different superbuilds.

Table IV
EXAMPLES OF YCM SUPERBUILDS ANALYZED IN THIS PAPER. WE REPORT THE NUMBER OF MODULES AND THE NUMBER OF MODULES SHARED WITH OTHER SUPERBUILDS.

Project	WALK-MAN	CoDyCo	CENTAURO	CogIMon
# modules	79	17	37	48
# modules shared with at least 1 other projects	28	5	21	15
# modules shared with at least 2 other projects	18	4	14	9

VIII. CONCLUSIONS AND FUTURE WORK

In this paper we discussed how the "*publish or perish*" model, the lack of practical knowledge of students and researchers, and the absence of specialized build and release managers heavily affect the re-usability of the code written for academic research projects. We emphasized that the build system has a key role for achieving a modular software that can be reused in other projects and that minimizes code duplication. We described the concept of "superbuild" and explained how this instrument can overcome several of the issues related to the code development process in research projects. Finally, we examined the existing tools that are used for handling dependencies in software development, and we studied some of the superbuilds used for building open source projects.

With all of this in mind, we developed YCM, a set of CMake modules that helps in the creation and maintenance of build systems for new projects, and "build system based superbuilds". One of the distinguishing features of our approach is that it favors distribution of source code using SCM tools. This, as opposed to distribution of binaries, has several advantages: it ensures that developers and users are able to compile the code, it favors collaborations and code contributions, and finally, it removes the cost of maintaining and distributing binary distributions for different systems.

Finally we showed how the introduction of the superbuild in our workflow contributes at the creation of reusable code, and encourages the collaboration between different teams.

We aim to use YCM for the entire iCub software ecosystem. The iCub Software Repository has grown in the last few years, up to a size that makes it hard to maintain. The build system is complex and delicate and most of the changes have unexpected side effects. Several libraries and modules that should have been generic and usable by other robots, have dependencies on parts that are iCub specific, and therefore a considerable maintenance effort is required to move them in separate repositories. We believe that by using a superbuild approach, the strong coupling will dissolve, allowing us to achieve a better collaboration and widespread use of the iCub software components. The "robotology-superbuild"[9] is currently under development, and we hope to adopt it in our future workflow.

Finally we will continue contributing upstream patches and modules from YCM to CMake, since we believe that in the future the useful features offered by YCM should be available to all CMake users.

ACKNOWLEDGEMENTS

This project has received funding from the European Union's Seventh Framework Programme for research, technological development and demonstration under grant agreement no. 611832 (WALK-MAN project).

[9]https://github.com/robotology/robotology-superbuild

References

[1] L. Natale, A. Paikan, M. Randazzo, and D. E. Domenichelli, "The icub software architecture: Evolution and lessons learned," *Frontiers in Robotics and AI*, vol. 3, p. 24, 2016. [Online]. Available: http://journal.frontiersin.org/article/10.3389/frobt.2016.00024

[2] D. F. Kelly, "A software chasm: Software engineering and scientific computing," *IEEE Software*, vol. 24, no. 6, pp. 118–119,120, 2007.

[3] D. L. D. Liu, S. X. S. Xu, and M. Brockmeyer, "Investigation on Academic Research Software Development," *2008 International Conference on Computer Science and Software Engineering*, vol. 2, pp. 626–630, 2008.

[4] S. Mcintosh, Y. Kamei, B. Adams, and A. E. Hassan, "The Impact of Code Review Coverage and Code Review Participation on Software Quality: A Case Study of the Qt, VTK, and ITK Projects," in *Proceedings of the 11th Working Conference on Mining Software Repositorie (MSR 2014)*, 2014, pp. 192–201.

[5] R. Cooper, "Why You Won't Fix It Later," 2007. [Online]. Available: http://on-agile.blogspot.it/2007/04/why-you-wont-fix-it-later.html

[6] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2008.

[7] M. de Jonge, "Build-level components," *IEEE Transactions on Software Engineering*, vol. 31, no. 7, pp. 588–600, 2005.

[8] A. Pawlik, J. Segal, and M. Petre, "Documentation practices in scientific software development," in *2012 5th International Workshop on Cooperative and Human Aspects of Software Engineering, CHASE 2012 - Proceedings*, 2012, pp. 113–119.

[9] G. Robles, J. M. Gonzalez-Barahona, and J. J. Merelo, "Beyond source code: The importance of other artifacts in software development (a case study)," *Journal of Systems and Software*, vol. 79, no. 9, pp. 1233–1248, 2006.

[10] B. Adams, H. Tromp, K. De Schutter, and W. De Meuter, "Design recovery and maintenance of build systems," *IEEE International Conference on Software Maintenance, ICSM*, pp. 114–123, 2007.

[11] S. McIntosh, "Build system maintenance," *2011 33rd International Conference on Software Engineering (ICSE)*, no. Section 4, pp. 1167–1169, 2011.

[12] S. McIntosh, B. Adams, T. H. Nguyen, Y. Kamei, and A. E. Hassan, "An empirical study of build maintenance effort," *2011 33rd International Conference on Software Engineering (ICSE)*, pp. 141–150, 2011.

[13] KDE Developers, "kdesrc-build – KDE Build Tool." [Online]. Available: https://kdesrc-build.kde.org/

[14] Gentoo Developers, "Portage." [Online]. Available: https://wiki.gentoo.org/wiki/Portage

[15] Homebrew Developers, "Homebrew." [Online]. Available: http://brew.sh/

[16] Microsoft, "vcpkg." [Online]. Available: https://github.com/Microsoft/vcpkg

[32] B. Hoffman and K. Martin, "CMake," in *The Architecture of Open Source Applications - Elegance, Evolution, and a Few Fearless Hacks*, A. Brown and G. Wilson, Eds., 2011, ch. 5. [Online]. Available: http://www.aosabook.org/en/cmake.html

[17] .NET Foundation, "NuGet." [Online]. Available: https://www.nuget.org/

[18] RealDimensions Software, LLC., "Chocolatey." [Online]. Available: https://chocolatey.org/

[19] pip Developers, "pip." [Online]. Available: https://pip.pypa.io/

[20] RubyGems Developers, "RubyGems." [Online]. Available: https://rubygems.org/

[21] NPM Developers, "NPM." [Online]. Available: https://www.npmjs.com/

[22] PEAR Developers, "PEAR." [Online]. Available: http://pear.php.net/

[23] CPAN Developers, "CPAN." [Online]. Available: http://www.cpan.org/

[24] Gradle Developers, "Gradle." [Online]. Available: https://gradle.org

[25] Biicode Developers, "Biicode." [Online]. Available: http://biicode.github.io/biicode/

[26] Conan Developers, "Conan." [Online]. Available: https://www.conan.io/

[27] SoftBank Robotics, "qiBuild." [Online]. Available: https://github.com/aldebaran/qibuild

[28] T. Straszheim, B. Gerkey, and S. Cousins, "The ROS build system," *IEEE Robotics and Automation Magazine*, vol. 18, no. 2, 2011.

[29] Open Source Robotics Foundation, "catkin." [Online]. Available: https://github.com/ros/catkin

[30] ——, "ament." [Online]. Available: http://design.ros2.org/articles/ament.html

[31] A. Mallet, "robotpkg." [Online]. Available: http://robotpkg.openrobots.org/

[33] J. Hughes, "CPM." [Online]. Available: http://cpm.rocks

[34] T. Becker, "cmakepp." [Online]. Available: https://github.com/toeb/cmakepp

[35] R. Baratov, "Hunter." [Online]. Available: https://www.hunter.sh

[36] The Qt Company, "Qt." [Online]. Available: https://www.qt.io/

[37] Kitware, "OpenChemistry." [Online]. Available: http://www.openchemistry.org/

[38] ——, "Paraview." [Online]. Available: http://www.paraview.org/

[39] R. Tedrake and the Drake Development Team, "Drake: A planning, control, and analysis toolbox for nonlinear dynamical systems," 2016. [Online]. Available: http://drake.mit.edu

[40] V. Hoffmann, H. Lichter, and A. Nyß en, "Processes and Practices for Quality Scientific Software Projects," in *Proceedings. of the Third International Workshop on Academic Software Development Tools and Techniques*, 2010, pp. 95–108. [Online]. Available: http://www.info.fundp.ac.be/wasdett2010/wp-content/uploads/2010/08/HLN-WASDeTT3.pdf

[41] P. Fitzpatrick, G. Metta, and L. Natale, "The CMaking of a Humanoid," *Kitware Source*, no. 13, pp. 7–9, 2010. [Online]. Available: http://www.kitware.com/media/html/TheCMakingOfAHumanoid.html

[42] D. E. Domenichelli, L. Natale *et al.*, "YCM 0.6.0," Dec. 2017. [Online]. Available: http://dx.doi.org/10.5281/zenodo.1123272

[43] N. G. Tsagarakis, S. Morfey, G. M. Cerda, L. Zhibin, and D. G. Caldwell, "COMpliant huMANoid COMAN: Optimal joint stiffness tuning for modal frequency control," in *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, may 2013, pp. 673–678.