THE CMAKING OF A HUMANOID

In the RobotCub project [1], we've been building a humanoid robot called the iCub. There are about a dozen iCubs in existence today (see Figure 1). The iCub design is free and open source. The designs for the mechanical and electrical components of the robot, its bill of materials, and all of its software are released publicly under free and open source licenses [2,3]. The iCub was created as part of a European project which began in September 2004 and ended in January 2010. There are several follow-on projects using and extending the robot in many ways, and we hope it will be a catalyst for research in robotics, embodied cognition, experimental psychology, and perhaps fields we've never even heard of. There were a lot of challenges in building the robot and making it usable. On the software side, CMake has been very important to us. We'd like to say a big "thank-you!" to the CMake developers, and explain a little about how CMake helped us along the way.

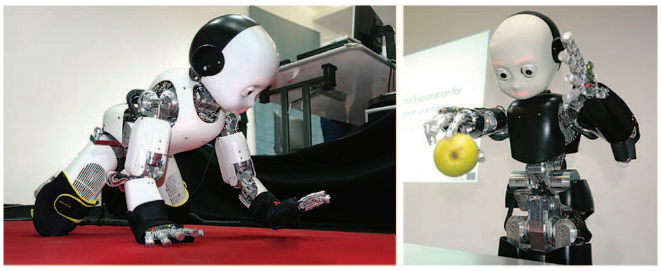


Figure 1: The iCub is a full humanoid with legs, dexterous hands, and a simple expressive face. The iCub is the size of a young child.

Making a humanoid robot is a huge task, requiring both novel research and patient integration work. It is practically impossible for any one person to be an expert in everything involved. Typical members of our development team are graduate students and postdocs who are experts in their fields, but may not necessarily be accustomed to working with others on software development. CMake turned out to be a wonderful tool for us, for the following reasons.

CMAKE HELPS DEVELOPERS WORK TOGETHER

People with all sorts of backgrounds end up working in humanoid robotics, so we see a lot of diversity in the development environments people are comfortable with. The biggest divide, when we began, was between users of Microsoft Visual Studio on Windows, and users of gcc+make+emacs/vi/... on Linux. From the start, RobotCub committed to formally supporting both these groups. But it was a lot of tedious work to keep all the build files synchronized, even with some custom scripting to help. And it was difficult to force individual developers to spend time dealing with tedious details of environments they didn't use or care about. This was a recipe for long unproductive discussions about why everyone should be using the one true development environment (whichever that was).

The first CMakeLists.txt file entered one of our repositories in April 2006 with the commit message "CMake, for easier Windows distribution" (this was clearly the perspective of a Linux developer tired of dealing with Windows). After some experimentation, doubts, and dabbling with alternatives, it became clear that CMake was a huge win. It gave a first-class experience to MSVC users when compiling projects developed by Linux users, and vice versa. Most of the time, developers could now forget about any development environment other than their own, and our community became a lot less "balkanized".

CMAKE HELPS INDIVIDUAL DEVELOPERS

Although we did not anticipate this, we discovered that CMake saved individual developers a huge amount of time, even before their work was used by others. One of our developers said "I can't imagine how much time I spent in my life cloning Visual Studio project files". CMake proved to be a better way of managing projects than was possible within MSVC itself. Similarly, beginner Linux users were very happy to stop looking at Makefiles that they didn't really understand and had a hard time modifying. CMake democratized the build process, through simplification.

Another unanticipated benefit was noticed as time went by. For MSVC users, as new compiler versions were released (VC8, VC9, VC10), they were saved from dealing with a lot of problems because their projects were already described in a compiler-neutral way, and CMake was prompt in its support for those compilers. Linux users found they could easily experiment with alternative development environments such as KDevelop or Eclipse. It was great for the RobotCub project not to have to dictate or even care about the development environment our developers worked with.

Since 2007, there's been a big movement from Windows toward Linux amongst students, principally driven by the phenomenal growth of Ubuntu. At our 2006 summer school, Windows was dominant; by 2009, the numbers were completely reversed. However, people whose background lies outside of computer science seem more likely to be still using Windows. Today, some robotic software projects neglect support for Windows (typical quote: "let's be serious: it's a robotic tool!"). Since humanoid robotics is a very interdisciplinary field, we believe it would be a mistake for us to adopt that attitude. In any case, good Windows support, via Cmake, has actually helped individuals working on RobotCub make the leap from Windows to Linux, since they know they'll be able to compile their code from day one. In general, CMake reduces lock-in to any particular development environment, which feels very liberating.

CMAKE HELPS SUMMER SCHOOLS SUCCEED

Since 2006, the RobotCub project has had an annual summer school ("VVV 06-09", and the school will continue in 2010 under the auspices of the ITALK project). The schools are essentially hackathons where we take whatever hardware we have to a nice locale in Italy and invite anybody with the necessary skills to come play. The schools are BYOL (Bring Your Own Laptop), and have proven to be an excellent "forcing function" for making the iCub usable and integrated. From the start, CMake has been a big help in getting such a mob of programmers up and running and collaborating. Here's how we introduced CMake at the first summer school, VVV06 (see Figure 2):

> *"We'd like you all to use the development environment you are used to, and not force you to switch to something else -- no Linux/g++/emacs vs Windows/DevStudio vs Mac/... fights please! To achieve this without complete chaos, we ask you to install "CMake". CMake lets us describe our programs and libraries in a cross-platform way. CMake takes care of creating the makefiles or workspaces needed by whatever development environment you like to work in."*

We suspect that the open attitude that CMake facilitates has played no small role in the success of the school.

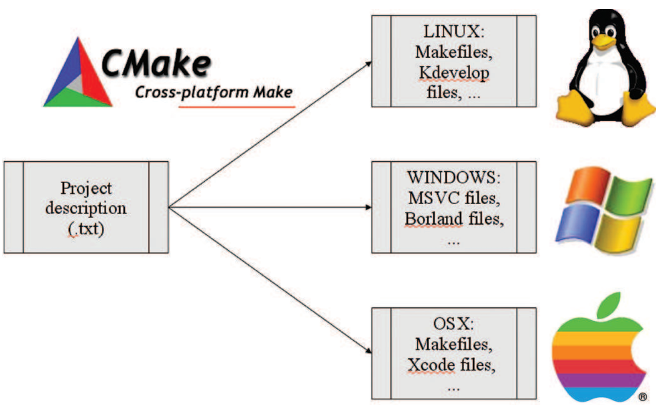

CMake
Cross-platform Make
Project description (.txt)
LINUX: Makefiles, Kdevelop files, ...
WINDOWS: MSVC files, Borland files, ...
OSX: Makefiles, Xcode files, ...

Figure 2: how we presented CMake to students in a presentation.

CMAKE HELPED US SCALE UP OUR REPOSITORY

RobotCub's community of developers is open and dispersed. The member institutes of the formal RobotCub consortium are scattered throughout Europe and we have collaborators worldwide. Software development mostly happens within an academic environment, where project and thesis schedules may conflict and the backgrounds or skill-level of developers vary widely. The most interesting software is often novel, experimental, created by researchers whose main skills (naturally and rightfully) lie in their problem domain rather than in software engineering. Due to paper submission deadlines, this software is often written in a hurry with little thought to reuse by others . For all these reasons, building up a common software repository for the iCub robot has been an interesting experience.

With a large community of loosely-coordinated developers it was difficult to reduce the list of dependencies required to compile the whole repository. In addition, some of the low-level code was not fully redistributable and thus, could not be included in the repository (this was the case for some device drivers required to interface with the hardware). Quite soon, being able to compile every single module in the repository required a large list of dependencies. This is something that often scares users away, especially in Windows, where package management is not as evolved and helpful as in other operating systems. For understandable reasons, new users are not very eager to spend a long time setting up software. CMake turned out to be very helpful in this. Our repository has a modular structure, where a group of flags govern the activation of groups of modules (e.g., graphical user interfaces, low-level modules providing hardware support, the simulator and so forth). Users that do not have all the required libraries can still compile a subset of the code, and start playing with it. Our policy is that the build process should not raise fatal errors when dependencies are not met, but rather should make available only the group of modules that can be compiled with the available dependencies. These flags controlling the build are made clear in the CMake GUI so that the user is aware that some parts of the build are not available. Getting and building the repository is an incremental and friendly process that avoids scaring away (new) users with incomprehensible error messages about unmet (and sometimes unwanted) dependencies.

For contributing to the repository, we have kept a low barrier of entry. RobotCub folks are told, in the top-level README, to "feel free to place new modules in [the repository] that aren't yet well tested, are currently broken, came to you in a deranged dream, etc." We found that it was very important to keep the initial barrier to entry very low, or code tended to stay hidden within institutions until it was fully done and perfect (i.e. never). With a low barrier to entry, there was mutual visibility between developers in the different institutions, and collaboration was easier to get going at the programmer level. Once developers are sure their module is no longer a "deranged dream" and, more specifically, are confident that it compiles on machines other than their own, they are encouraged to add it to the global build for the repository. This is quite a simple process even for inexperienced users: in most cases it is enough to customize a simple template CMakeLists.txt file and include it in the main build. Once in the global build, the module is tested automatically on our "officially supported" Linux/gcc and Windows/MSVC version combinations and we do some quality control at that point.

The plot below illustrates how the RobotCub software repository has grown during the project from its establishment in 2006, to the end of the project. At the end of the project our community consists of 32 developers (all contributing code in the past year), with almost 1 million lines of code and about 120 modules (programs and libraries) compiled in a single project (see Figure 3):

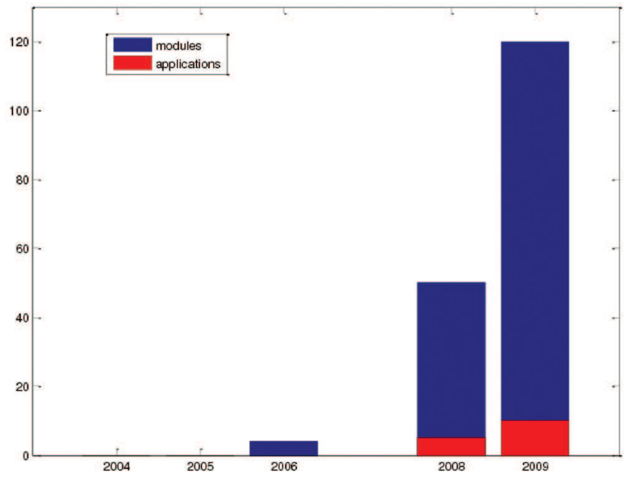


Figure 3: growth of the RobotCub repository. The nomenclature we use is a little non-standard. A "module" is any program or library, and an "application" is a logical collection of intercommunicating programs that performs a specific robot function (such as gazing, grasping, or crawling). The graph counts the modules that were compiled in the main project. In 2006, the repository was established with a few modules. By 2008, there were 50 modules and 5 applications. In 2009, there were 120 modules and 15 applications.

CONCLUSIONS

CMake helped keep our developers from being divided by operating system, and no-one was given the impression that they were a second-class citizen. Everyone spent more time working and less time fiddling around. Our repository grew to the scale we needed to operate a multi-functioned humanoid robot, without trouble. New users were able to build the parts of the project they were curious about first, and work their way up to everything else as they needed to. Being relatively early adopters of CMake, we made a bit of a muddle of our build structure (we had never used CMake before RobotCub), but things worked out remarkably well in spite of that. More recently, we've noticed many peer projects starting to use CMake, for example Player/Stage and ROS. It is good to see this "bottom-up" convergence, since it makes collaboration easier.

It is striking how many free and open source projects played a vital role in developing the iCub: CMake, Doxygen (which allowed us to easily pool and browse documentation, and provided "soft" pressure on developers to actually write that documentation by making its absence very obvious), SWIG (was very useful in expanding our pool of users), the OpenCV library (which served as a de facto standard for basic computer vision algorithms, rather than everyone writing their own). The list goes on and on. We hope that the iCub robot or a descendent will find a similarly useful role in robotics, and are grateful to our funder, the European Commission, for allowing us to give back to the free software community that has served us so well.

REFERENCES

[1] RobotCub project website, http://www.robotcub.org

[2] RobotCub hardware designs, http://eris.liralab.it/wiki/RobotCub

[3] Aggregrated RobotCub software, http://eris.liralab.it/brain/

Paul Fitzpatrick *received his MEng in computer engineering from the University of Limerick, Ireland, and his PhD in computer science from MIT for work addressing developmental approaches to machine perception, implemented on the humanoid robots Cog and Kismet.*

Giorgio Metta *is senior scientist at the IIT and assistant professor at the University of Genoa where he teaches courses on anthropomorphic robotics and intelligent systems for the bioengineering curricula. He holds a MS with honors and PhD in electronic engineering from the University of Genoa. His research interests include biologically motivated and humanoid robotics, in particular lifelong artificial systems which demonstrate the abilities of natural systems.*

Lorenzo Natale *is Team Leader in the Robotics Brain and Cognitive Sciences Department at the Italian Institute of Technology. Since 2000 he has been involved in various humanoid robotic projects, working on aspects related to perception, motor control and learning. He is also interested in software development and is one of the main developers of YARP, an open-source middleware for robotics.*

PARAVIEW IN AERODYNAMICS

CFS Engineering is a consultancy company founded in August 1999, based in the business park at the Swiss Federal Institute of Technology in Lausanne, Switzerland. CFS Engineering offers services in the Numerical Simulation of Fluid Mechanics and Structural Mechanics Engineering Problems, and in particular, is active in the development of the Navier Stokes Multi Block (NSMB) solver used in a variety of aerospace problems ranging from flows over civil and military aircraft to hypersonic flows including air chemistry over future re-entry vehicles.

The NSMB solver solves the Navier Stokes equations using the finite volume approach. Space discretization schemes include central and upwind methods. The equations are integrated in time using the LU-SGS scheme.

NSMB offers a variety of modern turbulence models. Different levels of chemistry modeling are available for hypersonic flows, ranging from chemical equilibrium to thermal chemical non-equilibrium. NSMB has been parallelized using the SPMD paradigm using MPI as message passing.

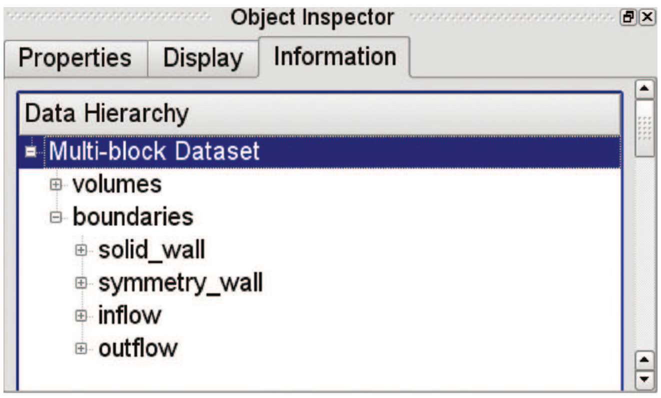
ParaView has been an important asset to this small company's portfolio of tools, and is used in mesh and fluid flow visualization tasks.

The NSMB solver archives its data in a proprietary file format. Large and complex flight geometries can produce gigabytes of data for steady and transient simulations.

Our interface has grown along-side the release cycles of ParaView, and we are today, extremely grateful to Kitware and the open-source community at large for all the features available. Our ParaView plugin for the NSMB solver supports a wide range of data analysis tasks, which we succinctly present here.

HIERARCHICAL MESHES, QUANTITATIVE AND QUALITATIVE DISPLAYS

Our reader plugin relies on a low-level access API for the coordinates and data arrays, and lets us concentrate on building the VTK objects used by a hierarchy of multiple containers of 3D and 2D grids. Enabling parallel reading was particularly important, given today's multi-core servers and desktops. In fact, our rendering needs are often easily handled by a single hardware renderer (the ParaView client), whereas a pvserver can share the work load amongst several machines.



Data Hierarchy
- Multi-block Dataset
  - volumes
  - boundaries
    - solid_wall
    - symmetry_wall
    - inflow
    - outflow

A typical flight configuration may include several thousands of body-fitting 3D curvilinear grids, along with surface patches for the solid geometry, and the different boundary conditions walls. The data will include standard flow variables (density, pressure, velocity) and wall shear stress vectors. ParaView enables the graceful presentation and sub-setting of such hierarchies of data blocks along with data fields and multiple time-cycles.

Multi-parameter studies, where flow conditions may change and the simulation is re-executed can take advantage of the side-by-side viewing available in Comparative View mode, as shown below for a hypersonic flow simulation.