Virtualization Techniques for Cross Platform Automated Software Builds, Tests and Deployment

Thomas Müller and Alois Knoll
Robotics and Embedded Systems
Technische Universität München
Blotzmannstr. 3, 85748 Garching, Germany
{muelleth,knoll}@cs.tum.edu

***Abstract*—In this paper, an integrated approach for cross platform automated software builds and the implementation of a test framework is described. The system introduced here utilizes state-of-the-art virtualization tools to accomplish this task. In this way a single off the shelf PC can be used to build and test the software and in addition to that create deployable packages for multiple target platforms. Hence, the main contribution is an architectural concept for automated cross platform builds and testing based on virtualization of the processes that are platform dependent. The system was introduced for a large academic software library project in the field of robotics and machine vision. Here it is now productive and provides continuous feedback for developers, as well as includes the possibility to obtain the latest binary releases for end users.**

***Index Terms*—Cross-Platform Software Builds; Virtual Software Testing and Deployment; Build Report Automation;**

I. Introduction

It should be of no question for any software developer to write a little test for the classes or methods / functions he implements. But this is by far not enough to ensure reasonable quality and stability of a final software product. Thus, one can argue that it would be much better to implement a more general, more sophisticated framework to automate the testing process and deployment.

The test toolkit introduced in this paper incorporates the advantageous capability to integrate into an automated deployment framework, i.e., with nightly builds, automated virtual deployment and test execution. The system was motivated by the need for a sound process to automate these tasks for a large software library project in the context of robotics and machine vision. Here, many developers code at a time while others use the library in a productive robotic setup at the same time. Hence, the system has to ensure stability and robustness of the generated library to these end users on multiple target platforms.

However, the focus of this paper is not the implementation of certain tests, nor to discuss different methods for testing software, as there is a magnitude of work in this field, e.g. [1] [2] [3]. Instead, the contribution of this paper is to introduce an architectural concept for automated cross platform software development based on virtualization of the processes that are platform dependent.

The remainder of this paper is structured as follows: The next sections gives a brief description of the framework and what it was built for and how it fits in what has been achieved before. Section IV explains the setup of our host system and how it runs the utilized virtual machines. In Section V we show, how the proposed framework can be used to automate test execution and reporting. The final Sections VI and VII explain details about our experimental setup, draw conclusions and name possible extensions and further directions regarding the proposed framework.

II. Overview and System Requirements

The library project, which is the application scenario of the proposed approach, comprises of three major parts, (1) the library itself, (2) multi-layer tests and (3) tutorial applications also detailed in a reference guide. A brief overview of the requirements for the automated build, test and deploy framework are given in Table I.

	Library	**Tests**	**Tutorials**
Source repository	Subversion	Subversion	Subversion, HTTP
Binary repository	Subversion, HTTP	Subversion	-
Code Variability	High	Medium	Low
Build Environment	Yes	Yes	No
Dependencies	3rd Party	3rd Party, Library	3rd Party, Library
Platform	Windows, Linux, (Mac)	Windows, Linux	Platform specific
Package Contents	Library binaries, Headers, Reference manual	-	Sources, Binaries, User guide
Error Reporting	Build, Packaging, Installation	Build, Execution	Build
Access Permissions	Developers, End-Users	Developers	Developers End-Users

TABLE I
Framework overview

As one can see from the table, the three parts have different dependencies, build specifications, deployment scenarios and access permissions.

III. Related Work

There is certainly well-known literature on automation of software builds (e.g. [4] for a nice introduction). The major

challenge here is to build the software for different target platforms in a clean and scaleable manner. The usual approach would be to setup a computer running the target operating system and configuration, then setup a build environment on that platform, e.g. compiler and third party dependencies, and build the binaries or packages for end users.

On the other hand, virtualization is widely known as a tool for composing processes that need to be run on different operating systems on a single host.

Combining the two above approaches, we follow the idea of having just one host machine that automates not only software builds for a single target platform / configuration, but also handles automatic testing and reporting. For this purpose we setup a set of virtual machines on the host that actually perform the build, install and testing. Furthermore, the host is configured to collect reports and binaries provide them on a website for authorized users.

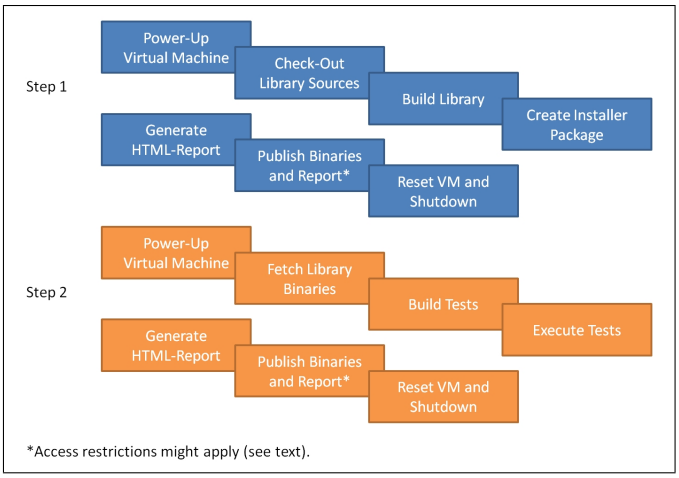


Fig. 1. Toolchain

IV. Virtual Machine Setup

According to the project partitioning, a host machine has to be set up to perform the required tasks automatically for cross platform target systems. In this stage of expansion the physical host PC runs instances of virtual machines themselves running different operating systems. For each target operating system there is a dedicated virtual OS for compilation and packaging of the library and for compilation and execution of the tests and compilation tutorials. Moreover, a clean, freshly installed virtual OS for unattended library installation and test execution is used.

This strategy helps avoiding corrupt test results and ensures that a certain OS is compatible. Furthermore, the architecture is open, so whenever a new OS shall be supported, one only needs to setup a virtual machine running the new target OS and run the toolchain (see below) on that system. For cross-platform builds and configuration of unattended installation the open-source build system *CMake* is used, which creates *NullSoft* installer packages for Windows and Debian packages and tarball-archives for Linux. Please refer to [5] and [6] for further details.

Figure 1 shows the toolchain for library build, packaging, deployment and report generation (step 1) and unattended library installation, test build, execution and report generation (step 2). The toolchain for part three of the project, the tutorials, is analogous to the process for testing.

As indicated in the figure and Table I, access restrictions might apply depending on the permission status assigned to a user. Developers may access each of the library source revisions as well as reports regarding the result of a build and the binaries (installer packages for each supported target platform). They are also allowed to view all reports, be it library build reports, test build and execution reports or tutorial build reports.

Users are only allowed to download the latest binary package of the library when the tests were run successfully. However, they are allowed to view reports on the library builds, as this might provide useful information for debugging or additional information about new features, e.g., from the change log.

The host PC system runs a webserver that lists files from directories, which allow write access for the virtual machines. The virtual machines first collect information about success of the build, deploy, install and test execution locally and then create a timestamped subfolder for each cycle of the toolchain and upload their data to that folder after completion. Figure 2 shows a snapshot of the resulting generated public directory structure on the filesystem of the host machine.

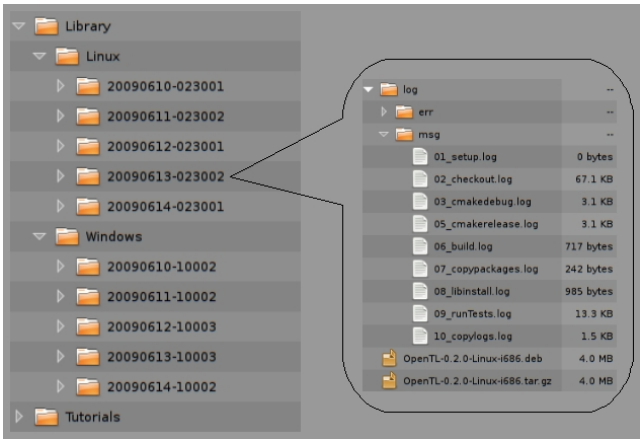


Fig. 2. Generated directory structure on the filesystem of the host after a build, deploy and test run on a Windows and a Linux virtual machine.

From here, authorized users may download and inspect the log files or binaries using a web-interface. The access permissions are applied considering *Apache*'s per folder directive mechanism[7].

V. Library Testing

To give the reader a better understanding of what the system actually has to achieve, Figure 3 shows the multi-layer architecture of the library project [8], [9] which has to be built, tested and deployed by the proposed framework.

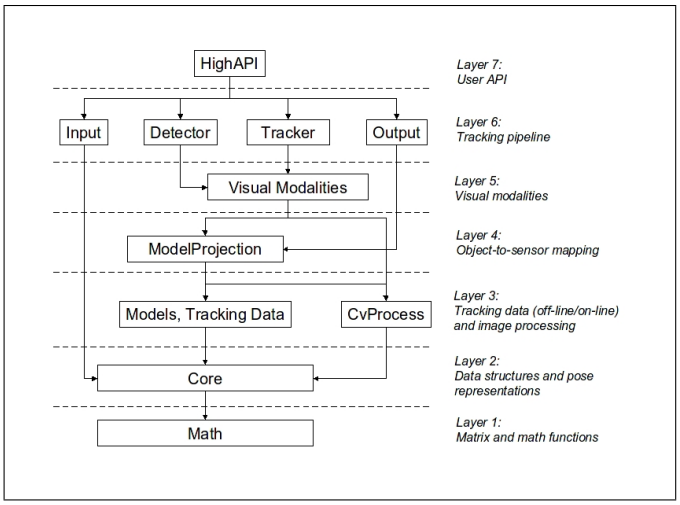


Fig. 3. Library Modules (September 16, 2008)

The library itself is a rather complex compilation of modules with both extrinsic and intrinsic dependencies. Extrinsic dependencies occur throughout the modules and require certain third-party software to be installed at build-time, e.g., the *boost* c++ libraries or *GSL*, the GNU Scientific Library (details at [10] and [11]). These dependencies are taken "as is" and are thus not particularly subject matter of this paper.

Intrinsic dependencies are structured hierarchically, as one can see from the architecture snapshot in Figure 3. This layered architecture allows for a bottom-up design of the test system. The test framework executes the functional tests written by developers for each of the modules, in a bottom-up manner. Thus, when tests on a lower layer fail, testing is aborted and the report is generated. For functionally tested lower level modules - each of which becomes a software library module (e.g. `.dll` on Microsoft Windows or `.so` on Linux) - integration tests take place as a next step. This is an intuitive process, as for example, after testing the `Math` module, the `Core` module's tests are run, which heavily rely on the underlying math algorithms. Hence, the `Core` tests automatically do black-box and integration testing with these.

If all tests succeed hierarchically, some (up to this moment trivial) black box tests are applied. These only execute some predefined `High-API` applications. As the subject of the library is mainly model-based stochastic visual tracking, processes happening in the library are intrinsically random - the only objective measure is the accuracy of the tracking result.

As an example of a high-level test executed by the framework, one can see from Figure 4 the 3D face tracking application implemented as part of the library. A screenshot of the input video is shown on the left. On the right side in the figure, the computed 3D data, being the output of the tracking application, is back-projected to the image plane. This visual feedback is only neccessary for manual evaluation, as for human it is most convenient to judge about the result by inspection - one can see, whether the output of the tracker makes sense.

Fig. 4. For manual evaluation the tracking result has to be back-projected to the original video sequence

On the other hand, when the framework automatically executes tests inside a virtual machine, there is no need for such visual feedback. The framework rather takes advantage of the intrinisc representation of a tracking result. Clearly, the result is set of parameters, in this example composed of the 3D pose and a motion and deformation estimate. Thus, for test automation, a corresponding set of values has to be stored in a ground truth data file for each test manually by the developer. The test is considered to be successful, only if the parameters computed by the tracking application on the input video match the ones being loaded from this ground truth data.

As mentioned above, the test results are made available to dedicated user groups on the project website. To automate this, the free tool *CDash* [12] is used. CDash generates the executables for the testcode from library developers inside a virtual machine. These tests are then run by the framework and the results are published to a dashboard that is provided by the host machine. These results can then be reviewed in great detail and with all available history information by authorized users. Figure 5 shows a screenshot of the generated dashboard startsite. In the figure, one can see successful submissions from a Linux and a Windows virtual machine.

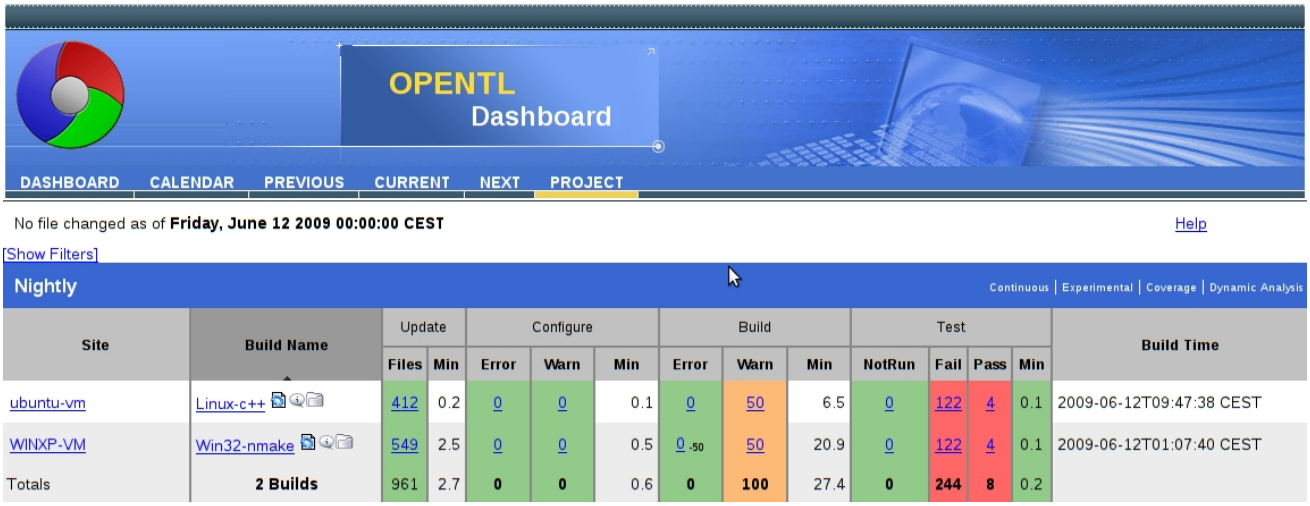
VI. Experiments and Results

The system proposed in this paper is run on a off-the-shelf host PC, precisely a AMD Athlon(tm) 64 X2 Dual Core Processor 3800+ with 2048 MB of RAM.

Each of the virtual machines are allowed to use one core of the host CPU, 512 MB of RAM and 10 GB of disc storage at the most. As described in Section IV, each of the steps in the toolchain requires an instance of a virtual machine, so in order to avoid exceeding the physical limits of the host, a scheduling algorithm is applied. The host OS is Ubuntu 8.10 and thus scheduling can be easily done by installing `cron` jobs for the desired tasks.

Furthermore, as virtualization software the free VMWare Server 2.0.0 is used, so running the virtual machines in *non-persistent mode* leaves the actual image unchanged after performing the desired tasks.

The system is scheduled to distribute library builds, test builds and executions and tutorial builds over a whole day.



Site	Build Name	Update		Configure			Build			Test				Build Time
		Files	Min	Error	Warn	Min	Error	Warn	Min	NotRun	Fail	Pass	Min	
ubuntu-vm	Linux-c++	412	0.2	0	0	0.1	0	50	6.5	0	122	4	0.1	2009-06-12T09:47:38 CEST
WINXP-VM	Win32-nmake	549	2.5	0	0	0.5	0 -50	50	20.9	0	122	4	0.1	2009-06-12T01:07:40 CEST
Totals	**2 Builds**	961	2.7	**0**	**0**	0.6	**0**	**100**	27.4	**0**	**244**	**8**	0.2	

Fig. 5. The *CDash* start page used to make testing results available for the software library *OpenTL*

Therefore, the chance that virtual machines interfere with each other is very low, although up to now three operating systems and several configurations are supported.

Figure 6 shows screenshots of two virtual machines after a library build. From the directory structure one can see the fresh checkout of the trunk source code from subversion [13]. Inside this folder the binaries are being built. For this purpose, the `environment_template` folder contains the target folder structure for each build. At a scheduled execution of the build script, this folder is renamed to the timestamped version and all processes write to that folder. After completion, the source code is being removed and the log files are copied together with the resulting binaries (if any) to the host system. The host system's public share is either mounted directly into the filesystem (Linux) or mapped as a network drive (Windows) as `atknoll131`.

The contents of the host's public share is shown in Figure 7 form a Windows virtual machine perspective. Here, the `.htaccess` file contains the access permissions, the footer, header and style templates for the website and some additional software (e.g. third party software) users need to run library applications.

Figure 8 contains a detail of the generated website depicts the actual content that is accessible to authorized users.

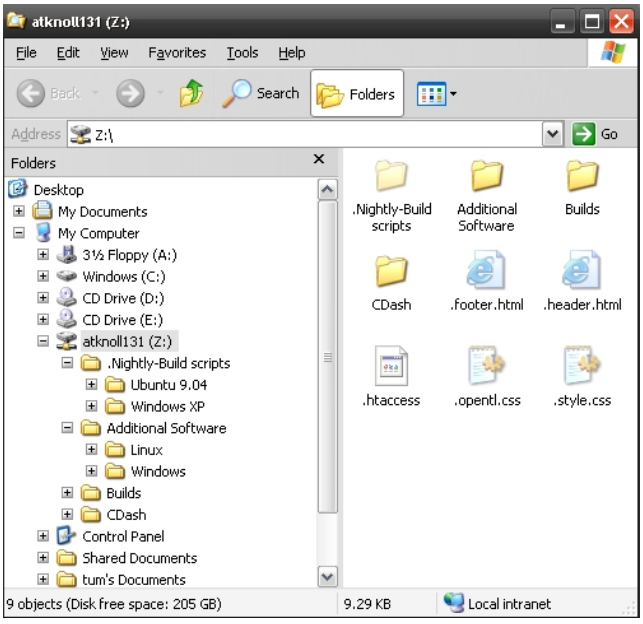


Fig. 7. Screenshot of the public share of the host from within a Windows virtual machine.

VII. Conclusion and Future Work

The work proposed in this paper covers implementation and setup of a software build, deploy and test framework. This framework is able to accomplish tasks for cross platform target operating systems and various software configurations. This can be achieved due to virtualization. Virtual operating systems are instantiated for each task in a non-persistent mode, so at start-up it can be guaranteed, that the virtual machine is in a "good" state. Thus the system produces repeatable results for equal sources on each of the target platforms. These results are manually being cross-checked on physical instances of the target OS from time to time, so it can be assumed, that using virtual OS is equivalent to using physical ones and the proposed system is transparent.

Furthermore, based on the setup of the virtual machines, it is quite easy to exactly specify the requirements of physical hardware for the end user and investigate the performance difference emerging from higher memory capacity, better CPU availability, etc. For this purpose one simply has to reconfigure the virtual machine, which facilitates the evaluation process enormously. Finally, it is also possible to replace the host system conveniently, as the virtual machine images can simply be copied to the new host and run directly. All the new host needs to do is provide a public folder for the results and take this folder as the document root for a webserver.

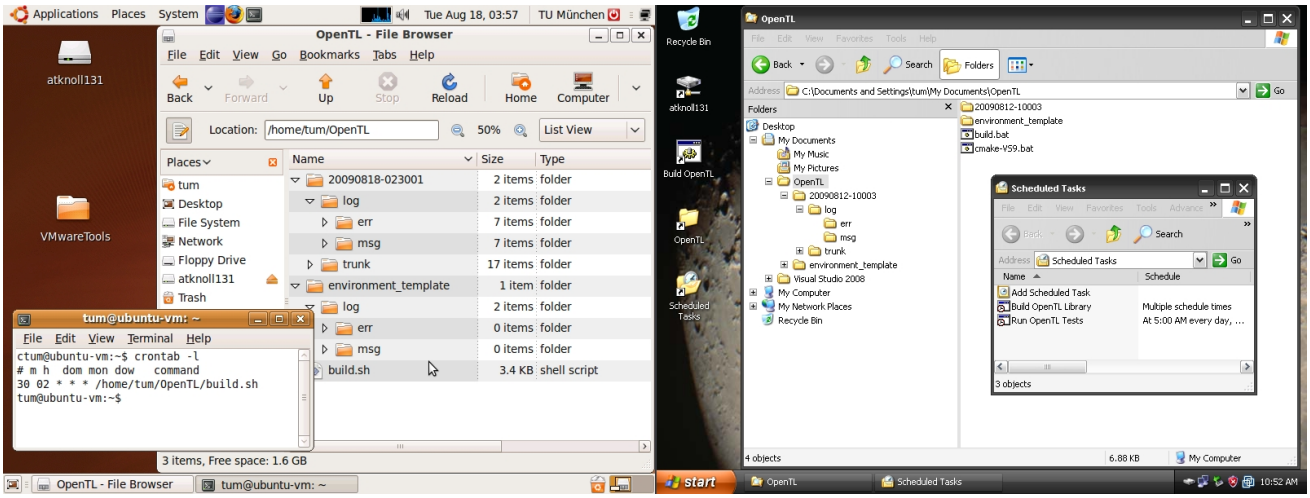


Fig. 6. Screenshots from within the virtual machines, showing the scheduled tasks and local build structure

Fig. 8. Snapshot of the website generated for public / restricted access

Nevertheless, there are some problematic aspects in the setup described here. First, which is a rather general aspect in software testing, it is rather difficult to automate the interactive parts of an application. This is of minor relevance for the proposed system in the current scenario, as its purpose is testing a mostly non-interactive software library. Second, incremental bottom-up software testing of modules sometimes is inconsistent, as there is not only one module on each layer (see Figure 3).

One of the next steps is the extension of the system to include GUI- and / or interactive testing tools in the specific virtual machines to automate a high-level test cycle. Another idea is to separate platform dependent from platform independent code modules and only apply virtualized testing to the platform dependent pieces. This could increase the performance, i.e. the duration of a test run, significantly, as there are very little platform dependent modules.

ACKNOWLEDGMENT

This work is supported by the German Research Foundation (DFG) within the Collaborative Research Center SFB 453 on "High-Fidelity Telepresence and Teleaction".

REFERENCES

[1] M. Fewster and D. Graham, *Software Test Automation*. ACM Press, 1999.
[2] R. Black, *Pragmatic Software Testing: Becoming an Effective and Efficient Test Professional*. Wiley Publishing Inc., 2007.
[3] D. J. Mosley and B. A. Posey, *Just Enough Software Test Automation*. Prentice Hall PTR, 2002.
[4] J. Minnihan, "Build and Release Management ," 21-Aug-2009, http://freshmeat.net/articles/build-and-release-management.
[5] CMake, "Cross Platform Make," 21-Aug-2009, http://www.cmake.org.
[6] NSIS Wiki, "nullsoft scriptable install system," 21-Aug-2009, http://nsis.sourceforge.net.
[7] Apache HTTP Server, "Directive Quick Reference," 21-Aug-2009, http://httpd.apache.org/docs/2.2/en/mod/quickreference.html.
[8] G. Panin, C. Lenz, S. Nair, E. Roth, M. in Wojtczyk, T. Friedlhuber, and A. Knoll, "A unifying software architecture for model-based visual tracking," in *IS&T/SPIE 20th Annual Symposium of Electronic Imaging*, San Jose, CA, Jan. 2008.
[9] G. Panin, E. Roth, T. Röder, S. Nair, C. Lenz, M. Wojtczyk, T. Friedlhuber, and A. Knoll, "ITrackU: An integrated framework for image-based tracking and understanding," in *Proceedings of the International Workshop on Cognition for Technical Systems*, Munich, Germany, Oct. 2008.
[10] boost C++ Libraries, "Getting Started," 21-Aug-2009, http://www.boost.org.
[11] GNU Project, "GSL - GNU Scientific Library," 21-Aug-2009, http://www.gnu.org/software/gsl.
[12] CDash, "An open source, web-based software testing server," 21-Aug-2009, http://nsis.sourceforge.net.
[13] Tigris.org, "Subversion," 21-Aug-2009, http://subversion.tigris.org.