MORSE AND STATE-MACHINE-BASED TESTING FOR
THE HUMANOID ROBOT HEAD FLOBI

International Workshop on MORSE and HRI 2014
FLOBI FACTS

Designed as a comic-like human face to avoid uncanny effects
Hair, eyebrows, lips and frontal face parts are exchangeable
Features stereo vision, stereo audio and a gyroscope
Innovative magnetic actuation mechanism
Custom design motor control boards
Eye saccades reach ~400 degree/s
Overall 18 DoF

More about Flobi: http://pub.uni-bielefeld.de/publication?ftext=flobi
Rapid prototyping & testing required a simulator
FLOBI SOFTWARE STACK

Virtual Robot  
MORSE [5] (of course)

Physical Robot

running in virtual or hardware mode
virtualizes controller boards in "-v mode"
simulates motor characteristics (ramps, vel, etc.)

setRegister(0x80, 30)
ros.pub(msg, JointState)
setJoint("NECK_PAN", 30)
Flobi API  
C++ java Python

ROS:: / RSB[2]
FLOBI SOFTWARE STACK

https://www.youtube.com/watch?v=PBs0c2LzMVM
But...
... how precise is our simulator?
/off: who doesn’t know CI?
Continuous integration

Continuous integration (CI) is the practice, in software engineering, of merging all developer working copies with a shared mainline several times a day. It was first named and proposed as part of extreme programming (XP). Its main aim is to prevent integration problems, referred to as "integration hell" in early descriptions of XP. CI can be seen as an intensification of practices of periodic integration advocated by earlier published methods of incremental and iterative software development, such as the Boooh method. CI isn't universally accepted as an improvement over frequent integration, so it is important to distinguish between the two as there is disagreement about the virtues of each.[citation needed]

CI was originally intended to be used in combination with automated unit tests written through the practices of test-driven development. Initially this was conceived of as running all unit tests and verifying they all passed before committing to the mainline. This helps avoid one developer's work in progress breaking another developer's copy. If necessary, partially complete features can be disabled before committing using feature toggles.

Later elaborations of the concept introduced build servers, which automatically run the unit tests periodically or even after every commit and report the results to the developers. The use of build servers (not necessarily running unit tests) had already been practised by some teams outside the XP community. Nowadays, many organisations have adopted CI without adopting all of XP.

In addition to automated unit tests, organisations using CI typically use a build server to implement continuous processes of applying quality control in general — small pieces of effort, applied frequently. In addition to running the unit and integration tests, such processes run additional static and dynamic tests, measure and profile performance, extract and format documentation from the source code and facilitate manual QA processes. This continuous application of quality control aims to improve the quality of software, and to reduce the time taken to deliver it, by replacing the traditional practice of applying quality control after completing all development. This is very similar to the original idea of integrating more frequently to make integration easier, only applied to QA processes.

In the same vein the practice of continuous delivery further extends CI by making sure the software checked in on the mainline is always in a state that can be deployed to users and makes the actual deployment process very rapid.

http://en.wikipedia.org/wiki/Continuous_integration
### CI PRIMER

<table>
<thead>
<tr>
<th>Name</th>
<th>Last Status</th>
<th>Last Failure</th>
<th>Last Duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenSource CI/EC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MorsePublic-1.0-master-build</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MorsePublic-1.0-master-testing-BASE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MorsePublic-1.0-master-testing-ROBOTS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MorsePublic-1.0-master-testing-MIDDLEWARES</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

CI PRIMER

Console Output

Started by upstream project "MorsePublic-1.0-master-build" build number 831
originally caused by:

Started by an SCM change
Building remotely on Ubuntu64-Precise-CLF-only (CLF-only) in workspace /vol/clf/releases/precise/x64/morse-testbed-fixed/label/Ubuntu64-Precise-CLF-only

Fetching changes from the remote Git repository
Fetching upstream changes from https://github.com/morse-simulator/morse.git
Checking out Revision f60cf5114116cf2417a70337b356f5fe5574dd08 (origin/master)
[Ubuntu64-Precise-CLF-only] $ /bin/sh -xe /tmp/hudson5366427741821729592.sh
+ echo ---- SLAVE MACHINE INFO ----
+ cat /etc/issue
Ubuntu 12.04.4 LTS \\n
+ uname -a
Linux sanarium 3.2.0-52-generic #78-Ubuntu SMP Fri Jul 26 16:12:14 UTC 2013 x86_64 x86_64 x86_64 GNU/Linux
+ echo ---- Building MORSE ----
+ echo DONK. Via http://jenkins-ci.org/hanoi/setup.sh
DONK. Via https://jenkins-ci.org/hanoi/setup.sh
+ echo ---- Fetching and Re-Building ----
---- Fetching and Re-Building ----
+ cd /vol/clf/releases/precise/x64/morse-testbed-fixed/src/morse
+ mkdir -p build
+ cd build
+ rm -zf BaseTest.xml CMakeCache.txt CMakeFiles CTestTestfile.cmake DartConfiguration.tcl Makefile MiddlewareTest.xml RobotsTest.xml Testing bin bindings cmake_install.cmake cmake_uninstall.cmake install_manifest.txt label src testing version.py
+ cmake -DCMAKE_INSTALL_PREFIX=/vol/clf/releases/precise/x64/morse-testbed-fixed/ -DPYMORE_SUPPORT=ON -DPYTHON_EXECUTABLE=/vol/clf/releases/precise/x64/morse-testbed-fixed/bin/python3.3 -DBUILD_ROB_SUPPORT=ON -DCMAKE_BUILD_TYPE=Release ..

-- The C compiler identification is GNU
-- The CXX compiler identification is GNU
-- Check for working C compiler: /usr/bin/gcc
-- Check for working C compiler: /usr/bin/gcc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Found PythonInterp: /vol/clf/releases/precise/x64/morse-testbed-fixed/bin/python3.3 (found suitable version "3.3", required is "3.2")
-- Found PythonLibs: optimized:/vol/clf/releases/precise/x64/morse-testbed-fixed/lib/libpython3.3m.so;debug:/vol/clf/releases/precise/x64/morse-testbed-fixed/lib/libpython3.3m.so (found
FLOBI SIMULATION VALIDATION
FLOBI SIMULATION VALIDATION
FLOBI SIMULATION VALIDATION
FLOBI SIMULATION VALIDATION

Worked well, so far…
Does **not scale** for multiple projects/test
Time consuming configuration via GUI
Not 100% **deterministic** (default Jenkins job scheduler)
Requires profound “Jenkins knowledge” to setup test scenarios
Technology for **developers**, which often excludes, i.e., social sciences
New approach: FSMT
1 WHY DO YOU NEED TO TEST HRI SCENARIOS?
Autonomous robots are highly relevant targets for interaction studies, but can exhibit behavioral variability that confounds experimental validity.

TESTING REAL SYSTEMS PREVENTS ERRORS IS VERY LABOUR-INTENSIVE OFTEN HAPPENS TOO LATE

2 WHAT IS TO BE TESTED?
Detailed testing of experiment designs, their software realizations and hardware is required.
DIFFICULT TO MAINTAIN INHERENTLY COMPLEX

3 WHO IS INVOLVED IN TESTING?
Experimenters as well as system developers must cooperatively design and integrate their experiments as easily and often as possible to improve the test and evaluation process.

STRICT ADHERENCE OF AN EXPERIMENT PROTOCOL REQUIRES PROFOUND TECHNICAL SKILLS IN DISTRIBUTED SYSTEMS SEQUENCING IS CRUCIAL
FINITE-STATE-MACHINE-BASED-TESTING

a) Establish experiment prototyping. This uses simulation environments including a virtual human component b) Extend the concept of an experiment protocol to the orchestration of software components, and c) Execute and assess the results of a prototype experiment in an automated, easy-to-use fashion.
FINITE-STATE-MACHINE-BASED-TESTING

a) Establish experiment prototyping. This experiment protocol to the orchestration of software tests and easy-to-use fashion.

[environment]
FSMPREFIX=/media/FSM-Lab/releases/precise/x64/
MORSE_ROOT=/media/FSM-Lab/releases/precise/x64/

[component-1]
name = eval_human_pose
command = python eval_xyz_pose.py
path = /opt/roscpp/groovy/bin/
execution_host = localhost
check_execution = True
criteria = ,correctly setup to run MORSE
run_order = 5
result_assessment_order = 5

[run]
name = MORSE_Test
run_order = 5
run_execution_duration = 40
result_assessment_order = 5
result_assessment_execution_duration = 40
a) Establish experiment prototyping. This uses simulation environments including a virtual human component. 

b) Extend the concept of an experiment protocol to the orchestration of software components, and 

c) Execute and assess the results of a prototype experiment in an automated, easy-to-use fashion.
FINITE-STATE-MACHINE-BASED-TESTING
FINITE-STATE-MACHINE-BASED-TESTING
FINITE-STATE-MACHINE-BASED-TESTING

Provided benefits
No Framework “lock-in"
Human readable *.ini syntax
Automated translation into a SM
Already includes evaluation scripts

Provided benefits
Archiving of results
Early and continuous feedback
Centralized testing infrastructure
Gradually increase level of simulation complexity
FINITE-STATE-MACHINE-BASED-TESTING

Project FSMT-Flobi-Simulation-Motion-Test
Automated execution and evaluation of an the FloBi simulation using FSMT.

Workspace
Last Successful Artifacts
View of Simulation

FloBi_Head_Position

Joint angle vs. Time in seconds

Graph showing the movement of a joint over time.
LIVE Demo :)
Of course we will make FSMT open soon

https://opensource.cit-ec.de
Thank you!
REFERENCES


