

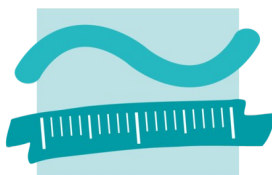
Praxisprojekt

Automatisiertes Software Testing



Katharina Ziegler
Matrikel Nummer 791521
Studiengang
Medieninformatik Bachelor online
WS 2016/17
Korbinianstraße 18
82515 Wolfratshausen
Tel 017623494763
lichttechnik.ziegler@web.de

Firma:
MA Lighting Technology GmbH
Betreuer: Gerhard Krude
Dachdeckerstraße 16
97297 Waldbüttelbrunn
Tel 0931497940
gerhard.krunde@malighting.de



BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences



Abstract

Dieser Bericht beschreibt mein Praxisprojekt bei der Firma MA Lighting Technology GmbH. Meine Aufgabe ist es durch die Erweiterung eines vorhandenen Software-Tests Wechselwirkungen mit bereits vorhandenen Funktionen der Konsolen-Software aufzudecken, die durch Weiterentwicklung entstehen können. Dieser Regressionstest wird über XML-Dateien in die Konsole geladen und führt in der Skriptsprache Lua geschriebene Befehlsfolgen aus.

Prüfungen zu Änderungen oder Erweiterungen werden im Betrieb hauptsächlich durch menschliche Tester durchgeführt. Dies ist in der Zukunft auch nicht wegzudenken.

Angesichts der enorm umfangreichen Funktionen bleibt die Wahrscheinlichkeit für eine vollständige Abdeckung gering. Das Ziel ist die Weiterentwicklung einer Konsole, die bei ihrem Einsatz, auf keinen Fall durch Software-Schwächen auffällt und damit eventuell eine Großveranstaltung lahm legen könnte.

Einführung

In der Lichttechnik im Bereich der Veranstaltungstechnik ist eine Steuerung der eingesetzten Lampen, sowie sonstigen Geräten, wie zum Beispiel Nebelmaschinen, nötig um verschiedene Lichtstimmungen zu erzeugen, wiederzugeben oder zu manipulieren. Diese Steuerung beinhaltet vielfältige Funktionen von Dimmen über Farbwechsel bis hin zu Bewegungen, je nach verwendeter Lampenart.

Hierbei wird eine zentrale Steuereinheit, ein Lichtstellpult, meist über Kabel mit den Geräten im Daisy-Chain-Prinzip verbunden.

Das Steuersignal wird mit dem Protokoll DMX 512 übertragen. DMX bedeutet Digital Multiplex und ist ein digitales Steuerprotokoll, welches sich zum Standard in der Veranstaltungstechnik entwickelt hat. Hier können 512 Kanäle in einer Auflösung von 8 oder 16 Bit genutzt werden.

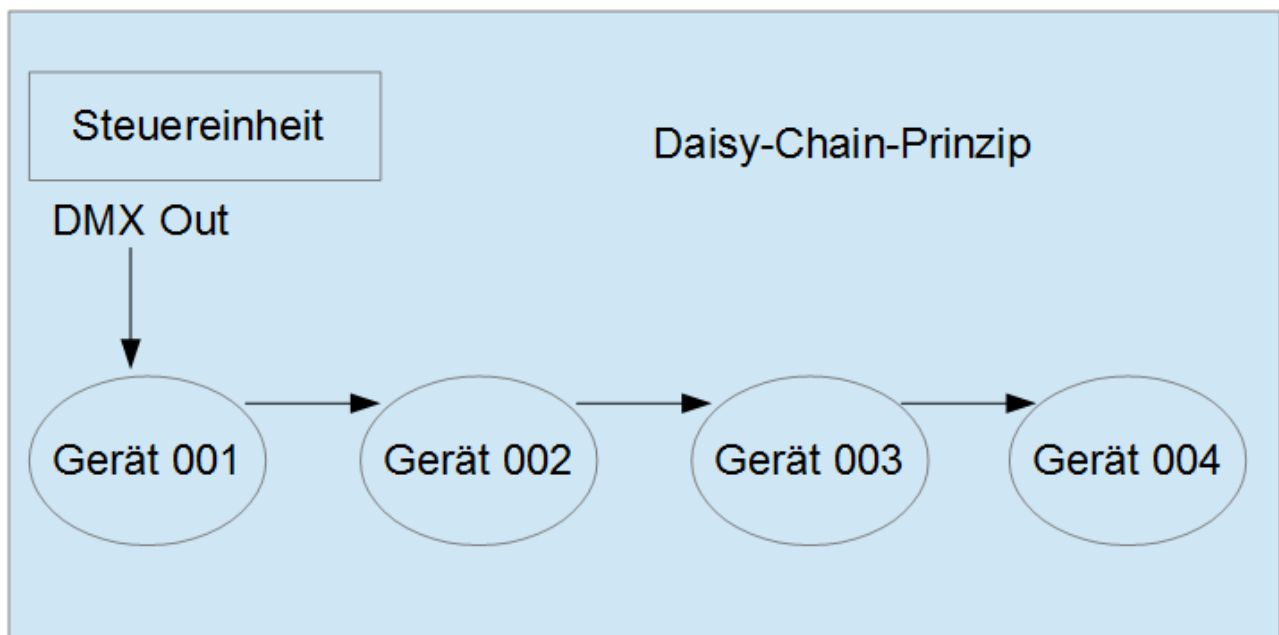


Abb. 1: Das Daisy-Chain-Prinzip

Oft genutzte Lampen sind die sogenannten Moving Lights. Ein Moving Light bewegt über einen elektromechanischen Antrieb den Lichtstrahl und verfügt über Funktionen wie Farbwechsel und Grafikeffekte. Ein Martin Mac Viper Performance, als Beispiel für ein oft

genutztes Moving Light, benötigt 32 Kanäle für seine Funktionen. Deshalb vervielfältigen sich die Kanäle in DMX- Universen. Ein DMX-Universum verfügt über 512 Kanäle. Die aktuelle Lichtsteuerkonsole grandMA2 von MA Lighting ist eine Echtzeitsteuerung für bis zu 65 536 Parametern pro Session in Verbindung mit einer MA NPU (Network Processing Unit). Dies entspricht 256 Universen.

Die Firma MA Lighting ist einer der marktführenden Hersteller von computergesteuerten Lichtstellpulten. 1983 gegründet von Michael Adenau, werden verschiedene Produkte zur Steuerung diverser Medien, die ihren Einsatz in der Showtechnik finden, entwickelt und hergestellt.

Die Produktpalette erstreckt sich von Geräten der Netzwerktechnik über Dimmer, hin zu Lichtsteuerkonsolen. Das Falgschiff der aktuellen Lichtstellkonsolen-Serie ist die grandMA2 fullsize. Das grandMA2 OnPC ist das kostenlose Windows-Derivat der Konsolen-Software.



Abb. 2: grandMA2 fullsize



Abb. 3: grandMA2 onPC

Ablauf und Zeitplan

An zwei Tagen pro Woche editiere und ergänze ich die Funktionstests. Um die zu testenden Funktionen in korrekte Testabläufe zu integrieren, ist es unumgänglich die Funktionsweisen und die entsprechenden Befehlsfolgen der Konsole zu kennen. Deshalb treffe ich Klaus Rupprath, Tester und MA Trainer, und lerne die Grundfunktionen und im Speziellen, was für meinen aktuellen Test erforderlich ist.

Der Kollege Oliver Becker hatte die Anfänge der Tests betreut, wurde aber einem anderen Projekt zugewiesen. Er ist ein guter Ansprechpartner für meine anfänglichen Fragen.

Der Testrahmen und -treiber wurden von Senior Developer Olaf Reusch geschrieben.

Die ersten 10 Tage bestanden aus Lernen der Befehlsstruktur für die Konsole und einlesen in die vorhandenen Skripte und Dokumentationen.

Die Grundfunktionen sollten im November abgeschlossen sein.

Anschließend warten weitere umfangreichere Tests auf mich.

Aufbau und Funktionsweise der Testumgebung

Die Software der grandMA2

Die Software der grandMA2, beziehungsweise des onPCs, ist in C++ geschrieben.

Über die eingebettete Skriptsprache Lua können Funktionen über die Command Line aufgerufen werden oder ganze Abläufe initiiert werden.

Über das Command Line Feedback ist ein Protokoll der Eingaben zu sehen. Ebenso kann man über den list -Befehl eine Übersicht der Struktur einsehen und über cd .. oder cd / durch diese navigieren.

Des Weiteren gibt es den System Monitor, welcher alle systemrelevanten Ereignisse ausgibt. Das Betriebssystem der Konsole ist Linux.

Die Schnittstelle zwischen grandMA2 Software und Testskript

Über XML-Dateien können in die grandMA2 PlugIns geladen werden, welche wiederum auf LUA-Dateien verweisen, die mit der Anwahl des jeweiligen PlugIns ausgeführt werden.

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <MA xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://schemas.malighti
3   <Info datetime="2012-03-26T11:48:43" showfile="lua_test" />
4
5   <Plugin index="0" luafile="test_init_fixtures.lua" name="ShowfileInit"/>
6
7   <Plugin index="1" luafile="test_cancel.lua" name="Tests|Cancel"/>
8   <Plugin index="2" luafile="test_main_matrix.lua" name="Run|Tests"/>
9
10  <Plugin index="3" luafile="test_channel_delete.lua" name="Channel|Delete" />
11  <Plugin index="4" luafile="test_channel_assign.lua" name="Channel|Assign"/>
12  <Plugin index="5" luafile="test_channel_at.lua" name="Channel|At"/>
13  <Plugin index="6" luafile="test_channel_clone.lua" name="Channel|Clone"/>
14  <Plugin index="7" luafile="test_channel_highlight.lua" name="Channel|Highlight" />
15
16  <Plugin index="8" luafile="test_cue_store.lua" name="Cue|Store"/>
17  <Plugin index="9" luafile="test_cue_delete.lua" name="Cue|Delete"/>
18  <Plugin index="10" luafile="test_cue_at.lua" name="Cue|At"/>
19  <Plugin index="11" luafile="test_cue_call.lua" name="Cue|Call"/>
20  <Plugin index="12" luafile="test_cue_copy.lua" name="Cue|Copy"/>
21  <Plugin index="13" luafile="test_cue_edit.lua" name="Cue|Edit"/>
22  <!--Plugin index="14" luafile="test_cue_kill.lua" name="Cue|Kill"/-->
23  <Plugin index="15" luafile="test_cue_move.lua" name="Cue|Move"/>
24  <Plugin index="16" luafile="test_cue_selfix.lua" name="Cue|SelfFix"/>
25  <Plugin index="17" luafile="test_cue_update.lua" name="Cue|Update"/>
26
27  <Plugin index="18" luafile="test_cuepart_store.lua" name="CuePart|Store"/>
28  <Plugin index="19" luafile="test_cuepart_delete.lua" name="CuePart|Delete"/>
29  <Plugin index="20" luafile="test_cuepart_at.lua" name="CuePart|At"/>
30  <Plugin index="21" luafile="test_cuepart_call.lua" name="CuePart|Call"/>
31  <Plugin index="22" luafile="test_cuepart_copy.lua" name="CuePart|Copy"/>
32  <Plugin index="23" luafile="test_cuepart_edit.lua" name="CuePart|Edit"/>
33  <Plugin index="24" luafile="test_cuepart_move.lua" name="CuePart|Move"/>
34  <Plugin index="25" luafile="test_cuepart_selfix.lua" name="CuePart|SelfFix"/>
35  <Plugin index="26" luafile="test_cuepart_update.lua" name="CuePart|Update"/>
```

Abb.4: test_main_matrix.xml repräsentiert die Verknüpfung mit den LUA-Dateien

Plugin	1 LUA	2 ShowfileInit	3 Tests Cancel	4 Run Tests	5 Channel Delete	6 Channel Assign	7 Channel At	8 Channel Clone	9 Channel Highlight
10 Cue Store	11 Cue Delete	12 Cue At	13 Cue Call	14 Cue Copy	15 Cue Edit	16 Cue Move	17 Cue SelfFix	18 Cue Update	19 CuePart Store
20 CuePart Delete	21 CuePart At	22 CuePart Call	23 CuePart Copy	24 CuePart Edit	25 CuePart Move	26 CuePart SelfFix	27 CuePart Update	28 DMX Assign	29 DMX Delete
30 DMXS Store	31 DMXS Delete	32 DMXS Copy	33 DMXS Move	34 DMXS Update	35 Fixture Delete	36 Fixture Assign	37 Fixture At	38 Fixture Clone	39 Fixture Highlight

Abb.5: PlugIn Pool nach Import Test_main_matrix.xml

Load Show- eine Test-Show laden

Mir steht ein leeres Showfile zur Verfügung, welches ich für die Testausführung benutzte.

PlugIns und XML Dateien

Die Datei `test_init_fixtures.lua` wird durch das PlugIn „Showfileinit“ repräsentiert.

Das bedeutet, dass durch einen Klick auf das PlugIn diese Datei ausgeführt wird.

`test_init_fixtures.lua` wiederum integriert `property.xml` und `properties.xml`.

`Properties.xml` enthält die Werte mit welchen getestet werden kann. Weiter enthalten ist ein Server Flag, eine Versionsnummer, einen boolean-Wert, der angibt, ob Vergleichsobjekte generiert werden sollen, sowie die Bezeichnungen und Anzahlen für die Dimmer und Lampen, welche in den Tests angesprochen werden.

`Property.xml` erfüllt einen ähnlichen Zweck und enthält eine destruktive table, eine Tabelle, welche die Namen der Tests enthält, die die Testumgebung invalide machen würden indem grundlegende Objekte gelöscht oder verschoben werden, die für anschließende Tests noch gebraucht werden.

Das PlugIn „Showfileinit“ wird nach dem Laden des Showfiles ausgeführt um eine Menge von Lampen und Dimmern zu adressieren (patchen). Mit diesen Objekten kann dann getestet werden.

Das PlugIn „Tests cancel“ führt die Datei `test_cancel.lua` aus und dient dem Abbrechen des Testdurchlaufs.

Das PlugIn „Run Test“ startet den gesamten Testdurchlauf indem `test_main_matrix.lua` ausgeführt wird. Die weiteren PlugIns repräsentieren die einzelnen Tests.

Das PlugIn Nummer 1 ist gesperrt und wird für die Lua-Engine genutzt.

Die Datei `test_main_matrix.xml` ist eine Auflistung der Einzelskripte.

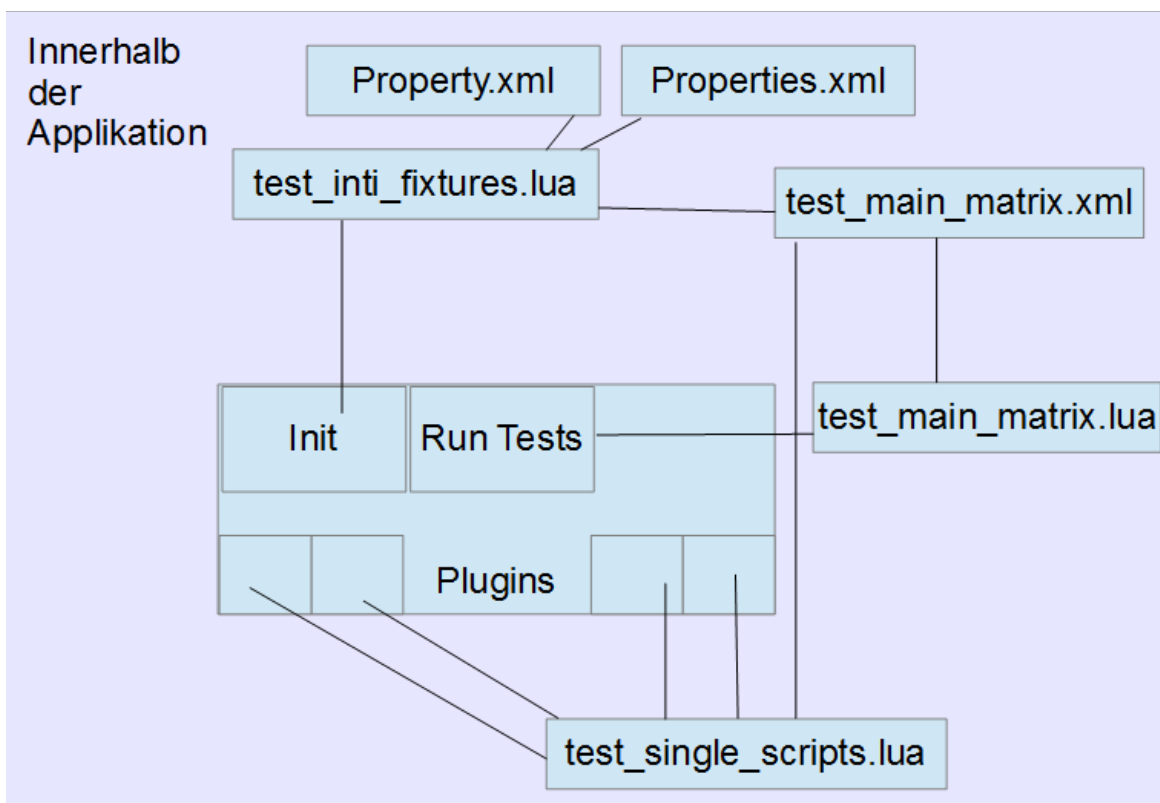


Abb.6: Zusammenhang der einzelnen Dateien als Testumgebung

Die Testmatrix

Der Testrahmen und -treiber wird von der Datei `test_main_matrix.lua` repräsentiert und umfasst zu meinem Einstiegszeitpunkt 1712 Zeilen Code und ist äußerst sparsam kommentiert. Auffällig ist auch, dass zwei relativ große Bereiche im Code identisch sind mit denen in der Datei `test_init_fixtures.lua`. Dies ist in der Programmierung nicht zulässig. Da das Testsystem in dieser Form zum Einsatz kommt und gewartet wird, vermute ich, dass es hierfür einen triftigen Grund geben muss.

```
800     local gs_res = nil;
801     if(handle and not fdw) then
802         if(not golden_handle) then
803             if (create_missing_golden_samples and not blacktest) then
804                 -- try to create a golden sample in the first attempt
805                 if ask_every_single_time and not host_version == "3.1.2.5" then
806                     gs_res = gma.gui.confirm("LUA Processing", "Do you want to create a golden sample for " .. self:name() .. "?");
807                 else
808                     gs_res = true;
809                 end
810             if gs_res then
811                 F("Trying to create my own golden sample");
812                 if ON == "CUE" then
813                     if dev then
814                         F('Creating Golden Sample silent...');
815                     end
816                     errMsg = C("COPY " .. ON .. " " .. test_index .. " AT " .. verify_param.gold .. " /nc");
817                 else
818                     if self.object_name == "CUEPART" then
819                         errMsg = C("COPY " .. ON .. " " .. test_index .. " AT " .. ON2 .. " " .. verify_param.gold .. " /nc");
820                     else
821                         errMsg = C("COPY " .. ON .. " " .. test_index .. " AT " .. verify_param.gold);
822                     end
823                 end
824                 if self.object_name == "CUEPART" then
825                     C("LABEL " .. ON2 .. " " .. verify_param.gold .. " " .. "" .. self:name() .. "");
826                 else
827                     C("LABEL " .. ON .. " " .. verify_param.gold .. " " .. "" .. self:name() .. "");
828                 end
829             end
830             if(errMsg) then
831                 if(dev) then
832                     E("CMD-Error: " .. errMsg);
833                 end
834                 return "Abort by user", gs_res;
835             end
836         end
837     end
838
839     if(ON2) then
840         golden_handle = G.handle(ON2 .. " " .. verify_param.gold);
```

Abb . 7: Ausschnitt der Datei `test_main_matrix.lua`, hier die Stelle an der die Testobjekte kopiert werden

Die einzelnen Testskripte

```
1  __*****
2  -- This script tests to release a fixture
3  -- Seq 131, Executor 40, DMXsnap 258
4  __*****
5
6  local fixturecount = get_prop_fixturecount();
7
8  local test=
9  {
10     method_name     = "RELEASE";
11     object_name      = "FIXTURE";
12     pre              = 'ClearAll; Delete Sequence 131 /nc; delete DMXsnapshot 258 /nc; delete Executor 40 /nc';
13
14     steps =
15     {
16         {
17             pre = 'Fixture 1 thru '.. fixturecount[2] '..' at 0 thru 100; Store sequence 131 cue 1 /nc; assign sequence 131 at Executor 40';
18             cmd = 'Release Fixture 1; Fixture 1 thru '.. fixturecount[2] '..' at 50; store sequence 131 cue 2; clearall; go executor 40';
19             post = 'label sequence 131 "REL_FIX_1"; label executor 40 "REL_FIX_1"; label DMXsnapshot 258 "REL_FIX_1";
20             cleanup = 'off executor thru; Off sequence thru; off channel thru; off fixture thru';
21             test_dmx=true;
22             test = 258;
23             gold = 758;
24         }
25     };
26 };
27
28 RegisterTestScript(test);
29
30 __*****
31 -- module entry point
32 __*****
33
34 local function StartThisTest()
35     StartSingleTestScript(test);
36 end
37
38 return StartThisTest;
```

Abb.8: Screenshot eines Einzelskripts, welche alle im Aufbau identisch sind

Die Verifizierung

Bei der grandMA2 kann am DMX Ausgang der Output gemessen werden. Diese Messungen lassen sich in DMX-Snapshots speichern und im DMX-Snapshot-Pool ablegen.

Als Golden Samples werden sie beim ersten Durchlauf des Gesamttests durch Kopieren erstellt und dienen als Vergleichsobjekte.

Objekte, die nicht auf einen DMX-Output zurückzuführen sind, wie zum Beispiel Sequenzen oder Gruppen, können auch als solche verglichen werden. Ihre Golden Samples liegen in den entsprechenden Pools.

Der Code für den Vergleich befindet sich in der Hauptsoftware.

Wie beeinflusst die geladene Show den Test?

Belegte Ressourcen, wie bereits vorhandene DMX-Snapshots werden beim erneuten Ausführen nicht überschrieben. Das bedeutet, dass diese als Golden Samples verwendet werden können. Änderungen an den Befehlsfolgen oder Werten sollten zur Folge haben, dass der Vergleich negativ ist und der Testfall somit als failed gekennzeichnet wird. Aber auch, dass die Testobjekte eine freie Position vorfinden müssen um ihr Vergleichsobjekt zu speichern. Das wird sichergestellt, indem zu Beginn eines jeden Einzeltests skripts die verwendeten Ressourcen initial gelöscht werden. Auch muss sichergestellt sein, dass kein anderer Output als der des Testobjekts den Vergleich stört. Das wiederum wird

gewährleistet, indem jedes Einzelskript in einem Cleanup-Befehl alle verwendeten Komponenten wieder ausschaltet.

Run Tests- die gesamten Tests ausführen

Der gesamte Testdurchlauf geht durch alle Einzelskripte und dokumentiert in einer Zusammenfassung am Schluss die vorhandenen Fehler.



```
10h46m28.750s : All test done.
10h46m28.760s :
10h46m28.760s : Duration: 3 min 34 sec
10h46m28.760s : Summary: 25 of 267 Scripts in the Testmatrix failed!
10h46m28.760s :   STORE_CUE STEP 9 VERIFY ERROR (106.2): golden comparison failed
10h46m28.760s :   STORE_VIEW STEP 2 VERIFY ERROR (104): golden comparison failed
10h46m28.760s :   STORE_WORLD STEP 1 VERIFY ERROR (115): golden comparison failed
10h46m28.760s :   DELETE_AGENDA STEP 1 VERIFY ERROR (-1): golden comparison with an empty object
10h46m28.760s :   DELETE_CUE STEP 3 TEST ERROR : CMD-Error: OBJECT DOES NOT EXIST
10h46m28.760s :   DELETE_MACRO_SCRIPT_PREPARE ERROR : OBJECT DOES NOT EXIST
10h46m28.761s :   DELETE_VIEW_SCRIPT_PREPARE ERROR : OBJECT DOES NOT EXIST
10h46m28.761s :   ASSIGN_AGENDA STEP 1 VERIFY ERROR (5): golden comparison failed
10h46m28.761s :   COPY_AGENDA STEP 1 PREPARE ERROR : CMD-Error: RESIZE FORBIDDEN
10h46m28.761s :   COPY_CUE STEP 6 VERIFY ERROR (106.1): golden comparison failed
10h46m28.761s :   COPY_MACRO_SCRIPT_PREPARE ERROR : OBJECT DOES NOT EXIST
10h46m28.761s :   COPY_VIEW_SCRIPT_PREPARE ERROR : OBJECT DOES NOT EXIST
10h46m28.761s :   COPY_WORLD STEP 1 VERIFY ERROR (101): golden comparison failed
10h46m28.761s :   REMOVE_SEQUENCE STEP 1 VERIFY ERROR (101.2): golden comparison failed
10h46m28.761s :   EDIT_WORLD STEP 1 VERIFY ERROR (105): golden comparison failed
10h46m28.761s :   EXPORT_MACRO STEP 2 VERIFY ERROR (105): golden comparison failed
10h46m28.761s :   EXPORT_PRESET STEP 2 VERIFY ERROR (1.110): golden comparison failed
10h46m28.761s :   EXPORT_WORLD STEP 2 VERIFY ERROR (107): golden comparison failed
10h46m28.761s :   IMPORT_EFFECT STEP 1 VERIFY ERROR (107): golden comparison failed
10h46m28.761s :   IMPORT_PRESET STEP 1 VERIFY ERROR (1.111): golden comparison failed
10h46m28.761s :   IMPORT_SEQUENCE STEP 1 VERIFY ERROR (211): golden comparison failed
10h46m28.761s :   IMPORT_WORLD STEP 1 VERIFY ERROR (108): golden comparison failed
10h46m28.761s :   MOVE_WORLD STEP 1 VERIFY ERROR (113): golden comparison failed
10h46m28.761s :   UPDATE_WORLD STEP 1 VERIFY ERROR (119): golden comparison failed
10h46m28.761s :   INSERT_WORLD STEP 1 VERIFY ERROR (109): golden comparison failed
10h46m28.761s :
10h46m28.761s : Destructive tests had been skipped.
```

Abb.9: Screenshot der Dokumentation des Gesamtdurchlaufs nach der Initialisierung

Meine Arbeit an diesem System

Organisation und Planung des Testens

Beim Einlesen in die Skripte war schnell klar, dass die bereits vorhandenen Tests in lexikographischer Ordnung angelegt waren. Auf den ersten Blick sehr übersichtlich und strukturiert. Auf den zweiten Blick sehr aufwendig neue Tests einzupflegen, denn damit mussten die Skripte in ihrem Index verschoben werden, was ein aufwendiges manuelles Nachnummerieren bedeutete.

Auch war im Dialogfenster der Software während des Tests zu sehen, dass der Testablauf nach Methoden in alphabetischer Reihenfolge ausgeführt wurde.

Mit Ausnahme der Funktionen „Store“ und „Delete“, welche für jedes Objekt vorrangig behandelt wurden.

Diese Vorgehensweise ist für einen Testaufbau äußerst ungünstig gewählt, weil Abhängigkeiten nicht beachtet werden.

Ein sinnvollerer Aufbau wäre nach Objekten und deren Abhängigkeiten angelegt.

Umstrukturierung des Testablaufs

Da der Testrahmen nach Methoden aufgebaut ist, bedeutet eine Umstellung eine weitgehende Änderung des Testrahmens.

In Absprache mit Kollege Oliver Becker wird entschieden, dass eine Änderung dieser zwar nötig aber nicht dringend ist, weil momentan eine funktionierende Testumgebung vorhanden ist und eingesetzt werden kann.

Deshalb wird der Ergänzung der fehlenden Einzelskripte Vorrang erteilt.

Anordnung der einzelnen Komponenten

Ziel: Übersichtlichkeit und einfache Erweiterbarkeit in der Zukunft.

Durch Erstellung einer Tabelle mit der Nummerierung der Ressourcen und Komponenten verschaffe ich mir einen Überblick. Da nun die Anzahl der Tests stetig steigt, entscheide ich mich die Bereiche der Nummerierung zu vergrößern und entsprechend zu verschieben. Das muss nach wie vor in manueller Tätigkeit erledigt werden.

Siehe Tabelle Nummerierung im Anhang.

Nummerierung der PlugIns für die Einzelskripte

Die Index-Angabe in test_main_matrix.xml wird nicht verwendet und ist daher trivial.

Die PlugIns werden chronologisch geladen.

Fehlersuche anhand der Testdokumentation

Die als failed eingestuften Testskripte haben keine für mich ersichtliche Fehlerursache.

Untersuchen der Determiniertheit des Systems

Bereits beim Schreiben der fehlenden Einzelskripte, merkte ich, dass es möglich ist das System in einen Zustand zu überführen, in dem das Ergebnis des Einzeltests immer „golden comparision failed“ ist.

Ergo ist es möglich die benutzten Komponenten, wie Sequenzen oder Exekutoren, aber auch die DMX-Snapshots, so zu belegen, dass der Vergleich nicht erfolgreich ist.

Auch auffällig ist, dass das Ergebnis des gesamten Testdurchlaufs bei der ersten Ausführung deutlich anders ist als beim wiederholten Ausführen. Es ist nicht möglich zweimal hintereinander das selbe Ergebnis zu erzeugen.

```
10h40m46.699s : Duration: 4 min 0 sec
10h40m46.699s :
10h40m46.699s : Summary: 7 of 267 Scripts in the Testmatrix failed!
10h40m46.699s :   EXPORT_MACRO STEP 2 VERIFY ERROR (105): golden comparison failed
10h40m46.699s :   EXPORT_PRESET STEP 2 VERIFY ERROR (1.110): golden comparison failed
10h40m46.699s :   EXPORT_WORLD STEP 2 VERIFY ERROR (107): golden comparison failed
10h40m46.699s :   IMPORT_EFFECT STEP 1 VERIFY ERROR (107): golden comparison failed
10h40m46.699s :   IMPORT_PRESET STEP 1 VERIFY ERROR (1.111): golden comparison failed
10h40m46.699s :   IMPORT_SEQUENCE STEP 1 VERIFY ERROR (211): golden comparison failed
10h40m46.699s :   IMPORT_WORLD STEP 1 VERIFY ERROR (108): golden comparison failed
10h40m46.699s :
10h40m46.699s :   Destructive tests had been skipped.
10h40m46.699s :
10h40m46.699s : 9 of 267 Scripts were ignored:
10h40m46.699s :   DELETE_CHANNEL (destructive)
10h40m46.699s :   DELETE_DMX (destructive)
10h40m46.699s :   DELETE_DMXUNIVERSE (destructive)
10h40m46.699s :   DELETE_FIXTURE (destructive)
10h40m46.699s :   DELETE_SELECTION (destructive)
10h40m46.699s :   ASSIGN_DMX (destructive)
10h40m46.699s :   EXPORT_PLUGIN
10h40m46.699s :   IMPORT_PLUGIN
10h40m46.699s :   MOVE_DMXUNIVERSE (destructive)
10h40m46.699s :
10h40m46.699s : Also the following 8 Steps and 3 Scripts were skipped:
10h40m46.699s :   STORE_EXECUTOR: Version SKIP : Scriptversion isn't compatible with Host System. Script will be skipped.
10h40m46.699s :   DELETE_PRESET STEP 4: Version SKIP : Stepversion isn't compatible with Host System. Step will be skipped.
10h40m46.699s :   EDIT_EFFECT: Version SKIP : Scriptversion isn't compatible with Host System. Script will be skipped.
10h40m46.699s :   MOVE_CUE STEP 2: Version SKIP : Stepversion isn't compatible with Host System. Step will be skipped.
10h40m46.699s :   MOVE_CUE STEP 4: Version SKIP : Stepversion isn't compatible with Host System. Step will be skipped.
10h40m46.699s :   MOVE_CUEPART STEP 3: Version SKIP : Stepversion isn't compatible with Host System. Step will be skipped.
10h40m46.699s :   MOVE_CUEPART STEP 4: Version SKIP : Stepversion isn't compatible with Host System. Step will be skipped.
10h40m46.699s :   MOVE_DMXSNAPSHOT STEP 3: Version SKIP : Stepversion isn't compatible with Host System. Step will be skipped.
10h40m46.699s :
[Channel]>
```

Abb.10: Erster Durchlauf nach der Initialisierung

```
10h23m28.222s : Summary: 26 of 267 Scripts in the Testmatrix failed!
10h23m28.222s :   STORE_AGENDA STEP 1 VERIFY ERROR (2): golden comparison failed
10h23m28.222s :   STORE_CUE STEP 8 VERIFY ERROR (105.1): golden comparison failed
10h23m28.222s :   STORE_MACRO SCRIPT PREPARE ERROR : OBJECT DOES NOT EXIST
10h23m28.222s :   STORE_VIEW SCRIPT PREPARE ERROR : OBJECT DOES NOT EXIST
10h23m28.223s :   STORE_WORLD STEP 1 VERIFY ERROR (115): golden comparison failed
10h23m28.223s :   DELETE_AGENDA STEP 1 VERIFY ERROR (-1): golden comparison with an empty object
10h23m28.223s :   DELETE_CUE STEP 3 TEST ERROR : CMD-Error: OBJECT DOES NOT EXIST
10h23m28.223s :   DELETE_MACRO SCRIPT PREPARE ERROR : OBJECT DOES NOT EXIST
10h23m28.223s :   DELETE_VIEW SCRIPT PREPARE ERROR : OBJECT DOES NOT EXIST
10h23m28.223s :   ASSIGN_AGENDA STEP 1 VERIFY ERROR (5): golden comparison failed
10h23m28.223s :   COPY_AGENDA STEP 1 PREPARE ERROR : CMD-Error: RESIZE FORBIDDEN
10h23m28.223s :   COPY_CUE STEP 6 VERIFY ERROR (106.1): golden comparison failed
10h23m28.223s :   COPY_MACRO SCRIPT PREPARE ERROR : OBJECT DOES NOT EXIST
10h23m28.223s :   COPY_VIEW SCRIPT PREPARE ERROR : OBJECT DOES NOT EXIST
10h23m28.223s :   EXPORT_WORLD STEP 1 VERIFY ERROR (101): golden comparison failed
10h23m28.223s :   EDIT_WORLD STEP 1 VERIFY ERROR (105): golden comparison failed
10h23m28.223s :   EXPORT_MACRO STEP 2 VERIFY ERROR (105): golden comparison failed
10h23m28.223s :   EXPORT_PRESET STEP 2 VERIFY ERROR (1.110): golden comparison failed
10h23m28.223s :   EXPORT_WORLD STEP 2 VERIFY ERROR (107): golden comparison failed
10h23m28.224s :   IMPORT_EFFECT STEP 1 VERIFY ERROR (107): golden comparison failed
10h23m28.224s :   IMPORT_PRESET STEP 1 VERIFY ERROR (1.111): golden comparison failed
10h23m28.224s :   IMPORT_SEQUENCE STEP 1 VERIFY ERROR (211): golden comparison failed
10h23m28.224s :   IMPORT_WORLD STEP 1 VERIFY ERROR (108): golden comparison failed
10h23m28.224s :   MOVE_WORLD STEP 1 VERIFY ERROR (113): golden comparison failed
10h23m28.224s :   UPDATE_WORLD STEP 1 VERIFY ERROR (119): golden comparison failed
10h23m28.234s :   INSERT_WORLD STEP 1 VERIFY ERROR (109): golden comparison failed
10h23m28.234s :
10h23m28.234s :   Destructive tests had been skipped.
10h23m28.234s :
10h23m28.234s : 9 of 267 Scripts were ignored:
10h23m28.234s :
[Channel]>
```

Abb.11: Wiederholter Durchlauf mit anderem Ergebnis

Dieses Verhalten möchte ich nun dokumentieren um mich dann an den Senior Developer zu wenden und nach Erfahrungswerten zu fragen.

Es gibt die Möglichkeit das Error-Log des Testdurchlaufs zu speichern, sowie Screenshots des Desk-Status der grandMA2 anzulegen.

Der gesamte Command Line Output kann nur dokumentiert werden, indem man die Netzwerkverbindung der Software mitloggt. Hierzu habe ich Putty verwendet.

Die Logfiles befinden sich im Ordner DokuDeterminiertheit anbei.

Besonders auffällig ist, dass der Test „Delete Cue“ bei der ersten Ausführung passed ist, bei der Wiederholung den Fehler „OBJECT DOES NOT EXIST“ und bei der anschließenden Ausführung des entsprechenden Einzeltests passed ist.

Beim Test „Copy Cue“ bei der ersten Ausführung passed, bei der Wiederholung „golden comparision failed“ und bei der Ausführung des Einzeltests passed.
Andere ähnlich Beispiele liegen vor.

Mögliche Ursache für die Nichtdeterminiertheit

Ich vermute, dass die Ursache bei der Nummerierung zu suchen ist.

Beim Schreiben des Einzeltests wird manuell eine Position für den DMX-Snapshot und für das Golden Sample festgelegt. Hier habe ich einfach an der Stelle des letzten Tests angeknüpft und bemerkt, dass die Nummerierung zu diesem Zeitpunkt schon ein Durcheinander war.

Beim ersten Durchlauf werden die Golden Samples kreiert und ich habe beobachtet, dass mehrere Test die selbe Stelle verwenden. Ist dies der Fall, kommt die Ausgabe „Trying to create my own Golden Sample...“ und der aktuelle Test überschreibt nicht diese Stelle durch eine Kopie des Golden Samples.

Regressionfähigkeit

Wird festgestellt, dass die Ressource schon belegt ist, müsste abgefragt werden ob und wohin gespeichert werden soll.

Konzept für neue Nummerierung der Komponenten

Um Fehler bei der manuellen Nummerierung zu vermeiden, sollte diese automatisiert werden.

Die Schwierigkeit hierbei liegt darin, dass man den gesamten Testdurchlauf ausführen kann und die Einzeltests in beliebiger Reihenfolge.

Das bedeutet, dass die Nummerierung fest vergeben sein muss.

So ist eine Nummerierung, die aus einem Array dynamisch vergeben wird nicht praktikabel. Ein anderer Ansatz wäre eine Liste, in der man die verwendeten Objekte nummeriert und statisch lädt. Nachteil ist, dass man trotzdem wieder eine manuelle Nummerierung innerhalb der Liste vornehmen müsste. Vorteil ist, dass man erforderliche Änderungen nur an dieser einen Stelle machen müsste.

Versuch des Detektierens der Nichtdeterminiertheit

Um zu wissen, ob die Nichtdeterminiertheit an den Mehrfachbelegungen der verwendeten Ressourcen liegt oder auf eine andere, unbekannte Ursache zurück zu führen ist, habe ich die Nummerierung der Ressourcen manuell berichtigt. Das Verhalten bleibt unverändert.

Infragestellen des gesamten Tests

Nach anfänglichen Verständnisproblemen meinerseits wie die Golden Samples an welcher Stelle im Code entstehen und weshalb eine Verifizierung durch einen Vergleich mit der Kopie valide sein soll, hatte ich Gelegenheit mit dem Initiator der Lua-Tests, Andres Glad, zu sprechen. Nach einem Blick auf mein Test-Showfile stellte er fest, dass zu Beginn der Lua-Tests ein Showfile existierte, welches eine Reihe geprüfter Golden Samples enthielt. Dieses Showfile muss irgendwo abhanden gekommen sein. Weiteres entzieht sich meiner Kenntnis.

Entstehung des Komparators durch Kopieren

Beim initialen Ausführen des Gesamttests werden anschließend an die Befehlsfolge der Einzelskripte durch Kopieren des Testergebnisses die Vergleichsobjekte erstellt.

Exakt diese Kopie wird als Verifizierungsobjekt genutzt. Eventuelle Fehlfunktionen werden mit kopiert.

Lediglich die Prüfung der Gleichheit des Testobjekts und seiner Kopie macht den Test passed.

Meine Erkenntnis ist, dass dieser vorhandene Test lediglich ein Smoketest sein kann, keineswegs ein funktionierender Regressionstest, der Unregelmäßigkeiten in den Funktionen durch Weiterentwicklung der Konsole aufzeigen kann. Das bedeutet, es wird nur getestet ob die Software durch die Ausführung der Befehlsfolgen abstürzt.

Lösungskonzepte für valide Verifizierung

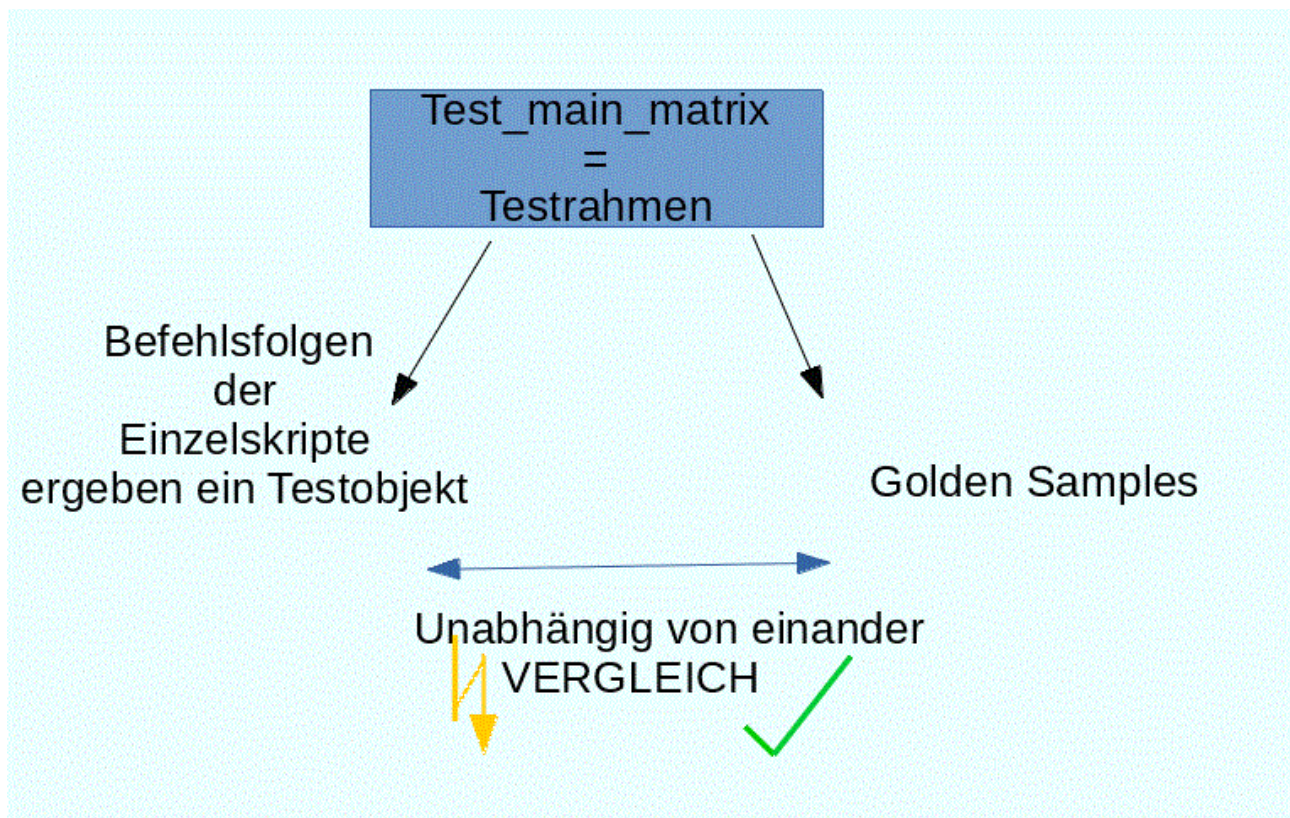


Abb. 12: valider Vergleich von unabhängigen Objekten

Ein valider Test basiert auf einem vordefinierten Verhalten der Software oder auf einem fixen Ergebnis. Diese sind das Testkriterium. Dieselben sind bei meinem Software Test nicht vorhanden.

Es gibt zwei Möglichkeiten valide Vergleichsobjekte zu schaffen. Die eine wäre die Befehlsfolgen der Einzeltests auf einer Konsole auszuführen und einen jeweiligen DMX-Snapshot zu speichern. Vorteil wäre, dass man die Funktionen optisch mit prüfen könnte indem man LED-Kacheln anschließt. Hier würde eine Datenbank an validen Verifizierungsobjekten entstehen, welche man in die Testmatrix importieren könnte. Nachteilig ist, dass weder das Importieren noch das Exportieren von DMX-Snapshots implementiert ist. Auch gibt es zu bedenken, dass es Objekte wie Gruppen oder Sequenzen gibt, die als Objekt verglichen werden und keinen direkten DMX-Output erzeugen. Der Testrahmen müsste entsprechend geändert werden und ein direkter Vergleich zwischen Testobjekt und Golden Sample geschaffen werden.

Dasselbe wäre auf dem onPC möglich mit einer Visualisierung über die Darstellungssoftware MA 3D.

Die andere Möglichkeit wäre, die Testobjekte in einem Showfile zu erzeugen und diese an eine feste Stelle zu speichern. Dieses Showfile würde als „Golden Showfile“ anschließend bis auf die Vergleichsobjekte entleert werden und dürfte nicht mehr geändert werden. Bei beiden Möglichkeiten wird davon ausgegangen, dass im Moment der Erstellung des Verifizierungsobjekts die grandMA2 Software korrekt arbeitet. Es können nur Fehlfunktionen entdeckt werden, die in der Zukunft entstehen, Bestehendes bleibt unsichtbar.

Den Test testen

Indem man den Test in eine Konsole lädt und einen bekannten Fehler testet könnte man prüfen, ob der Test diese Fehler findet. Manuell kann man auch eine Fehlfunktion in den Test schreiben und prüfen, ob es beim Vergleich mit dem Golden Sample gemeldet wird.

Den Vergleich testen

In der Hauptsoftware der grandMA2 ist der Vergleich zwischen Testobjekt und Verifizierungsobjekt implementiert.

Hier gibt es die Möglichkeit die Bitdatei der Vergleichsobjekte einzusehen.

Ich erstelle über das übliche Kopieren die Verifizierungsobjekte und ändere einen Wert in einem Einzelskript und vergleiche.



Abb. 13: Nach Anlegen des Golden Samples für den Test „Fixture full“ mit 100 % Output manuell veränderter Wert auf 50 % Output wird als identisch und der Testfall als PASSED eingestuft, hier wird sichtbar, dass die Vergleichsfunktion nicht funktioniert.

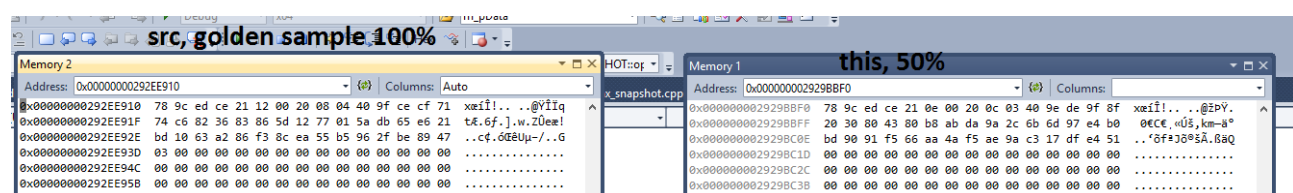


Abb.14: Bitweise Darstellung der Vergleichsobjekte. Offensichtlicher Unterschied wird als identisch interpretiert.

Da ich nicht in der Hauptsoftware arbeiten soll, wird der Fehler von Kollegen behoben.

Gewähltes Konzept für die Erstellung valider Verifizierungsobjekte

Ich ändere nun den Testrahmen in die zwei Bereiche, die Erstellung der Golden Samples und des Testens selbst.

So kann ein Golden Showfile erstellt werden, welches die Verifizierungsobjekte enthält und in das dann der Test geladen wird.

Umsetzung

Da der Vergleich über handles stattfindet ist das Vorgehen unproblematisch, solange der Testrahmen nicht neu geladen wird. Ist dies jedoch der Fall, so werden den Objekten neue handles zugewiesen und der Vergleich ist wieder bei den meisten Einzeltests failed.

Warum sind nicht alle Einzeltests failed wenn sich die handles unterscheiden?

Nach wie vor ist der Stand der Dinge, dass das System absolut nicht-determiniert ist.

Weitere Untersuchungen haben ergeben, dass beim erneuten Laden der Testobjekte neue handles vergeben werden, was den Vergleich unmöglich machen sollte. Ergo sollten theoretisch alle Testfälle failed sein, dies ist nicht so.

Wie muss der Testrahmen geändert werden, damit innerhalb eines Setups die selben handles verwendet werden?

Die Datei test_init_fixtures.lua, welche die Testumgebung vorbereitet enthält einen Property Loader und einen Property Interpreter. Der Codeteil, der den Property Loader umfasst deckt sich komplett mit dem Teil namens Property Loader in der Datei test_main_matrix.lua. Das war mir schon zu Beginn aufgefallen, jedoch wurde ich vom Kollegen gebeten dies so beizubehalten, da das File test_init_fixtures.lua auch einzeln verwendet werden sollte.

```
local handle = G.handle(ON .. " .. "..test_index"); -----handle of the test object -- Referenz auf test objekt, Handle des Testobjekts
local golden_handle = nil;

if(tonumber(verify_param.gold) > 0) then                                     --if golden handle is available
    if dev then
        if(ON2) then
            F("Handle-Request:" .. ON2 .. " .. "..verify_param.gold);
        else
            F("Handle-Request:" .. ON .. " .. "..verify_param.gold);
        end
    end
    if(ON2) then -----object name for Cueparts
        golden_handle = G.handle(ON2 .. " .. "..verify_param.gold);--concat object with golden handle
    else
        golden_handle = G.handle(ON .. " .. "..verify_param.gold); --objects which are no Cueparts -- Referenz auf GoldenSample, Handle des Gs
    end
else
    golden_handle = nil;
    F("Verification Process to an empty object...");
    if(G.compare(handle, golden_handle) == self.blacktest) then
        if(self.blacktest) then
            return "black comparison with an empty object";
        else
            return "golden comparison with an empty object";
        end
    end
end
```

Abb. 15: Handles werden zufällig vergeben, dennoch müssen sie innerhalb des Testdurchlaufs konstant bleiben.

Kleine Anzahl von Testobjekten-System ist nun determiniert

Durch auskommentieren und anpassen diverser Codeblöcke habe ich nun einen Stand erreicht, in dem sich das System mit einer niedrigen Anzahl von Einzeltests determiniert verhält. Die selben handles beibehält und mit validen Vergleichsoperationen passed ist. Nun werde ich nach und nach mehr Einzeltests dazu nehmen und beobachten wie die Wechselwirkungen sind und diese entsprechend eliminieren.

Sending an email aus der GrandMA2

Eine Anfrage aus dem Tech-Support für einen Lichtdesigner, der eine Architekturinstallation betreibt, beinhaltet, dass eine Email mit einer Statusabfrage aus der Konsole verschickt werden sollte. Diese Funktion wurde vom Entwicklerteam implementiert. Meine Aufgabe ist es nun einen Beispiel Code hinzuzufügen, welcher von den Anwendern über ein PlugIn benutzt werden kann.

Hierfür muss ich mir die Funktionsweise vom Verbindungsaufbau in Lua und die konsolenspezifischen Eigenschaften aneignen.

Einen Beispiel-Code kann man in der Lua-Dokumentation abrufen und modifizieren.

Unter Linux gibt es die Möglichkeit von der Kommandozeile mit dem Befehl sendmail eine Nachricht zu verschicken. Wenn das onPC auf einem Windows PC genutzt wird, müsste das vorhandene Email-Programm geöffnet werden.

Nun war nicht klar, ob der Kunde eine Konsole oder einen PC nutzt und ob diese sich überhaupt in einem Netzwerk befindet.

Des Weiteren muss festgelegt werden was das Versenden der Email triggert und was der Inhalt sein sollte. Beides ist in der Hauptsoftware nicht implementiert.

Nach einigem Experimentieren, auch mit Apache, und Beobachten unter wireshark, kam ich nicht wirklich weiter. Nach einer Diskussion im Software-Meeting wurde beschlossen, dass diese Funktion nicht supportet werden sollte.

Validen Testablauf ausweiten

Da ich nun den Zustand der Determiniertheit des Testsystems mit 30 Einzeltests erreicht habe, gehe ich nun schrittweise weiter durch die Einzeltests. Durch Einkommentieren der Einzeltests in der Auflistung in test_main_matrix.xml arbeite ich sukzessive den Test durch und prüfe erneut den Ablauf und den Zustand.

Das Ziel ist, möglichst bald ein Golden Showfile als Testorakel zu erzeugen und somit einen validen Test zu haben. Bevor dieses aber in Betrieb geht, möchte ich wirklich sicher sein, dass es keine groben Fehler mehr aufweist.

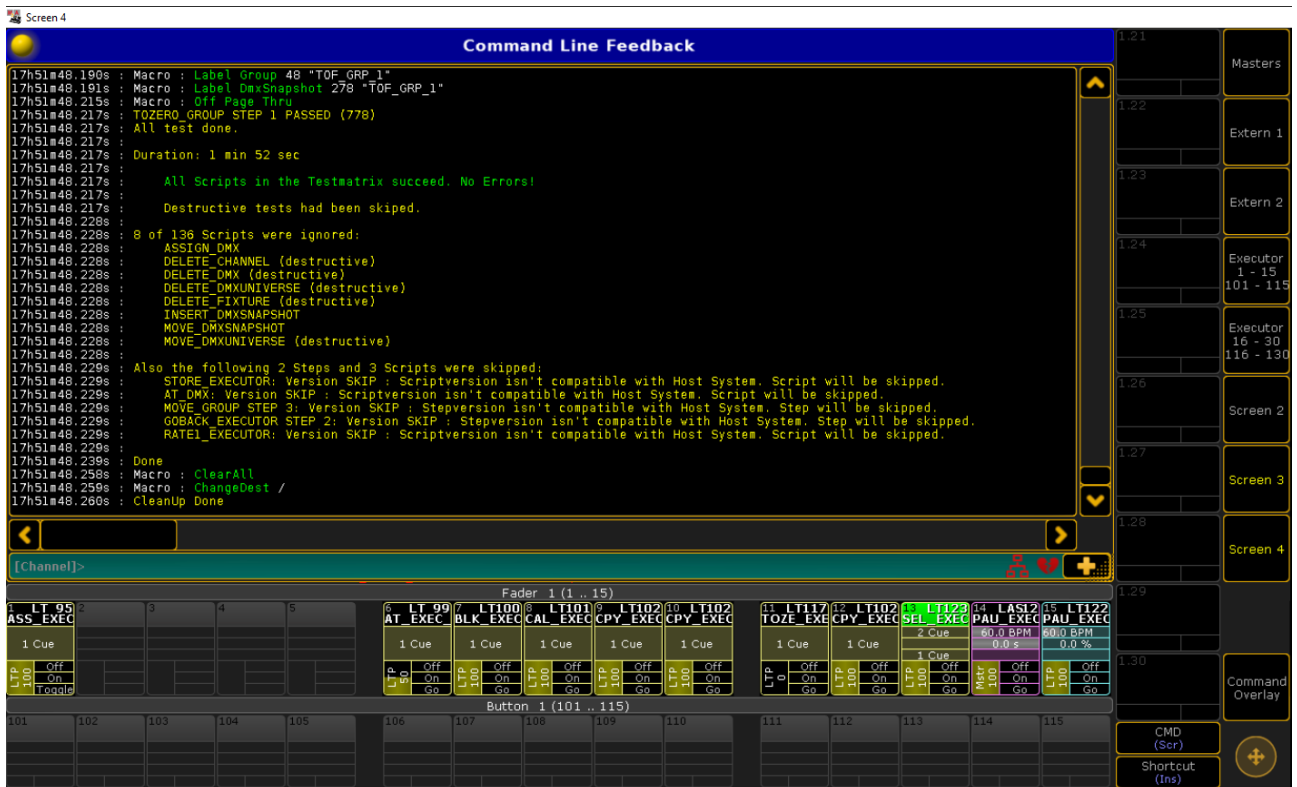
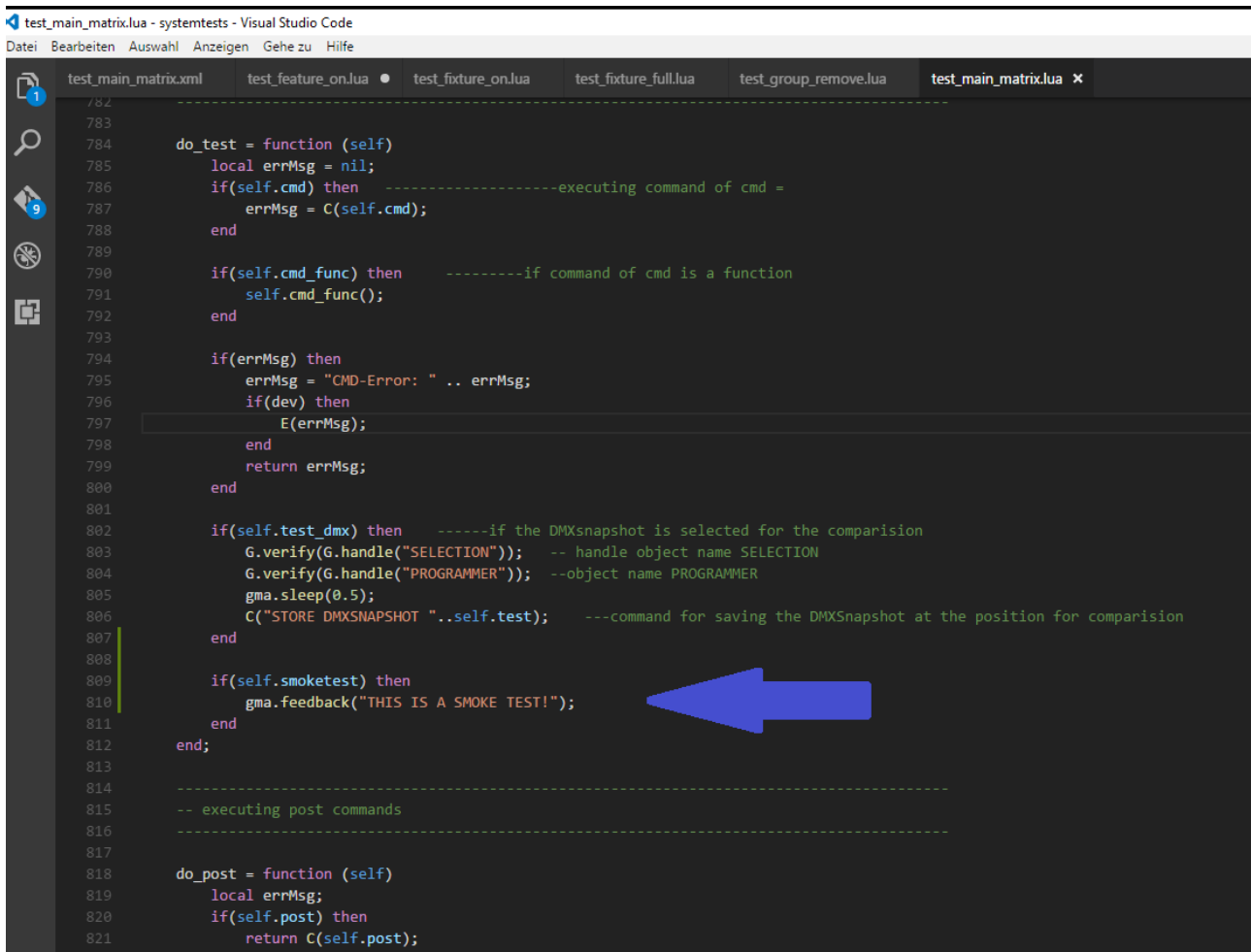


Abb. 16: Determiniertes, valides Verhalten des Testsystems

Agenda

Die grandMA2 verfügt über eine Kalenderfunktion, die zu bestimmten Terminen vorprogrammierte Abläufe ausführt. Da die Tests langfristig täglich durchlaufen sollten, kann man nicht in einem festen Testrahmen den Output für einen Termin testen. Er wäre immer failed, außer wenn der Test mit dem Termin zusammenfällt.

Die Agenda-Tests habe ich als Smoketests implementiert. Hierfür gibt es eine boolesche Variable, die eine „THIS IS A SMOKETEST!“ Message im Command Line Feedback ausgibt.



```
782
783
784 do_test = function (self)
785     local errMsg = nil;
786     if(self.cmd) then -----executing command of cmd =
787         errMsg = C(self.cmd);
788     end
789
790     if(self.cmd_func) then -----if command of cmd is a function
791         self.cmd_func();
792     end
793
794     if(errMsg) then
795         errMsg = "CMD-Error: " .. errMsg;
796         if(dev) then
797             E(errMsg);
798         end
799         return errMsg;
800     end
801
802     if(self.test_dmx) then -----if the DMXsnapshot is selected for the comparision
803         G.verify(G.handle("SELECTION")); -- handle object name SELECTION
804         G.verify(G.handle("PROGRAMMER")); --object name PROGRAMMER
805         gma.sleep(0.5);
806         C("STORE DMXSNAPSHOT "..self.test); ---command for saving the DMXSnapshot at the position for comparision
807     end
808
809     if(self.smoketest) then
810         gma.feedback("THIS IS A SMOKE TEST!");
811     end
812 end;
813
814 -----
815 -- executing post commands
816 -----
817
818 do_post = function (self)
819     local errMsg;
820     if(self.post) then
821         return C(self.post);
```

Abb.17: Feedback-Ausgabe für Tests ohne Output

DMX Tester und DMX-Snapshots

In den Einzeltests enthalten sind auch Tests für die Grundlagen wie DMX-Snapshots oder DMX-Outputs, die unabhängig von der eigentlichen Programmierung der Konsole sind. Sollte zum Beispiel die Funktion des Speicherns oder des Kopierens eines DMX-Snapshots defekt sein, ginge der ganz Test nicht.

Hier ist man an einem Punkt, wo ich sagen muss, dass gewisse Funktionen als vorhanden betrachtet werden müssen, sonst geht der Testablauf nicht.

Dies ist ein klarer Nachteil davon, dass der Testablauf innerhalb der Konsole stattfindet. Würde die Konsole die Befehlsfolgen ausführen und einen Ergebniswert ausgeben, der außerhalb weiterverarbeitet wird, wäre dieses Problem vermutlich weniger bedeutend.

Der DMX-Snapshot ist der bereits gespeicherte Output. Diesen kann man zwar vergleichen, aber nicht abspielen, in diesem Sinne. Die Funktionen MOVE oder COPY kommen zwar als Test vor, könnten aber die DMX-Snapshots im DMX-Snapshot-Pool verschieben und damit das Testorakel beschädigen. Deshalb habe ich diese im ignore table in property.xml hinzugefügt. Das bedeutet, dass sie übersprungen werden. Dazu habe ich noch eine Ausgabe eingefügt, falls man diese als Einzeltests manuell ausführt, nach der Ausführung der Test nicht mehr valide sein könnte.

DMX Tester

Mit dieser Funktion kann man DMX-Output erzeugen, ohne die programmierbaren Komponenten der Konsole zu nutzen. Diese Ausgabe hat die höchste Priorität. Ein Grund für die Nichtdeterminiertheit war, dass einer dieser Tests unbemerkt weiter Output sendete und damit die anderen Tests beeinflusste. Mir war diese Funktion bislang unbekannt.

Nightly Tests

Der Test soll nachts einmal über die Debug-Version des onPC und einmal über die Linux-Version der Software laufen. Dafür gibt es ein Bash-Skript, welches den Test vorbereitet. Es kopiert das Showfile in das aktuelle Verzeichnis und legt die Verzeichnisse an um die entsprechenden Logfiles zu speichern.

Über ein Macro wird dann der Testdurchlauf gestartet. Wird die Zeichenfolge „All tests done“ im Feedback ausgegeben wird die Applikation geschlossen. Das Bash-Skript sieht den Versand von drei Emails vor, einmal ein Testlog, eine Test-terminated-Nachricht und einen Crashlog.

Seit ich mit dem Projekt begonnen hatte, war immer nur eine Email verschickt worden. Da ich nun intensiv an den Tests gearbeitet hatte, habe ich die Kollegen gebeten, den genauen Inhalt der Email vorerst zu ignorieren, da es ein Zwischenstand ist.

Nun hatten Arbeiten am betreffenden Server den Email-Versand unterbrochen. Seit das System wieder arbeitete sah man, dass das Logfile an einer Stelle mitten im Durchlauf endete.

Dies führen die Kollegen auf einen Absturz des Systems zurück, jedoch wurde kein Crashlog verschickt. Nach einigen Diskussionen stellt sich heraus, dass die Version des onPCs gar keinen Crashlog schickt und das zu sehende Logfile von der Linux-Version stammt.

Der Kollege, der für die Server zuständig ist, repariert den Versand des Crashlogs.

In diesem Crashlog wird nun die Code-Zeile des Fehlers angegeben.

Das System stürzt ab, wenn auf einen Vergleich von Fixtures zugegriffen wird, der in der Hauptsoftware mit dem Kommentar „please implement“ versehen ist.

Warum stürzt nun mein onPC-System nicht ab beim Ausführen des Tests?

Offensichtlich gibt es ein unterschiedliches Verhalten des Test-Systems unter Windows und unter Linux.

Implementieren des Vergleichs von Fixtures in der Hauptsoftware

Ich habe nun angefangen mich in die Hauptsoftware einzulesen. Bis ich den Vergleich implementieren kann, muss ich die selbstdefinierten Datentypen und den Methodenaufbau der Software in Erfahrung bringen.

Stand des Projekts bei Ende des Semesters

Der aktuelle Stand ist, dass mit ungefähr der Hälfte der Einzelskripte der Test valide und determiniert auf meinem onPC-System läuft. Der Teil der Einzelskripte enthält noch Befehlsfolgen, die viele Abkürzungen enthalten, sehr kompliziert angelegt sind und auch nur zum Teil eine sinnvolle Testabdeckung der Funktionen bilden. Auch enthalten sie noch eine hohe Anzahl an Fehlern beziehungsweise Möglichkeiten den Test wieder in nicht-determiniertes Verhalten zu überführen. Diese Tatsache erfordert noch etwas Bearbeitungszeit.

Auch steht noch das Inbetriebnehmen des „Golden Showfile“ an. Das bedeutet, dass das Testshowfile die Vergleichsobjekte schon enthält und die neu geladenen Testobjekte mit ihnen vergleicht ohne sie direkt vor dem Vergleich durch Kopieren anzulegen.

Fazit

In diesem Praxisprojekt habe ich die Ansätze der Bedienung der grandMA2 gelernt. Über Testautomatisierung diverse Bücher gelesen, mich in die Materie eingearbeitet und dieses Wissen angewendet. Überrascht war ich, dass ich relativ selten programmiert habe und das Refactoring des vorhandenen Codes hauptsächlich über Modellierung und Neustrukturierung zu bewerkstelligen war. Auch habe ich zum ersten Mal in einem Betrieb gearbeitet und dessen Strukturen und Beteiligte kennen gelernt. Besonders die Software-Meetings waren für mich aufschlussreich um das große Ganze der Firma und insbesondere die Softwareabteilung zu verstehen.

Ausblick

Ich werde meine Tätigkeit bei MA Lighting in fester Teilzeit weiterführen und gegebenenfalls auch meine Bachelorarbeit dort machen.

Als besonders wichtige Aufgabe habe ich das weitere Erlernen und Festigen meiner Programmierkenntnisse in C++ begriffen, was in meinem Studium leider etwas spärlich enthalten ist. Das Testen an sich macht mir Spaß und ich könnte mir auch vorstellen mich in diese Richtung weiterzuentwickeln.

Quellen:

Die verwendeten Fotos und Screenshots sind copyright MA Lighting Technology GmbH.

Alle grafischen Abbildungen sind von mir erstellt worden.