HOCHSCHULE
MITTWEIDA
UNIVERSITY OF
APPLIED SCIENCES

BACHELORARBEIT

Herr
Marco Felsch

Automatisiertes Testsystem für Linux BSPs

Beispiel TQMa6x

2015

Fakultät **Elektro- und Informationstechnik**

BACHELORARBEIT

Automatisiertes Testsystem für Linux BSPs

Beispiel TQMa6x

Autor:
Marco Felsch

Studiengang:
Embedded Systems

Seminargruppe:
EI11wS-B

Erstprüfer:
Prof. Dr. Thomas Beierlein

Zweitprüfer:
Dipl. Ing. (FH) Daniel Gericke

Mittweida, 2015

Bibliografische Angaben

Felsch, Marco: Automatisiertes Testsystem für Linux BSPs, Beispiel TQMa6x, 95 Seiten, 34 Abbildungen, Hochschule Mittweida, University of Applied Sciences, Fakultät Elektro- und Informationstechnik

Bachelorarbeit, 2015

Satz: LaTeX

Referat

I. Inhaltsverzeichnis

II. Abbildungsverzeichnis

III. Tabellenverzeichnis

IV. Vorwort

1 Einleitung

In der heutigen Zeit, in der Produkte immer komplexer und die Entwicklungszeiten immer kürzer werden, wird das Testen der Funktionalität immer schwieriger und umfangreicher. Eine schier zahllose Menge an Testfällen ist nötig um die korrekte Funktionsweise sicherzustellen. Damit dabei die Kosten des Produktes nicht ins Unermessliche steigen, ist eine Automatisierung des Testverfahrens unumgänglich.

In der Softwareentwicklung werden schon lange automatisierte Testdurchführungen eingesetzt. Bei kompletten Produkten/Geräten ist ein ähnlicher Trend festzustellen. Sie werden in eigens entwickelten Prüfständen automatisch und ununterbrochen geprüft, dies wird auch ”Hardware-in-the-Loop (HiL)” genannt. In beiden Fällen gibt es eine Vielzahl von Herstellern, die sich auf das Automatisieren der Testdurchführung spezialisiert haben.

Die TQ Group GmbH beschäftigt sich mit der Entwicklung kundenspezifischer Elektronik-Komponenten (Abbildung 1.1). Um den Hardware-Entwicklungsaufwand möglichst gering zu halten wird die kundenspezifische Hardware(HW) von der gleichbleibenden HW getrennt. Die gleichbleibende HW wird als ”Embedded Modul” vertrieben und umfasst das System on a Chip (SoC), den Flash- und Arbeitsspeicher sowie eine Real-Time-Clock (RTC), einen Temperatursensor und einen optionalen EEPROM-Speicher (Beispiele TQMa6x, [7]). Das ”Embedded Modul”, folgend Prozessor-Modul genannt, wird über eine Schnittstelle mit der entsprechenden kundenspezifischen Plattform verbunden. Plattformen welche das selbe Prozessor-Modul einsetzen verfügen demzufolge alle über ein identische Interface. Dieses Prinzip ist dem der ”haushaltsüblichen” PC’s ähnlich. So kann eine Intel-CPU auf verschiedenen Mainboards des gleichen Sockels verbaut werden. Dabei ist die CPU dem Prozessor-Modul, das Mainboard der kundenspezifischen Plattform und der Sockel der Schnittstelle gleich zu setzen. Zusätzlich zu der HW wird ein minimales Softwarepaket zur Verfügung gestellt, welches auch Board-Support-Package (BSP) genannt wird. Dieses enthält einen angepassten Bootloader,

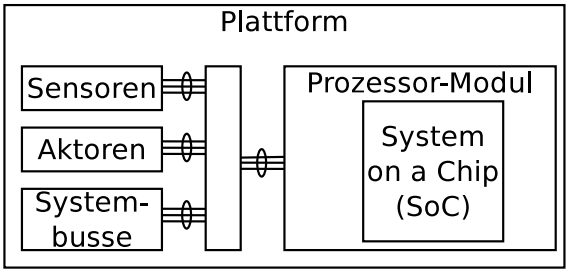


Abbildung 1.1: Schematischer Aufbau Elektronik-Komponenten

ein angepasstes Betriebssystem und einige vorinstallierte Programme zur Softwareentwicklung. Dadurch kann sich der Kunde auf die eigene Applikationsentwicklung konzentrieren.

Ziel dieser Arbeit ist der Entwurf eines Testsystems, dass die BSP's auf korrekte Funktionalität testet. Dies ist erfüllt wenn sich alle Schnittstellen der Plattform (GPIO, Audio, I2C, HDMI, usw.) wie gewünscht ansprechen lassen und verhalten. Es sollen keine Kernel-Funktionen wie die Inter-Prozess-Kommunikation (IPC), Echtzeitfähigkeit, usw. getestet werden. Die Testdurchführung soll weitestgehend automatisiert ablaufen und muss leicht anpassbar sein, da die TQ-Group eine Vielzahl von kundenspezifischen Plattformen entwickelt. Das Testsystem soll in die bereits vorhandene Build-Infrastruktur integriert werden, damit diese beibehalten werden kann.

Die Arbeit gliedert sich wie folgt:

1. Analyse des Ist- und des Soll-Zustandes
2. Kurzer theoretischer Einblick in die heute bekannten Teststrategien
3. Recherche über derzeitig verfügbare Testsysteme
4. Integration eines Testsystems in die bereits vorhandene Infrastruktur
5. Beispiel eines Testdurchlaufs
6. Zusammenfassung und Ausblick

2 Vorbetrachtungen / Theoretische Grundlagen - Teststrategien

Im folgenden Kapitel wird der Ist-Stand analysiert. Darauf aufbauend werden die Anforderungen an das neue Testsystem definiert. Weiterhin werden die benötigten theoretischen Grundlagen kurz dargestellt, die dem Leser als Basiswissen für diese Arbeit dienen sollen.

2.1 Begriffserklärungen

Die folgenden Begriffserläuterungen dienen dem Leser als Basishintergrundwissen für diese Arbeit.

Board Support Package (BSP)

Darunter ist ein Software-Paket zu verstehen, das es ermöglicht ein Betriebssystem auf einer bestimmten HW-Plattform einzusetzen (Abbildung 2.1, Seite 4). Es beinhaltet die folgenden Komponenten:

- Bootloader
  Dies ist das erste Programm welches nach einem ”System Power-On” ausgeführt wird. Er lädt den Kernel in den Arbeitsspeicher und führt ihn anschließend aus.
- Kernel
  Betriebssysteme können unterschiedlich aufgebaut sein, eine Möglichkeit ist die ”Kern-Schale-Architektur” (Abbildung 2.2, Seite 5) in der alle ”lebenswichtigen” und hardwareabhängigen Komponenten (z.B. Treiber) im Kern (kernel) enthalten sind [1]. Diese Komponenten besitzen uneingeschränkte Berechtigungen und können auf die reale HW zugreifen. Damit der HW-Zugriff korrekt erfolgt, muss der Kernel entsprechend angepasst werden. Wird beispielsweise der Linux-Kernel auf einem ARM-SoC eingesetzt, so erfolgt die Anpassung mithilfe des ”Device-Tree’s” (siehe unten). In der Schale (shell) werden die Benutzerprogramme sowie -dienste ausgeführt. Sie besitzen eingeschränkte Zugriffsrechte, welche ein direkten HW-Zugriff unterbinden.
- Root-Filesystem (rootfs)
  In ihm sind alle benötigten Programme und Bibliotheken enthalten, die vom Benutzer für die Applikationsentwicklung benötigt werden. In der ”Kern-Schale-Architektur” (siehe oben) befindet es sich in der Schale.

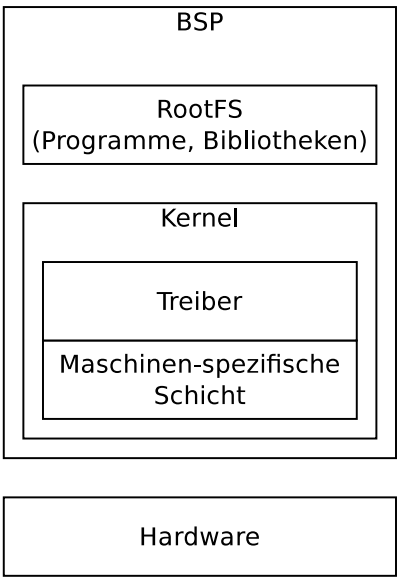


Abbildung 2.1: Aufbau eines BSP's

Device Tree

Ist eine Datenstruktur mithilfe derer die HW beschrieben wird. Dadurch wird es überflüssig die HW-Details fest in das Betriebssysteme zu schreiben (kodieren). Stattdessen wird die HW in einer Datenstruktur beschrieben, die dynamisch während des Bootvorgangs an den Kernel übergeben werden kann [8]. Somit wird eine Trennung des Betriebssystem-Kernels und der HW-Beschreibung hergestellt. Die Struktur wird als Textdatei (Device Tree Source, DTS) angelegt und muss durch einen Compiler in ein Binary (Device Tree Blob, DTB) übersetzt werden, das während des Bootvorgangs an den Kernel übergeben wird. Zum jetzigen Zeitpunkt bietet der Linux-Kernel einen Device Tree Support für sieben verschiedene Prozessor-Architekturen an [9].

Source Code Management (SCM)

Zur Verwaltung der Software-Quelldateien dient ein Versionskontrollsystem (VCS, Version Controlled System). Dieses "protokolliert Änderungen an einer Datei oder einer Anzahl von Dateien über die Zeit hinweg, so dass man zu jedem Zeitpunkt auf Versionen und Änderungen zugreifen kann" [10].

Versionskontrollsystem-Arten:

- Lokales Versionskontrollsystem
- Verteiltes Versionskontrollsystem
- Zentralisiertes Versionskontrollsystem

In der TQ Group GmbH wird ein verteiltes VCS namens "Git" [10] verwendet. Jede VCS-Art besitzt verschiedene Vor- und Nachteile, worauf allerdings nicht eingegangen werden soll, da dies nicht Bestandteil der Arbeit ist.

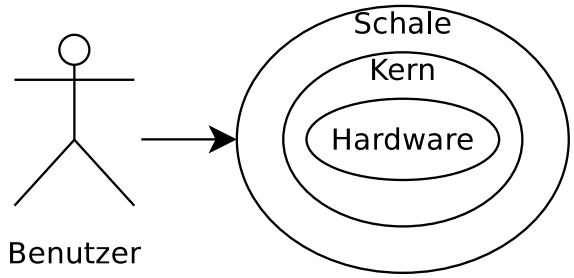


Abbildung 2.2: Betriebssystem Kern-Schale-Architektur, [1]

Build-System / Build-Server

Dies ist ein PC-Verbund[1] dessen einzige Aufgabe es ist, gegebene Quellcode-Dateien zu kompilieren. Um dies zu gewährleisten muss jedes Mitglied des Verbunds Zugriff auf die jeweiligen

- Compiler-Toolchains und
- Abhängigkeiten (z.B. bestimmte SW-Bibliotheken)

besitzen, die zur Übersetzung benötigt werden. Die Bearbeitung und Entwicklung der Quelldateien erfolgt auf separaten Entwicklungs-PC's. Vorteile dieser Trennung sind

- ein "Sandbox[2]"-artiger Kompiliervorgang,
- verringerte HW-Anforderungen an die Entwicklungs-PC's und
- ein 24/7 Kompilier-Service.

Durch die Kompilierung in einer "Sandbox" wird sichergestellt, dass die Build-Umgebung vor jedem Kompiliervorgang komplett bereinigt und anschließend vorbereitet wird. Dadurch können Abhängigkeiten einzelner Quellcode-Dateien schneller erkannt werden, da immer die gleiche Umgebung vorliegt. Für die Verwaltung der Build-Umgebungen (der "Sandboxes"), Build-Aufträge, Compiler-Toolchains und zusätzlicher SW (Bsp.: extra Bibliotheken) wird auf dem Build-System (der PC-Verbund) die Software "Jenkins" [11] eingesetzt. Der Kompilierprozess an sich soll nicht näher betrachtet werden. Es ist ausreichend zu wissen das es ein komplexer Prozess ist, in dem Compiler, Linker und Assembler zusammenwirken. Nach erfolgreichem Abschluss dieses Prozesses stehen ausführbare Dateien für die entsprechende Zielarchitektur zur Verfügung.

[1] Ein Verbund besteht aus mindestens einem PC

[2] Dies bezeichnet einen Bereich, der in sich abgeschlossen ist. Aktionen die innerhalb dieses Bereichs ausgeführt werden, haben keinen Einfluss auf die anderen Teile des Systems.

2.2 Bisheriges Testverfahren

2.2.1 Softwareentwicklungsprozess und Softwaretest

Startpunkt

Bevor mit der Entwicklung begonnen werden kann müssen die kundenspezifischen Anforderungen geklärt und festgehalten werden.

Beispiel-Anforderungen:

- Festlegung eines bestimmten Linux-Kernels
- Bestimmung des zu verwendenden Bootloaders und der Version
- Unterstützung bestimmter Treiber, die noch nicht im Linux-Kernel enthalten sind
- Bereitstellung bestimmter Programme und Bibliotheken, die der Kunde für die Entwicklung seiner eigenen SW benötigt

Einrichtung

Nachdem alle Anforderungen aufgestellt wurden, müssen einige Einrichtungsarbeiten erfolgen. Diese umfassen das Anlegen eines ”Git-Repository” auf dem Git-Server und das Erstellen eines ”Jenkins Build-Jobs”. Das ”Git-Repository” dient der zentralen Speicherung und Verwaltung der Quelldateien, der ”Jenkins-Job” zur Steuerung des Build-Ablaufes.

Build-Ablauf / Jenkins-Job:

1. Ein Build-Vorgang kann auf zwei verschiedene Arten gestartet werden:
   a) Er wird automatisch durch ein ”push” eines neuen ”Git-Commits” auf den Git-Server ausgelöst.
   b) Der Benutzer kann dem Build-Server (”Jenkins”) manuell mitteilt welche SW-Version er verwenden soll.
2. Vor Beginn der Kompilierung wird der gesamte Inhalt des Projektverzeichnisses gelöscht, damit eine aufgeräumte Build-Umgebung geschaffen wird.
3. Im Anschluss bezieht er alle nötigen SW-Quellen aus dem ”Git-Repository”, welche für die Kompilierung benötigt werden.
4. Nachfolgend werden die bezogenen Quelldateien kompiliert.
5. Nach Abschluss des Kompiliervorgangs werden alle SW-Entwickler durch eine Status E-Mail (Erfolg/Misserfolg) benachrichtigt.
6. Bei Erfolg werden alle erstellten Binaries auf einen internen zugänglichen Server geladen, wo sie von jedem TQ-Mitarbeiter heruntergeladen werden können.

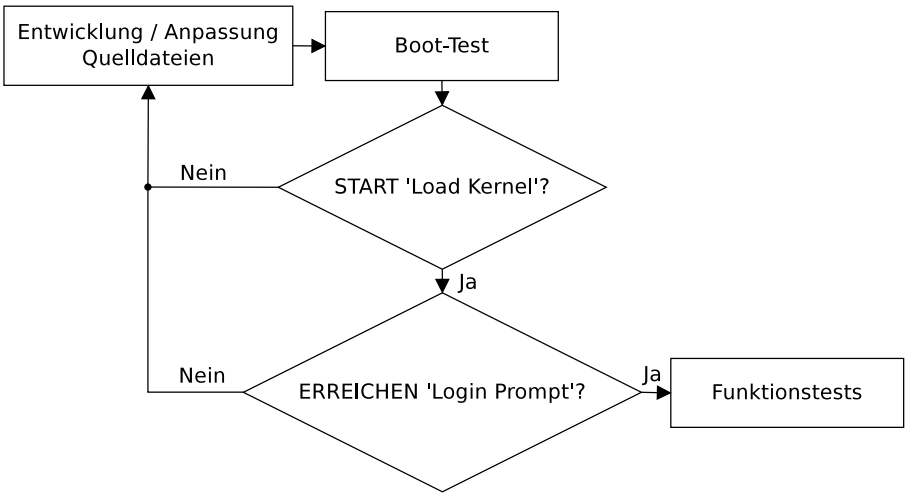


Abbildung 2.3: BSP Release Testablauf

Entwicklung

Nach Erledigung aller Vorausarbeiten kann mit der eigentlichen Entwicklung des kundenspezifischen BSP's begonnen werden. Die Bestandteile eines BSP's, wurden bereits in Abschnitt 2.1 besprochen.

Test

Im Anschluss an die Entwicklung werden die erstellten Binaries auf dem Zielsystem, auch "Device under Test" (DUT) genannt, getestet. Der Testablauf beginnt mit dem Test des Bootloaders:

- Wird die vom Bootloader unterstütze HW richtig erkannt?
- Wurde die Bootloader-Umgebung richtig eingestellt?
- Wird der Bootloader gestartet?
- Kann der Bootloader alle benötigten Images laden/beziehen?

Sobald der Linux-Kernel ausgeführt wird ist diese Testphase abgeschlossen.

Anschließend erfolgt der Startvorgang-Test des Linux-Kernels. In diesem wird überprüft ob alle HW-Komponenten richtig erkannt wurden. Weiterhin soll mit dem Test sichergestellt werden, dass der Kernel die "rootfs Ladephase" erreicht. Ist dies der Fall und das "rootfs" wurde geladen, kann mit dem Funktionstests der HW-Komponenten fortgefahren werden. Ein Beispiel eines solchen Tests ist in Anhang B.1 auf Seite 91 dargestellt. Der gesamte Testablauf ist in Abbildung 2.3 nochmals veranschaulicht.

Zur Verdeutlichung des gerade vorgestellten Softwareentwicklungsprozesses dient Abbildung 2.4 auf Seite 10.

2.2.2 Bewertung des Ist-Zustandes

Entwicklungsprozess

Der Gesamtprozess ist durch die Verwendung eines VCS und eines Build-Servers sehr gut aufgebaut. Das Zusammenspiel beider Systeme ermöglicht einen kontinuierlichen Entwicklungsprozess (Continues Integration, kurz: CI). Dadurch wird sichergestellt, dass alle SW-Versionen kompilierfähig sind/bleiben sowie, dass SW-Fehler rechtzeitig erkannt und schnell lokalisiert werden können.

Testverfahren

Das derzeit angewandte Testverfahren ist ein sogenanntes ad hoc[3]Verfahren. Dabei werden die Testdaten vom Tester manuell eingegeben und die Reaktion des Systems beobachtet sowie aufgezeichnet. Die Bewertung ob ein Test erfolgreich war oder fehlschlug erfolgt anhand des Systemverhaltens bzw. der Systemausgabe.

Vorteile

- Einfache Tests können ohne großen Mehraufwand durchgeführt werden.
- Die Testdurchführung ist für Menschen mit entsprechendem Hintergrundwissen intuitiv.

Nachteile

- Die Testbedingungen wurden nicht eindeutig definiert:
  - Ist die externe Testhardware immer angeschlossen, oder nur wenn die jeweilige Systemkomponente getestet wird?
  - Wird das System nach jedem Funktionstest einer Systemkomponente neu gestartet?
  - Wie soll vorgegangen werden wenn ein Funktionstest fehlschlägt? Soll der Testlauf fortgesetzt oder abgebrochen werden?
  - ...
- Nach jeder SW-Änderung muss theoretisch der gesamte Testdurchlauf neu ausgeführt werden, damit sichergestellt werden kann, dass die Änderungen keine negativen Auswirkungen auf den Rest des Systems haben. Dies ist sehr unwirtschaftlich und es kann nicht hundertprozentig sichergestellt werden, dass
  - eingegebene Testdaten immer die Gleichen sind,
  - der Testablauf immer in der gleichen Reihenfolge stattfindet und
  - Fehler bei der Ausgabe nicht übersehen werden.
- Testfälle sind nur in Textform vorhanden (Anhang B.1, Seite 91), dies birgt die Gefahren, dass
  - Testszenarien vergessen werden,

[3] lateinisch, eigentlich = zu diesem

  - Testfälle nur mit gewissem technischen Hintergrundwissen verstanden werden,
  - die Übersichtlichkeit verloren geht und
  - die Beschreibung der Testfälle unterschiedlich ist.
- Es gibt keine Angaben darüber, woraus die Testfälle (Anhang B.1, Seite 91) generiert werden.
- Die Testdurchführung (Anhang B.1, Seite 91) ist stark systemspezifisch. Können die gleichen Testfälle auch auf einem anderen Betriebssystem und/oder Bootloader ausgeführt werden?
- Die Testdokumentation muss während der Durchführung manuell erstellt werden.

2.3 Soll-Zustand der neuen Infrastruktur

Im Kapitel 1 wurde bereits kurz dargestellt welche neuen Eigenschaften das System besitzen soll. Folgend sollen diese näher betrachtet werden.

Kerneigenschaften

- Das neue Testsystem muss sich in die vorhandene Build-Infrastruktur (Abbildung 2.4, Seite 10) integrieren lassen.
- Es sollen nur Plattform-Schnittstellen und keine Kernel-Mechanismen (Inter-Prozess-Kommunikation (IPC), Echtzeitfähigkeit, usw.) getestet werden.
- Die Testfälle sollen so abstrakt wie möglich gehalten werden, damit eine einfache Adaption auf verschiedene Plattformen und Prozessor-Module möglich ist.
- Eine Testdurchlauf-Triggerung soll automatisch nach einem erfolgreichem Build-Vorgang erfolgen oder manuell unter Angabe des zu verwendenden DUT's und der Testfälle.
- Skripte die in den Testfällen aufgerufen werden müssen unabhängig vom Testsystem sein, damit sie auch außerhalb des Testsystems verwendet werden können. Dies kann z.B. für einen Schnelltest nötig sein. Deshalb dürfen in den Skripten keine Befehle, Pfade und Variablen benutzt werden, die ausschließlich von dem Testsystem bereitgestellt werden.

Zusatzeigenschaften

- Die Erstellung der Testdokumentation soll automatisch, nach Abschluss der Testdurchführung erfolgen.
- Testfälle sollen so generisch gehalten werden, dass sie komplett unabhängig von der verwendeten Soft- und Hardware des Targets sind.

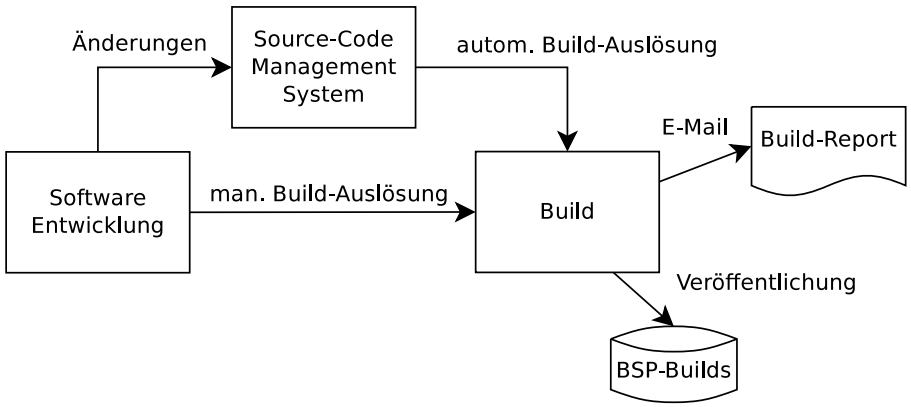


Abbildung 2.4: Softwareentwicklungszyklus bisher

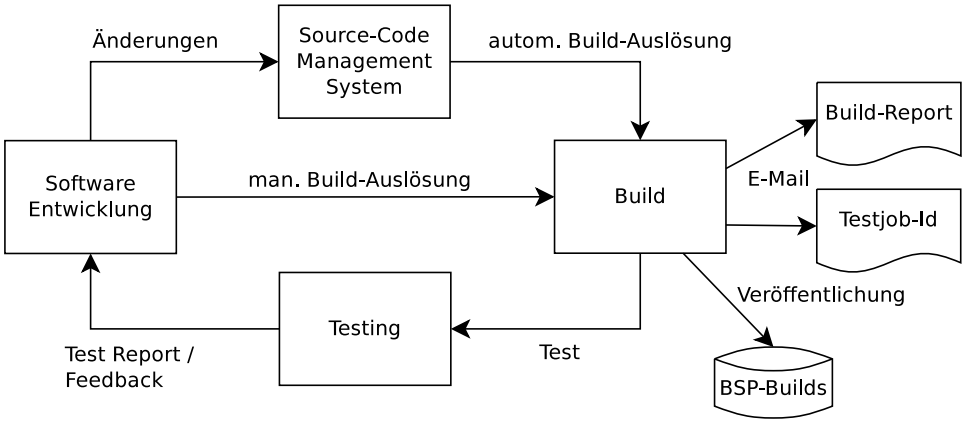


Abbildung 2.5: Softwareentwicklungszyklus danach

Das Testsystem soll sich wie in Abbildung 2.5 dargestellt in die bisherige Infrastruktur eingliedern. Für eine ausführlichere Beschreibungen der Anforderungen dienen die Tabellen 2.1 und 2.2 auf Seite 11 und 12.

2.4 Teststrategien in der SW-Entwicklung

Der folgende Abschnitt beschäftigt sich mit der theoretischen Betrachtung vorhandener SW-Teststrategien und ist eine Zusammenfassung von [2]. m Laufe der Zeit haben sich sehr viele SW-Prüftechniken entwickelt. [2] unterteilt diese nach Abbildung 2.6 auf Seite 14. Wie zu erkennen ist, unterteilt [2] die Prüftechniken in zwei große Gruppen.

Statische Prüftechniken

Bei der Verwendung statischer Techniken liegt das Hauptaugenmerk auf der Analyse eines gegebenen Programmcodes. Ein sehr bekannter statischer Test ist die Stilanalyse. Bei dieser wird ein gegebener Quellcode auf die Einhaltung eines sogenannten Coding-Styles geprüft, welcher vorher festgelegt wird und strikt eingehalten werden muss. Eine weitere Betrachtung der statischen Techniken soll

Art*	Nummer	Kurzbeschreibung	Beschreibung	Zusatz
R-TF	1	Betriebssystem	Es kommt eine auf Linux-basierende Distribution zum Einsatz.	
	2	Bootloader	Das DUT hat Zugriff auf einen lokalen Bootloader.	Bootloader-Quellen: SD-Karte SPI-Flash Speicher
	3	Unterstützung	Die auf dem DUT eingesetzten Hardwarekomponenten müssen vom verwendeten Linux-Kernel unterstützt werden.	
N-TF	1.1	Wiederverwendbarkeit	Die Testfälle müssen auf verschiedenen HW-Plattformen einsetzbar sein.	Bei sehr komplexen Testfällen, kann eine gewisse HW-Abhängigkeit entstehen
	1.2		Geringer Aufwand für die Adaption sehr komplexer Testfälle.	
	1.3		Die Testfälle müssen in komplexen Testabläufen und Einzeltests ausführbar sein.	
O-TF	1	Unabhängigkeit	Testfälle sind vollkommen unabhängig vom Betriebssystem und der HW zu implementieren	HW: Mikrocontroller, oder Mikroprozessor

Tabelle 2.1: Testfall Anforderungen

* Abkürzungen: TF – Testfall; R – Randbedingung; N – Notwendige Anforderungen; O – Optionale Anforderungen

Art**	Nummer	Kurzbe-schreibung	Beschreibung	Zusatz
R-TS	1	Eingabe	Das Testsystem erwartet ein erfolgreich erstelltes BSP	BSP enthält: Linux-Kernel Image Device-Tree Blob Root-Filesystem
N-TS	1.1	Eingabe	Es muss die Möglichkeit bestehen Testjobs automatisch auszulösen.	Dies ist durch einen neuen Jenkins-Job zu implementieren.
	1.2		Es muss die Möglichkeit bestehen Testjobs manuell auszulösen	Auslösung erfolgt durch den Benutzer über eine Benutzerschnittstelle
	1.3		Die Testbedingungen dürfen sich nie ändern, damit ein Regressionstest gewährleistet werden kann	Ausnahmen: Anpassung des Testjobs Anpassung des Testfalls Neuimplementierung
	2.1	Ausgabe	Dem Benutzer muss signalisiert werden wenn ein Testjob beendet ist	Signalisierungsarten: per Email Webfrontend
	2.2		Darstellung der Testjobergebnisse müssen übersichtlich sein. Dies ist durch einen einfachen ERFOLGREICH/NICHT-ERFOLGREICH Status zu implementieren	
	2.3		Der Benutzer muss über eine LOG-Datei genaue Details des Testjobs auslesen können	
	2.4		Der Benutzer muss bei einem Testjobfehlschlag durch eine Email benachrichtigt werden	
O-TS	1.1	Ausgabe	Es sollte die Möglichkeit bestehen Testjobergebnisse automatisch in einem Dokument zu sammeln und darzustellen.	Die Form des Dokumentes ist vom Benutzer bestimmbar.

Tabelle 2.2: Testsystem Anforderungen

** Abkürzungen: TS – Testsystem; R – Randbedingung; N – Notwendige Anforderungen; O – Optionale Anforderungen

nicht erfolgen, da sie nicht geeignet sind, um die Anforderungen aus Punkt 2.3 zu erfüllen. Für weitere Informationen sei auf das sehr ausführliche Buch [2] verwiesen.

Dynamische Prüftechniken

Alle dynamischen Prüftechniken benötigen zum Testen das ausführbare Programm, welches mit ausgewählten Eingabewerten getestet wird. Der Vorteil dynamischer Tests ist, dass sie in realen Betriebsumgebungen getestet werden können. Der Hauptnachteil dieser Tests ist, dass eine Aussage über die korrekte/unkorrekte Funktionalität einer SW nur für die getesteten Eingabewerte getroffen werden kann. Dies erfordert eine geeignete Auswahl der Eingabewerte, damit eine möglichst hohe Testabdeckung erreicht und eine Testdoppelung vermieden wird.

> ”Die dynamischen Testtechniken liefern leider aufgrund ihres Stichprobencharakters nur unvollständige Aussagen über die getestete Software. Dijkstra hat diesen Sachverhalt in der Aussage zusammengefasst, dass Testen die Anwesenheit von Fehlern demonstrieren könne, nicht jedoch deren Abwesenheit.” [2, S. 39]

Es ist weit verbreitet dynamische Testtechniken in ”Black Box” und ”White Box” Techniken zu unterteilen.

”White Box”-Techniken

Bei dieser Art wird die Struktur des Programmcodes verwendet. Zu ihnen zählen die strukturorientierten Tests, welche die Vollständigkeit des Tests anhand der Abdeckung des SW-Quellcodes beurteilen. ”Es können z.B. Anweisungen, Zweige oder Datenzugriffe verwendet werden.” [2, S.40] Die Bewertung der Testergebnisse erfolgt anhand der zuvor festgelegten Spezifikation.

”Black Box”-Techniken

Im Gegensatz zu den ”White Box” Techniken, wird die Struktur des Programmcodes nicht verwendet. Bei diesen Techniken wird die Testvollständigkeit und die Bewertung der Testergebnisse anhand der zuvor festgelegten Spezifikation beurteilt. Alle funktionsorientierten Testtechniken zählen zu der Gruppe der ”Black Box” Techniken.

Das Prinzip beider Techniken ist nochmals in den Abbildungen 2.7 und 2.8 auf Seite 15 dargestellt.

Eine Vorstellung jeder einzelnen Prüf-/Testtechnik aus Abbildung 2.6 würde den Rahmen dieser Arbeit übersteigen. Deshalb sollen nur die für diese Thema relevanten Techniken vorgestellt werden.

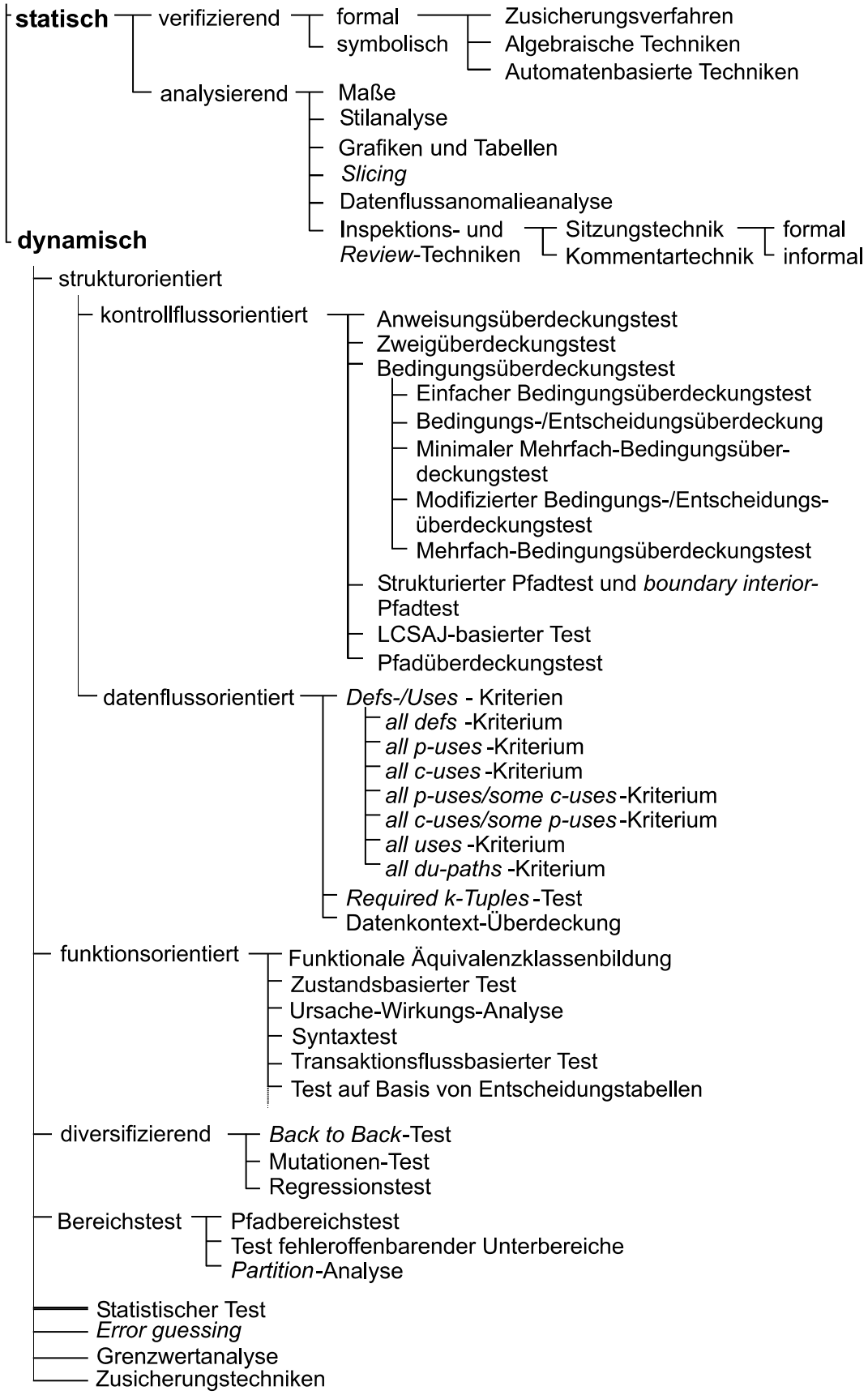


Abbildung 2.6: Klassifikation der Software-Prüftechnik, [2, S. 41]

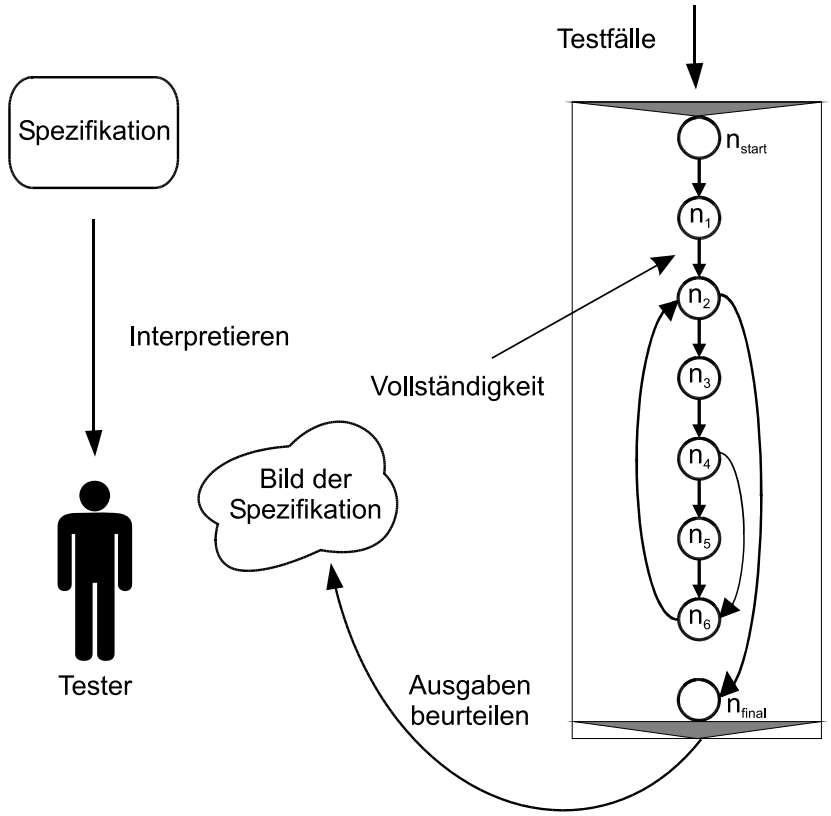


Abbildung 2.7: Prinzip des strukturorientierten Tests [2, S. 41]

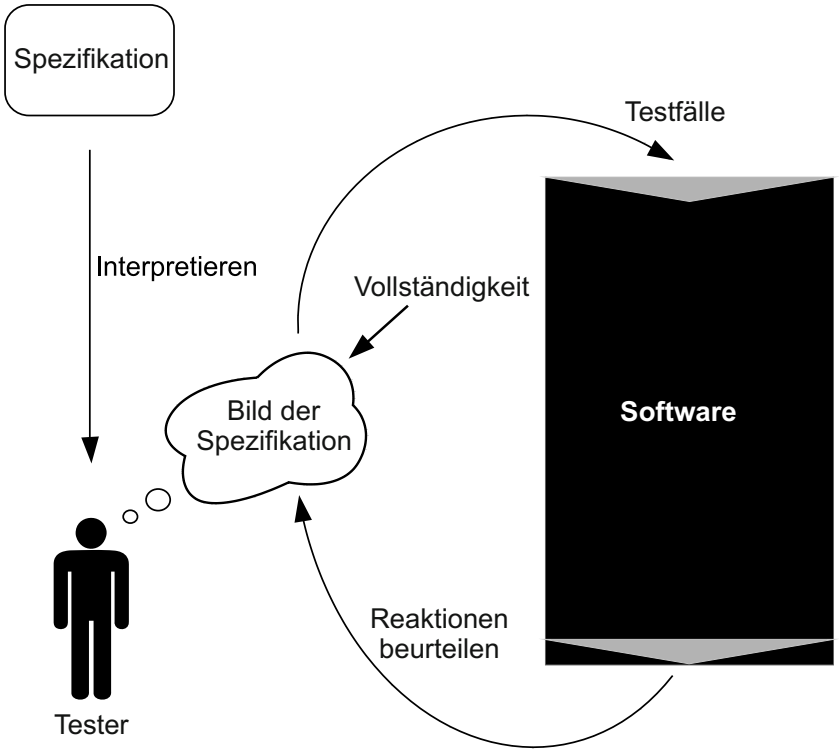


Abbildung 2.8: Prinzip des funktionsorientierten Tests [2, S. 41]

Abbildung 2.9: Hierarchie der Vollständigkeitskriterien [2, S.61]

2.4.1 Zustandsbasierter Test

Eigenschaften

Mit zustandsbasierten Tests werden gedächtnisbehaftete Systeme getestet. Diese Systemart arbeitet zustandsorientiert, ähnlich wie Automaten in der Digitaltechnik. Das Verhalten dieser lässt sich nach Formel 2.1 beschreiben, wobei **Z[n]** der Zustand zum Zeitpunkt **n** und **S** die Übergangsbedingung ist.

$$\begin{aligned} Z[0] &= initial \\ Z[1] &= Z[0] \wedge S \\ Z[2] &= Z[1] \wedge S \\ &\dots \\ Z[n] &= Z[n-1] \wedge S \end{aligned} \tag{2.1}$$

Gedächtnisbehaftete Systeme lassen sich meist als zweidimensionale Gebilde darstellen, da in einem betrachteten Zustand oft mehrere Ereignisse auftreten können, die unterschiedliche Aktionen und Folgezustände bewirken. Darum ist es nicht zweckmäßig die Spezifikation/Beschreibung solcher Systeme in Textform zu erstellen, womit sich Abläufe nur sequentiell darstellen lassen. Sinnvoller ist es hingegen die Beschreibung als Zustandsautomaten zu verfassen. Dies dient der Übersichtlichkeit und der Vermeidung einer unvollständigen Systembeschreibung. Weiterhin kann er als Basis zur Erzeugung von Testfällen dienen. Die Erklärung des Zustandsautomaten erfolgt im Absatz ”Darstellung”.

Testvollständigkeitskriterien

Die Vollständigkeit kann durch drei Kriterien beschrieben werden, welche hierarchisch aufgebaut sind (Abbildung 2.9):

1. In den Testfällen wird mindestens jeder Zustand des Automaten einmal Durchlaufen.
2. Jeder Zustandsübergang des Automaten wird mindestens einmal Durchlaufen.
3. Jedes auftretende Ereignis (event), dass einen Zustandsübergang hervorruft wird mindestens einmal Getestet.

Die Auswahl des Kriteriums, ist von dem zu testenden System abhängig, nicht immer ist das umfangreichste Kriterium auch das sinnvollste.

Darstellung

Als Darstellungsform sind

- der Zustandsautomat und
- die Zustandsübergangstabelle

möglich. Der Zustandsautomat (Abbildung 2.10, Seite 19) kann eins zu eins in eine Zustandsübergangstabelle (Abbildung 2.11, Seite 19) transferiert werden und umgekehrt. Zur Beschreibung der Zustandsautomaten wird üblicherweise die ”Unified Modeling Language” (UML) benutzt. Die Übergänge werden mithilfe gerichteter Kanten dargestellt. Das auslösende Ereignis eines Übergangs wird vor dem Schrägstrich notiert, die mit dem Übergang verknüpfte Aktion nach dem Schrägstrich. Die Kante des Initialzustandes wird mit einem Punkt dargestellt (Abbildung 2.10, Seite 19: Die Kante, die von rechts auf ”Getrennt” zeigt). Zur Verbesserung der Verständlichkeit, können den einzelnen Zuständen Bemerkungen hinzugefügt werden (Abbildung 2.10, Seite 19).

Es ist wichtig darauf zu achten, dass bei der Erstellung des Zustandsautomaten bzw. der Zustandsübergangstabelle alle Übergänge betrachtet werden. Dazu ist es meist sinnvoll den Automaten in eine Übergangstabelle zu übertragen. In dieser ist schnell ersichtlich welche Übergänge noch nicht betrachtet worden. Die mithilfe des Automaten bzw. der Tabelle erzeugten Testfälle, können oft nicht immer komplett getestet werden, deshalb unterscheidet man im Regelfall zwischen den folgenden drei Testsituationen:

1. Ereignisse, die einen Zustandsübergang und/oder eine Aktion auslösen.
2. Ereignisse, die auftreten können, aber ignoriert werden.
3. Ereignisse, die auftreten können und zu einem Fehlverhalten führen.

Bewertung

Vorteile:

- Der Zustandsautomat wird in vielen technischen Anwendungen verwendet, z.B. in der Automatisierungstechnik. [2, S. 64]
- Durch seine zweidimensionale Beschreibung können komplexe Sachverhalte besser dargestellt werden als in Textform. [2, S. 59-60]
- Bei der Darstellungsform wird den Test-Entwicklern Freiraum gelassen, da diese nicht genau spezifiziert ist.

Nachteile:

- Der Zustandsautomat kann bei sehr komplexen Sachverhalten schnell unübersichtlich werden.
- Ab einer gewissen Zustandsanzahl, können nicht immer alle Testvollständigkeitskriterien erfüllt werden.
- Durch die freie Darstellungsmöglichkeit wird eine Portierung der Testfälle innerhalb verschiedener Entwicklungs-Teams erschwert.

2.4.2 Transaktionsflussbasiertes Testen

Eigenschaften

Diese Testart kann als ein zustandsbasierter Test mit Zeitbasis angesehen werden. Transaktionsflussbasierte Tests werden zur Darstellung des System- bzw. SW-Verhalten verwendet, beispielsweise zur Beschreibung von Datenübertragungs-Protokollen.

> "... eine Transaktion ist ein Verarbeitungsmodul aus der Sicht eines Systembenutzers. Transaktionen bestehen aus einer Sequenz von Verarbeitungsschritten." [2, S. 75]

Darstellung

Wie bei den zustandsbasierten Tests ist keine einheitliche Darstellungsform vorgegeben. In der Protokollentwicklung sind Sequenzdiagramme (Abbildung 2.12, Seite 20 und Abbildung 2.13, Seite 20) weit verbreitet, in der Automatisierungstechnik wiederum die Flussdiagramme. Vorteil der Sequenzdiagramme ist, dass sie als Unterkategorie Bestandteil der UML sind. Dies erleichtert die Testfallerstellung, wenn das System bzw. die Software bereits als UML-Modell vorliegt.

Bewertung

Vorteile:

- Durch die Sequenz- bzw. Flussdiagramme werden mögliche Testfälle direkt angegeben.
- Gut geeignet für notwendige zu leistende Tests, welche ausgeführt werden müssen.

Nachteile:

- Sie besitzen nur exemplarischen Charakter.
- Es ist keine einheitliche Notation festgelegt.

Somit "... ist ein transaktionsflussbasierter Ansatz als notwendig, nicht aber als hinreichend zu bewerten." [2, S. 77]

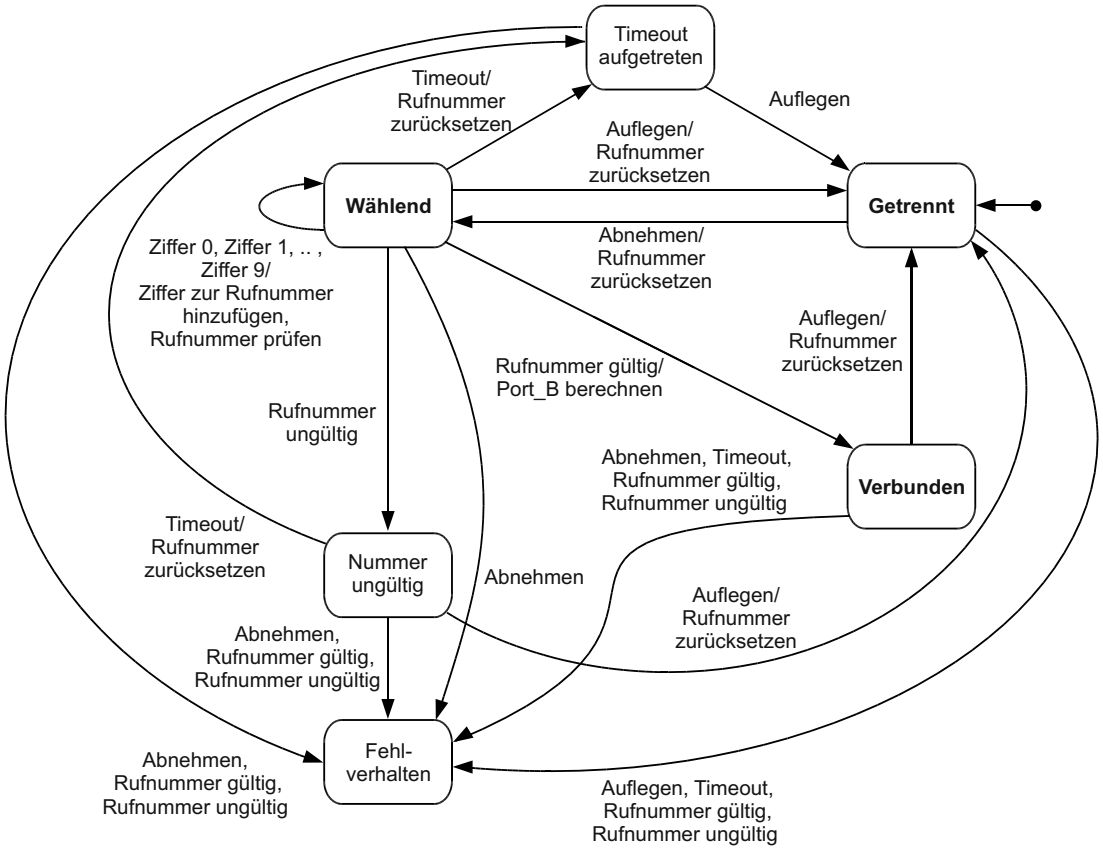


Abbildung 2.10: Vollständiger Zustandsautomat am Beispiel Telefon [2, S. 65]

Zustand / Ereignis	Getrennt	Wählend	Verbunden	Nummer ungültig	Timeout aufgetreten
Abnehmen	Wählend / Rufnr. zurücksetzen	Fehlverhalten / -	Fehlverhalten / -	Fehlverhalten / -	Fehlverhalten / -
Auflegen	Fehlverhalten / -	Getrennt / Rufnr. zurücksetzen	Getrennt / Rufnr. zurücksetzen, Verbind. abbauen	Getrennt / Rufnr. zurücksetzen	Getrennt / -
Ziffer 0	Getrennt / -	Wählend / Ziffer z. Rufnr. Hinzuf., Rufnr. prüfen	Verbunden / -	Nummer ungültig / -	Timeout aufgetreten / -
Ziffer 9	Getrennt / -	Wählend / Ziffer z. Rufnr. hinzuf., Rufnr. prüfen	Verbunden / -	Nummer ungültig / -	Timeout aufgetreten / -
Timeout	Fehlverhalten / -	Timeout aufgetreten / Rufnr. zurücksetzen	Fehlverhalten / -	Timeout aufgetreten / Rufnr. zurücksetzen	Timeout aufgetreten / -
Rufnummer gültig	Fehlverhalten / -	Verbunden / Verbindung aufbauen	Fehlverhalten / -	Fehlverhalten / -	Fehlverhalten / -
Rufnummer ungültig	Fehlverhalten / -	Nummer ungültig / -	Fehlverhalten / -	Fehlverhalten / -	Fehlverhalten / -

Abbildung 2.11: Vollständige Zustandsübergangstabelle am Beispiel Telefon [2, S. 65]

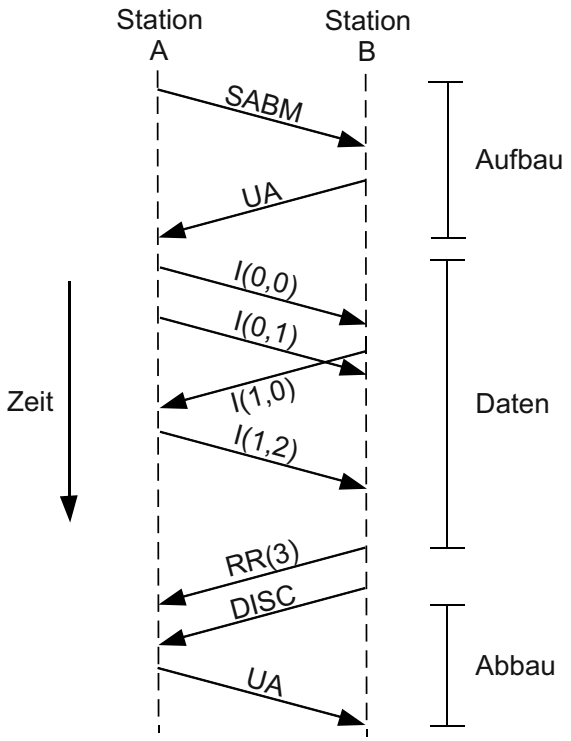


Abbildung 2.12: Sequenzdiagramm eines HDLC-Protokolls [2, S. 78]

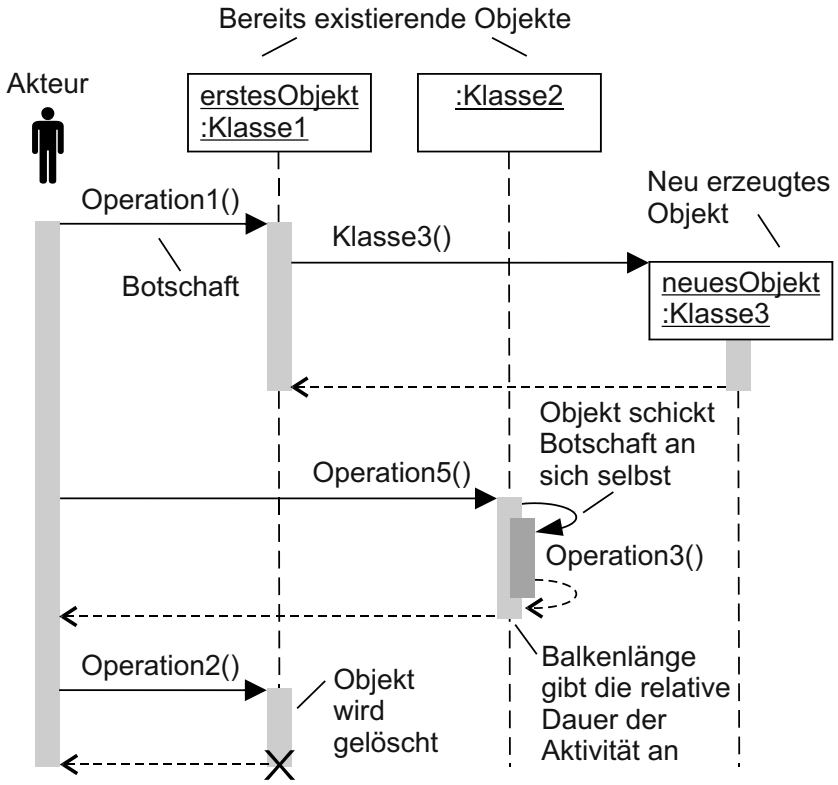


Abbildung 2.13: Squenzdiagramm in UML mit drei Instanzen [2, S. 78]

2.4.3 Regressionstest

Eigenschaften

Ziel ist es nachzuweisen, dass durch eine Modifikation eines vorhandenen SW-Standes keine Fehler bzw. Seiteneffekte auftreten. Dies wird durch die Wiederholung aller Testdurchläufe erreicht [2, S. 192]. Es wird ausschließlich auf Folgefehler getestet, da davon ausgegangen wird, dass die Testfälle alle schon einmal erfolgreich absolviert wurden. Somit ist die Anzahl der auftretenden Fehler typischerweise deutlich geringer, als bei der erstmaligen Durchführung der Testfälle.

Voraussetzungen

- Es ist eine SW mit zwei verschiedenen Versionen gegeben, der Version 1.0 und Version 1.1, wobei die Version 1.1 eine Weiterentwicklung der Version 1.0 darstellt.
- Die Testbedingung dürfen sich für die SW der Version 1.0 und Version 1.1 nicht ändern. Dies können zum Beispiel
  - Umweltbedingungen (z.B. Luftfeuchtigkeit),
  - SW-Bedingungen (z.B. Umgebungsvariablen),
  - HW-Bedingungen (z.B. Peripherie) und
  - die Reihenfolge der Aufrufparameter

  sein, welche immer gleich bleiben müssen. Da sonst der Regressionstest an Aussagekraft verliert.
- Die Softwareversion 1.0 hat alle Testdurchläufe erfolgreich bestanden.

Durchführung

1. Alle Testdurchläufe der Softwareversion 1.0 werden 1 zu 1 für die Softwareversion 1.1 übernommen und ausgeführt.
2. Im Anschluss daran werden die Testergebnisse miteinander verglichen.
   - Ist kein Unterschied zwischen den Ergebnissen der einzelnen Versionen festzustellen, war der Test erfolgreich.
   - Unterscheiden sich die Ergebnisse der einzelnen Softwareversionen, muss überprüft werden ob der Unterschied ein Fehler ist, oder eine neuen Funktionalität der Version 1.1 darstellt. Handelt es sich um eine neue Funktionalität, muss bei Erfolg des Tests, das Testergebnis, als neue Testbasis für die folgenden Tests verwendet werden.

In Abbildung 2.14 ist dieser Zusammenhang noch einmal dargestellt.

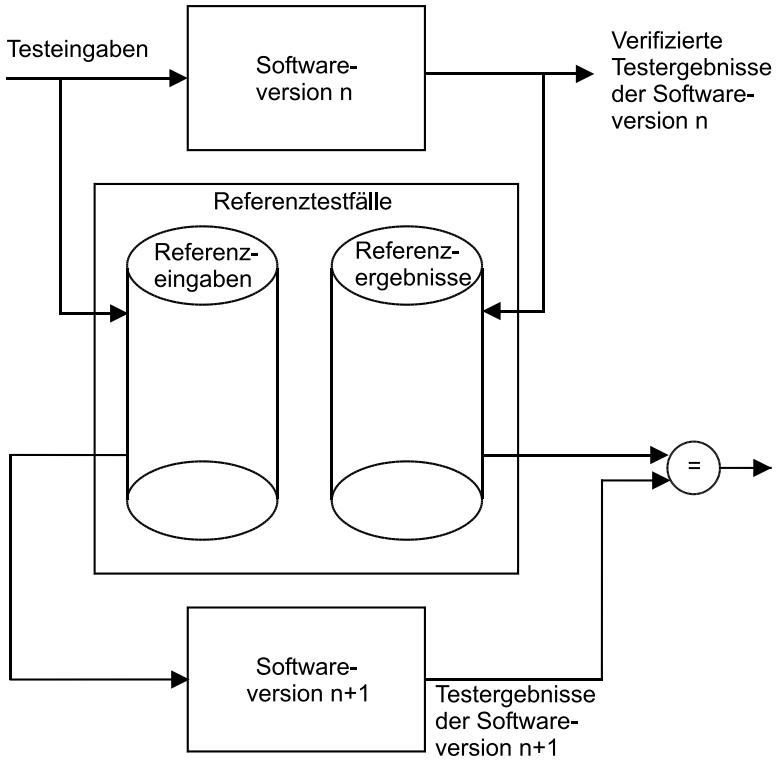


Abbildung 2.14: Prinzip des Regressionstests [2, S. 194]

Bewertung

Regressionstests können sowohl manuell als auch automatisch durchgeführt werden, wobei von der manuellen Methode abzuraten ist.

> ”Die Kombination aus der geringen Wahrscheinlichkeit, dass ein Fehlverhalten auftritt, und der hohen Wahrscheinlichkeit, dass dieses Fehlverhalten übersehen wird, führt dazu, dass manuelle Regressionstests in der Regel wirtschaftlicher Unsinn sind.” [2, S. 193]

Vorteile:

- Regressionstests sind Programmiersprachenunabhängig, da nur die I/O-Schnittstellen des Systems bzw. der SW betrachtet werden.
- Sehr gut automatisierbar:
  - Zeitersparnis für Tester
  - Anzahl der Testdurchläufe beliebig
  - Kostenersparnis über längeren Zeitraum
- SW-Fehlern und ungewollten Nebeneffekten werden schnell erkannt, da die Fehler/Nebeneffekte nur in der ”neuen” Version auftreten.

Nachteile:

- Bei manueller Testdurchführung ist es kaum möglich ein Test-Fehlverhalten festzustellen, da ein Herstellen gleicher Testbedingungen sehr schwer ist.
- Die manuelle Durchführung ist wirtschaftlich kaum bzw. nicht tragbar.
- Ein erfolgreicher Testfall, der allerdings semantisch nicht korrekt ist (vom Tester falsch umgesetzter Ersttest), bleibt unbemerkt.

2.4.4 Zusammenfassung

Wie eingangs erwähnt, gibt es wesentlich mehr Testtechniken, als die Vorgestellten. Diese sollen aber genügen, da sie die Anforderungen der Aufgabenstellung (siehe Kapitel 2.3) ausreichend erfüllen.

- Die Plattform-Schnittstellen können auf ihre Funktionalität überprüft werden (Validation).
- Mit Hilfe der Zustandsübergangstabelle, des Zustandsautomaten, des Sequenzdiagramms und des Flussdiagramms werden dem Tester nützliche Werkzeuge zur Testerstellung bereitgestellt. Diese geben den Testern und anderen Entwicklern einen schnellen Überblick über die Testfälle, ohne dabei die Testimplementierung kennen zu müssen.
- Unter Verwendung der UML können Testfälle generisch[4] gestaltet werden.

Der Testentwurfs- und Testablaufprozess kann unter Verwendung der vorgestellten Werkzeuge neu gestaltet werden.

Testentwurfs- und Testablaufprozess:

1. Es muss festgelegt werden welche Systemkomponente getestet werden soll.
2. Je nach Systemkomponente und Testumfang muss ein
   - Transaktionsflussbasierter Test, oder
   - Zustandsbasierter Test

   ausgewählt werden.
3. Nach der Wahl der Testart muss entschieden werden welche Beschreibungssprache genutzt werden soll. Hierfür steht
   - das UML-Sequenzdiagramm,
   - der UML-Zustandsautomat,
   - die Zustandsübergangstabelle, oder
   - das Flussdiagramm/Ablaufdiagramm

   zur Auswahl. Es besteht auch die Möglichkeit mehrere Beschreibungssprachen für einen Testfall zu benutzen.

[4] Die Testfälle sind vom Zielsystem Hard- und Softwareunabhängig

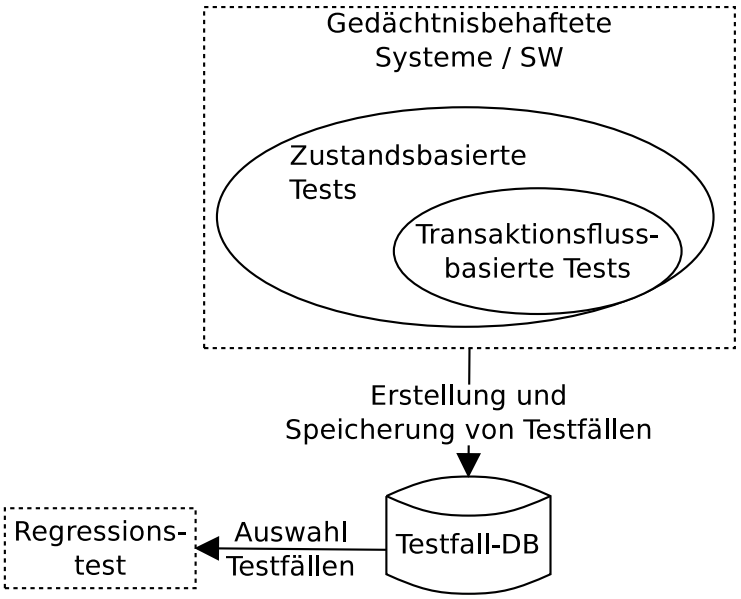


Abbildung 2.15: Zusammenhang der vorgestellten Testverfahren

4. Wurde ein geeignetes Beschreibungsmittel festgelegt, kann mit der Erarbeitung der Testlogik (Semantik) begonnen werden.
5. Im Anschluss daran erfolgt auf Basis der erstellten Testlogik (siehe Punkt 4), die Testimplementierung.
6. Der neu implementierte oder abgeänderte Testfall wird ausgeführt und die Testergebnisse werden kontrolliert. Arbeitet die Testlogik wie erwartet (ist die der Test semantisch korrekt) kann fortgefahren werden, anderenfalls muss zu Punkt 5 zurückgegangen werden. Im Extremfall muss bis zu Punkt 3 zurückgegangen werden.
7. Anschließend wird der Testfall an einem globalen Ort gespeichert, damit er für Regressionstests zur Verfügung steht. Hierfür bietet sich eine Datenbank (DB) oder ein Repository an, das von einem VCS verwaltet wird.
8. Die anschließenden Tests erfolgen mit Hilfe des Regression-Testverfahrens.

Der gesamte Ablauf ist nochmals in Abbildung 2.15 dargestellt.

3 Analyse vorhandener Testsysteme

Im folgenden Kapitel soll ein Überblick[5]über die Testsysteme gegeben werden, die sich während der Recherche als besonders geeignet herausgestellt haben, um die gestellten Forderungen zu erfüllen. Anschließend erfolgt, unter Beachtung der Anforderungen, eine Gegenüberstellung der Testsysteme. Am Ende des Kapitels wird, basierend auf dem Vergleich der Testsysteme, die Entscheidung für ein System getroffen.

3.1 LTP / LTP-DDT

Das Linux Test Project [12] (LTP) wurde mit dem Ziel ins Leben gerufen, dass der Open Source Community eine Testsuite zur Verfügung gestellt wird, die es ihr ermöglicht ihre Software (Linux-Kernel und Systembibliotheken) auf Sicherheit, Robustheit und Stabilität zu validieren. Das Projekt wird von Großunternehmen wie IBM, Cisco, SUSE sowie Red Hat genutzt und gewartet. Somit kann davon ausgegangen werden, dass es ein zuverlässiges Tool ist, welches regelmäßige Updates erhält.

Auf Basis des LTP-Projekts hat Texas Instruments eine Testsuite [13] entwickelt, die sich auf das Testen der Gerätetreiber eingebetteter Systeme fokussiert. Dafür wurde dem LTP-Projekt eine Vielzahl an Testfällen hinzugefügt, welche die Plattform-Schnittstellen (z.B. GPIO, I2C-Bus) testen. Damit sich die beiden Projekte namentlich unterscheiden, wurde das neue Projekt LTP-Device Driver Tests (LTP-DDT) genannt.

Im folgendem sollen die beiden Projekte gemeinsam betrachtet werden, da sie auf der gleichen Infrastruktur basieren.

3.1.1 Übersicht

Systemaufbau

- Testfall
  Er stellt die Basiseinheit des LTP-Projekts dar und beinhaltet eine Einzelaktion die getestet werden soll sowie deren Funktionsüberprüfung. Einzelaktionen können Plattform-Schnittstellentests oder Kernel-Funktionstests umfassen.

[5] Die Überblicke basieren auf den jeweiligen Projekt-Dokumentationen [3] [4] [5] [12] [13] und sollen dem Leser eine kompakte Zusammenfassung geben. Für ausführliche Informationen sei auf die jeweiligen Projektseiten verwiesen.

- Testprogramm
  Dies ist ein ausführbares Programm oder Shell-Skript, welches ein oder mehrere Testfälle enthält. Das Verhalten des Programms kann durch Übergabeparameter gesteuert werden.
- Test-Markierung (Tag)
  Verbindet eine eineindeutigen Markierung mit einem Testprogramm und den Übergabeparametern, mit denen es aufgerufen wird.

Ausgabeformat der Testfallergebnisse

- ”Exit-Style” Tests
  Führt das Testprogramm zu einem unerwartetem oder fehlerhaftem Ergebnis, wird es mit einem Exit-Status ungleich null beendet, z.B. ”exit(1)”. Verhält sich das Testprogramm wie erwartet wird es mit Null beendet, ”exit(0)”.
- ”Formated Output” Tests
  Das Ergebnis jedes Testfalls wird auf die Standardausgabe (STDOUT) geschrieben. Wurde das Ausgabeformat vorher festgelegt, ist es möglich das Ergebnis anschließend mit weiteren Tools zu analysieren.

Wie bereits erwähnt werden die Testprogramme als Shell-Skripte oder als C-Programme verfasst. Dadurch wird dem Test-Entwickler eine gewisse Implementierungs-Freiheit geboten. Diese wird jedoch durch den Coding-Style beschränkt. In ihm wurden Regeln aufgestellt, an die sich die Entwickler bei der Implementierung der Skripte und Quelldateien halten müssen, damit sich das Aussehen der Implementierten Testfälle möglichst wenig unterscheidet (die Code-Konsistenz gewahrt wird). Beispielsweise legt er LTP-spezifische Makros fest, welche die Entwickler verwenden müssen, da sie von den mitgelieferten LTP-Tools verwendet werden. Eines dieser Programme (LTP-Pan) dient der Ausführung der Tests, das andere (LTP-Scanner) der Analyse der Testergebnisse. Das LTP-DDT Projekt verwendet den gleichen Coding-Style und fügt diesem noch eine Richtlinie für die Namensgebung der Skripte und ausführbaren C-Programme hinzu.

LTP-Pan

Aufgaben:

- Es führt die Testprogramme aus.
- Es erfasst die Ausgabe der Testfälle.
- Das Programm stellt den Fortschritt des Testdurchlaufs in einer Statusdatei dar.

Arbeits-/Vorgehensweise:

1. Zu Beginn wird eine ”LTP-Pan” Datei eingelesen (Listing 3.1, Seite 28)
2. Darauffolgend wählt LTP-Pan einen Testtag aus und führt das mit ihm verbundene Testprogramm aus.

3. Gleichzeitig aktualisiert LTP-Pan eine Statusdatei (.zoo Dateiendung), die den Tester über den derzeitigen Testablauf-Status informiert.

Die ”LTP-Pan” Datei (Listing 3.1, Seite 28) repräsentiert eine Liste, in der die Test-Tags mit den Testprogrammen und den zugehörigen Aufrufparametern enthalten ist. LTP-Pan kann mit verschiedenen Aufrufoptionen gestartet werden:

- Mit der Option ’-a’ wird die Statusdatei festgelegt.
- Die Option ’-n’ legt den Test-Tag für das LTP-Pan Programm fest.

Anschließend erfolgt die Angabe des zu testenden Programms. Im Beispiel (Listing 3.2, Seite 29) wird das Testprogramm ”sleep 4” aufgerufen. Wie zu erkennen ist erfolgt die Ausgabe auf der Standardausgabe (STDOUT). Diese kann aber, wie bei Unix-Programmen üblich, umgeleitet werden. In der Statusdatei (Listing 3.2, Seite 29, Zeile 30-33) sind die Prozess-ID, der Test-Tag und ein Teil der Kommandozeilenparameter festgehalten. Beendete Tests werden durch ein ”#” in der ersten Spalte symbolisiert.

LTP-Scanner

Dieses Analyse-Tool übersetzt die LTP-spezifische Ausgabe (Listing 3.2, Seite 29) in eine Tabelle, die einen schnellen Überblick über die Testergebnisse gibt.

3.1.2 Zusammenfassung

Das LTP-Projekt bietet Entwicklern ein Tool, mit dem sie den Linux-Kernel und die Systembibliotheken ausgiebig testen können. Es beinhaltet eine Vielzahl von Testfällen, die das Erstellen eigener Testfälle meist überflüssig macht. Das LTP-DDT Projekt erweitert diese nochmals um eine Reihe an Plattform-Schnittstellentests. Für die LTP-Tools wurde die Programmiersprache ”C” verwendet, wie auch für die Testfälle. Einige Testfälle sind aber auch in Shell-Skripten implementiert.

Testdurchführungsbedingungen:

- Auf dem zu testendem Zielsystem (DUT) müssen das LTP-Pan Programm sowie die Testfälle/Testprogramme ausführbar vorliegen. Somit müssen diese Projektteile Cross-Kompiliert[6]werden.
- Für die nachfolgende Analyse ist es sinnvoll das Programm LTP-Scanner einzusetzen. Dieses muss nicht auf dem DUT ausgeführt werden, womit eine Cross-Kompilierung nicht nötig ist. Da die Ausgabe von LTP-Pan in eine Datei umgeleitet werden kann, welche später als Eingabe von LTP-Scanner dient.

[6] Bei einer Cross-Kompilierung erstellt ein System A (mit der CPU-Architektur A) Maschinencode für ein System B (mit der CPU-Architektur B). Dies ist im Embedded-Bereich üblich, da meist auf einem leistungsfähigen PC (System A, z.B. Intel-x86) entwickelt wird und anschließend das Ergebnis auf ein nicht so leistungsfähiges Zielsystem (System B, z.B. ARM-Cortex A9) übertragen wird.

Die Abbildung 3.1 auf Seite 30 stellt die Zusammenhänge der Einzelkomponenten nochmals grafisch dar.

Vorteile:

- Es ist ein großer Datensatz vorgefertigter Testfälle vorhanden.
  → Einarbeitungsaufwand wird gering gehalten, da die Testfälle übernommen werden können.
- Durch die Vorgabe des Coding-Styles wird der Wartungsaufwand der Testfälle verringert.
- Das LTP-DDT Projekt erweitert die Testfälle durch eine Vielzahl vorgefertigter Plattform-Schnittstellentests. Dies ist besonders für den Test eingebetteter Systeme interessant.
- Die Testfälle müssen nicht Cross-Kompiliert werden, wenn sie mittels eines Shell-Skripts implementiert wurden.
- Die Testfälle des LTP-Projekts (das LTP-DDT Projekt ausgenommen) sind sehr generisch gehalten.

Nachteile:

- Teile der Testsuite müssen Cross-Kompiliert werden.
- Die hinzugefügten Testfälle des LTP-DDT Projekts sind ist sehr Hardware-spezifisch und die Anpassung sehr zeitaufwändig.
- Das LTP-DDT Projekt hat kein festgelegtes Aktualisierungsintervall.
- Sowohl das LTP als auch das LTP-DDT Projekt bietet keine Datenbankanbindung. Dies kann für viele Aspekte hilfreich sein, z.B. zum automatisch globalem Speichern der Testergebnisse.
- Es wird vorausgesetzt, dass der Bootvorgang erfolgreich war.
  → Boot-Tests können nicht durchgeführt werden.
- Projekt bedingt sind die Testfälle ausschließlich auf das Testen eines Linux-Systems ausgelegt.

Listing 3.1: Beispiel LTP-Pan Datei

```
1  testtag testprogram -o one -p two other command line options
2
3  # This is a comment. It is a good idea to describe the test
4  # tags in your ltp-pan file. Tests programs can have different
5  # behaviors depending on the command line options so it is
6  # helpful to describe what each test tag is meant to verify
7  or # provoke.
8
9  # Some more test cases
10 mm01 mmap001 -m 10000
11
12 # 40 Mb mmap() test.
```

```
# Creates a 10000 page mmap, touches all of the map, sync's
# it, and munmap()s it.
mm03 mmap001 -i 0 -I 1 -m 100

# repetitive mmapping test.
# Creates a one page map repetitively for one minute.
dup02 dup02

# Negative test for dup(2) with bad fd
kill09 kill09

# Basic test for kill(2)
fs-suite01 ltp-pan -e -a fs-suite01.zoo -n fs-suite01 \
                                                -f runtest/fs

# run the entire set of file system tests
```

Listing 3.2: Beispiel LTP-Pan Aufruf

```
The most basic way to run ltp-pan is by passing the test
program and parameters on the command line. This will
run the single program once and wrap the output.

$ ltp-pan -a ltp.zoo -n tutor sleep 4

<<<test_start>>>

tag=cmdln stime=971450564

cmdline="sleep 4"

contacts=""

analysis=exit

initiation_status="ok"

<<<test_output>>>

<<<execution_status>>>

duration=103341903 termination_type=exited termination_id=0
corefile=no cutime=0 cstime=0

<<<test_end>>>

$ cat ltp.zoo

#9357,tutor,pan/ltp-pan -a ltp.zoo -n tutor sleep 4

#9358,cmdln,sleep 4
```

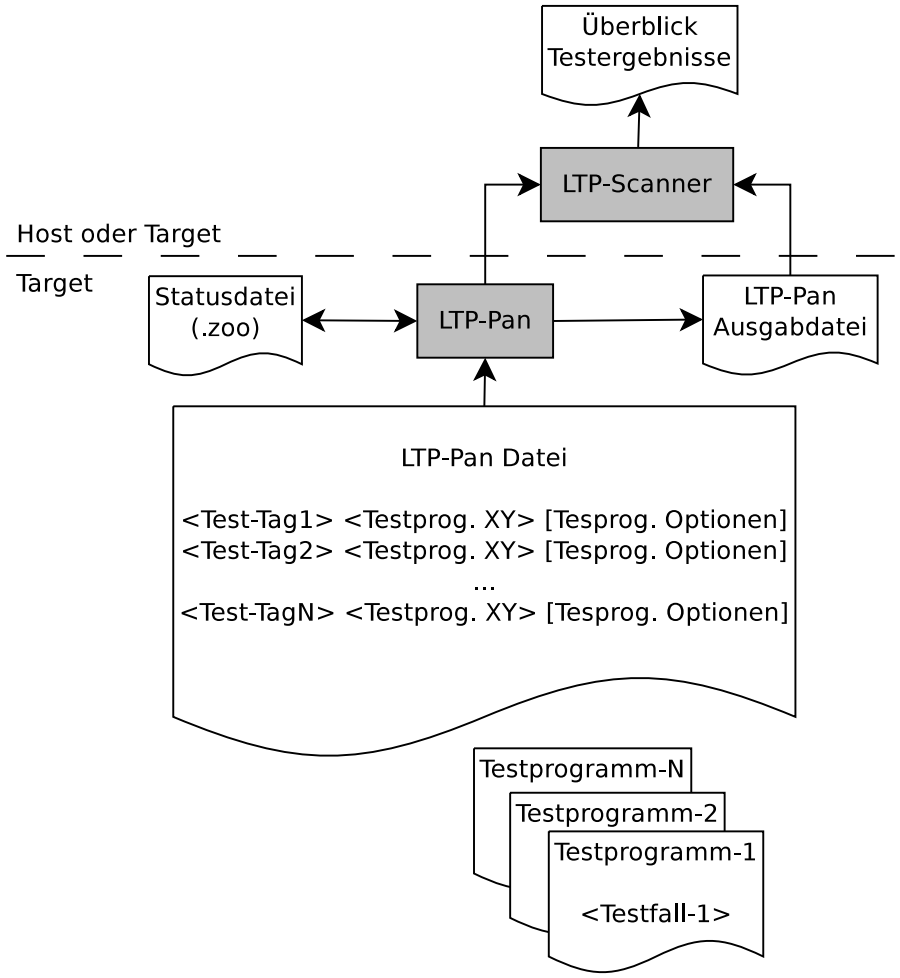


Abbildung 3.1: LTP Gesamtsystem

3.2 Opentest

3.2.1 Übersicht

Opentest ist ein skalierbares und verteiltes Testframework[7], welches auf dem ”Software Testing Automation Framework (STAF)” [14] basiert und von Texas Instruments entwickelt wurde.

> STAF ”is an open source, multi-platform, multilanguage framework desgined around the idea of reusable componentes, called services... The STAF framework provides the foundation upon which to build higher level solutions, and provides a pluggable approach supported across a large variety of platforms and languages” [14]

Texas Instruments hat das bestehende STAF-Framework durch zusätliche ”Services” erweitert und mit Testlink [15], einem Test-Management System, verbunden. Mit Opentest wird ein Testframework bereitgestellt, dass vollkommen auf Open Source Projekten basiert und durch dessen Hilfe eine Testautomation ermöglicht wird. Testfälle können so generisch implementiert werden, dass sie leicht auf verschiedene HW-Plattformen portierbar sind.

Die Abbildung 3.2 auf Seite 32 stellt das Zusammenspiel der beteiligten SW-Komponenten dar. Die folgenden Aufführungen sind für das Verständnis der Abbildung wichtig, da sie die Aufgaben der SW-Module beschreiben.

Opentest Hauptkomponenten

1. Test Management System (TestLink)
   Dessen Aufgabe ist es alle Daten wie z.B. Testfälle, Testergebnisse, usw. zentral zu speichern und zu organisieren.
2. Test Master Controller
   Dieser übernimmt das Scheduling[8]der Tests und sitzt als Bindeglied zwischen dem Test Management System und dem DUT.
3. Service Providers / Execution Engine Interface
   Diese stellen dem DUT Dienste zur Verfügung, welche benötigt werden um die Tests auszuführen.

Module des Test Master Controller

- Dispatcher
  Erhält alle Testanfragen des ”Test Management Systems” und interagiert mit dem ”Resource Manager” um den ”Service Provider” zu finden, an welchen

[7] testframework - dt. Testsystem
[8] scheduling - dt. Planung

die Testanfrage geleitet wird.

- Resource Manager
  Speichert alle im System verfügbaren ”Service Provider” mit ihren Eigenschaften und wählt den Bestmöglichen aus.
- Monitor
  Stellt Informationen zu den ”Service Providern” zur Verfügung.
- Test Management System (TMS) Writer
  Er übernimmt die Speicherung der Testergebnisse ins ”TMS”.

Service Provider Typen

- Build Execution Engines (BEEs)
  Kompilieren oder erhalten die zu testende SW.
- Test Execution Engines (TEEs)
  Führen die zu testende SW aus.

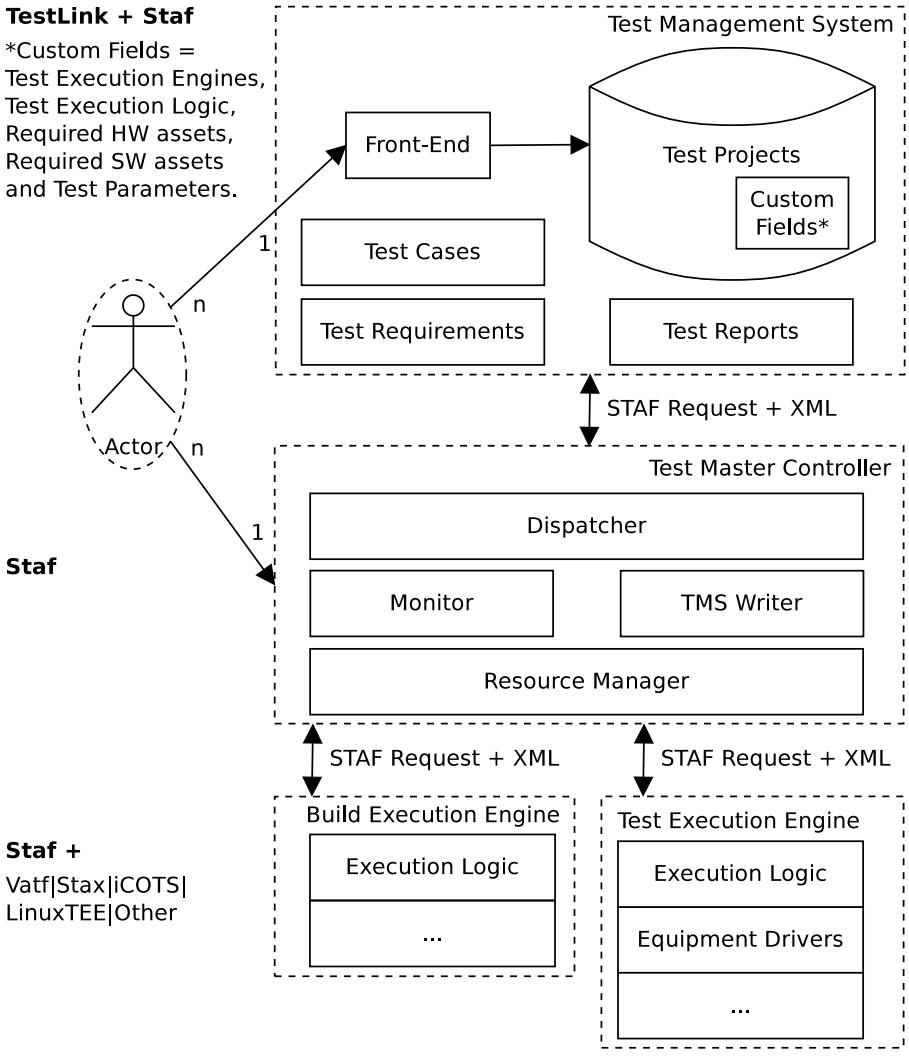


Abbildung 3.2: Überblick über die Opentest Architektur [3]

3.2.2 Zusammenfassung

Texas Instruments stellt mit Opentest ein vollständiges Testframework zu Verfügung, welches sich komplett aus Open Source Projekten zusammensetzt und mit Hilfe dessen eine vollständige Testautomation ermöglicht wird.

Opentest übernimmt die Verwaltung und Koordinierung der unabhängigen DUTs. Die einzige Voraussetzung zur Testausführung ist, dass die DUTs über einen ausführbaren ”STAF-Interpreter” verfügen, welcher vom ”STAF-Framework” bereitgestellt wird. Generische Testfälle werden durch die Verwendung von Variablen ermöglicht. Diese dienen als Platzhalter und werden während der Ausführung durch vorher festgelegte Werte ersetzt, die in sogenannten DUT-Konfigurationsdateien gespeichert sind.

Die Testdurchführung erfolgt direkt auf den DUTs. Nach Beendigung des Testdurchlaufs senden sie einen Testreport an den ”Test Master Controller”. Wie oben erwähnt können Tests generisch aufgebaut werden. Dazu wird das von Texas Instruments entwickelte ”Video Automation Test Framework (VATF)” verwendet. Dieses dient sozusagen als Zwischen-Schicht (Abstraktionsschicht) und stellt folgende Dienste zur Verfügung:

1. Das Lesen von Testparametern aus einer Datenbank.
2. Die Initialisierung des gesamten Testequipment, das für den Test benötigt wird.
3. Es ermöglicht die Ausführung von angegeben Testskripten.
4. Über die Angabe einer Testdatenbank wird das Speichern der Testergebnisse ermöglicht. Anschließend kann ein HTML-Testausführungsreport erstellt werden.

Im Anhang A.1 auf Seite 77 ist eine beispielhafte Implementierung eines solchen Testfalls dargestellt. Das Ergebnis eines Tests wird als Tabelle ausgegeben, die Aufschluss darüber gibt ob ein Test erfolgreich war oder fehlschlug. Für einen genaueren Bericht dient die Log-Datei der jeweiligen Tests.

Wie Anfangs erwähnt kann Opentest skalierbar aufgebaut werden, d.h. dass mehrere Opentest-Instanzen miteinander agieren können (Abbildung 3.3 auf Seite 34). Dabei muss beachtet werden, dass die ”Artifacts Databases” periodisch synchronisiert werden.

3.3 Autotest

Autotest [4] ist ein Testframework, welches ein vollständig automatisiertes Testen des Kernels und der HW ermöglicht. Das Projekt wurde Anfangs von zwei Google-Mitarbeitern initiiert und entwickelt [16] wird aber nun auch von Firmen wie Red Hat und IBM genutzt und weiterentwickelt.

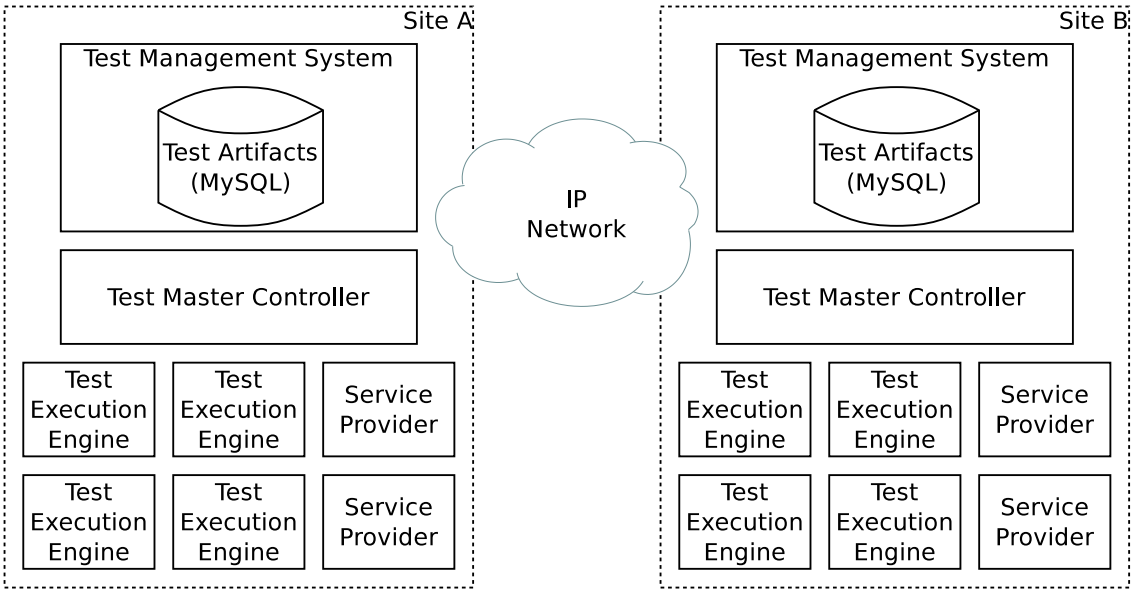


Abbildung 3.3: Beispiel Opentest Architektur, [3]

3.3.1 Übersicht

Das Testframework besteht aus der ”Benutzerschnittstelle”, dem ”Server” und dem ”Client” (Abbildung 3.4, Seite 35). Das dargestellte ”Result Repository” ist ein Verzeichnis auf dem Host-PC indem alle Testergebnisse gespeichert werden.

Benutzerschnittstelle (User-Interface)

Als Schnittstelle kann sowohl eine Web-basierte Anwendung, als auch ein Kommandozeilen-Tool verwendet werden. Die Funktionalität beider Tools ist identisch und umfasst

- die Verwaltung der Testjobs,
- die Verwaltung der Clients (DUTs) und
- die Darstellung der Testergebnisse.

Server

Dieser unterteilt sich in die folgenden drei Unterkomponenten:

- Datenbank
  In ihr sind alle Daten wie Testfälle, Clients, Benutzer, und alle Testjobs gespeichert.
- Dispatcher/”Monitor-DB”
  Er Wählt die Testjobs aus der Datenbank aus und startet einen entsprechenden ”autosrv” Prozess.
- ”autoserv” Prozesse
  Diese stellen die Verwaltungseinheiten dar, mit der die ”Clients” oder ein Verbund (”Cluster”) aus Clients im Server verwaltet werden. Jeder Testjob

ist mit einem ”autoserv” Prozess verbunden. Dessen Aufgabe ist es

- einen oder mehrere Clients zu verwalten,
- die Bereitschaft der Clients zu gewährleisten,
- die Testjobausführung zu verwalten und
- das die Autotest-SW vor jedem Testbeginn auf jedem beteiligten Client aktualisiert wird.

Der Dispatcher und die Datenbank sind unabhängig voneinander. Somit besteht die Möglichkeit ein verteiltes System aufzubauen, in welchem eine zentrale Datenbank existiert und auf die mehrere Dispatcher zugreifen können. Dies kann nötig werden, wenn die Anzahl der ”Clients” zu groß ist und eine Einzellösung vom Test-PC[9]zu viel Systemressourcen benötigt.

Client

In Autotest repräsentiert ein ”Client” ein DUT. Dies ist eine SW die auf dem DUT ausgeführt. Sie übernimmt die Steuerung des Tests und stellt somit die Test-Ausführungseinheit dar. Gestartet wird sie von der Serveranwendung mittels eines ”autoserv” Prozesses oder manuell vom Benutzer.

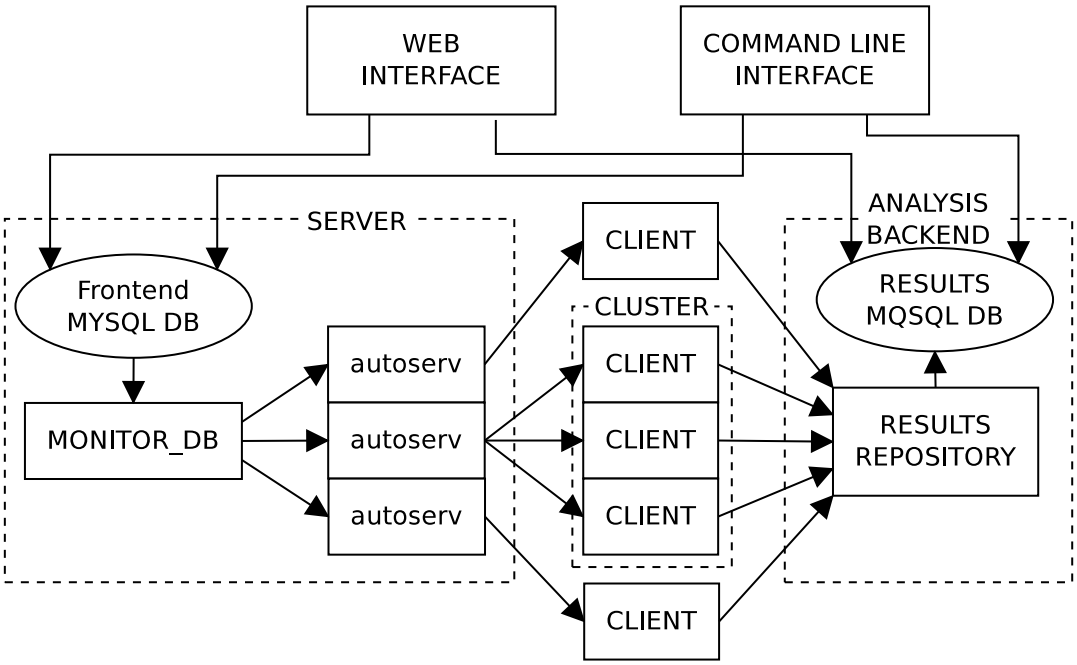


Abbildung 3.4: Autotest Gesamtstruktur [4]

[9] Dies ist der PC auf dem das Autotest Framework installiert ist und an dem die ”Clients” angeschlossen werden.

3.3.2 Zusammenfassung

Kerneigenschaften:

- Alle SW-Komponenten wurden in der Skriptsprache ”Python” [17] entwickelt.
- Es existiert eine ausführliche Dokumentation, welche die Einarbeitung in das Testsystem erleichtert.
- Durch die Unabhängigkeit von Dispatcher und Datenbank ist das Gesamtsystem gut skalierbar.

Damit Tests auf den DUTs ausgeführt werden können muss sichergestellt sein, dass dieses über einen ”Python-Skriptinterpreter” verfügen. Dieser wird für die Ausführung der Test-Ausführungeinheit benötigt, wie auch für die Tests, da diese ebenfalls in ”Python” implementiert werden. Dazu werden SW-Bibliotheken zur Verfügung gestellt, die das Entwerfen vereinfachen und HW-unabhängig machen.

Die Testdurchführung geschieht wie bei ”Opentest” auf dem DUT, anschließend wird ein Testreport zurück an den Hostrechner gesendet. Die oben erwähnte HW-Unabhängigkeit ist nur bedingt gegeben, da Autotest nicht in der Lage ist ”Embedded Testjobs[10]” auszuführen. Dies ist dadurch begründet, dass Autotest ursprünglich für x86 und PowerPC Architekturen entworfen wurde und bis heute noch keine Anpassung erfolgte, die das System für ”Embedded Testjobs” nutzbar macht.

Abschließend ist zu sagen, dass Autotest einen sehr guten Ansatz bietet, wie ein Testsystem aufgebaut werden kann. Allerdings ist es eher für das Testen großer Linux-Distributionen, bzw. Projekten geeignet. Nach der Analyse der ”Github Code frequency[11]” kann die Aussage getroffen werden, dass das Projekt die Hauptentwicklungsphase abgeschlossen hat und nur noch gewartet wird.

3.4 Linaro Automated Validation Architecture (LAVA)

> ”LAVA is a collection of participating components, the overall idea and evolving architecture that allows us to make testing, quality control and automation. LAVA-the-stack aims to make systematic, automatic and manual quality control more approachable for projects of all sizes.” [5]

3.4.1 Übersicht

Eine LAVA-Instanz besteht einer ”Benutzerschnittstelle”, dem ”Scheduler-Daemon” und dem ”Dispatcher” (Abbildung 3.5, Seite 37). Wie ”Autotest” und ”Opentest” bietet auch

[10] Beispielsweise können die Bootloader-Parameter und Boot-Quellen nicht angegeben werden.
[11] Sie zeigt an zu welchem Zeitpunkt Änderungen am Code eines SW-Projekts vorgenommen wurden.

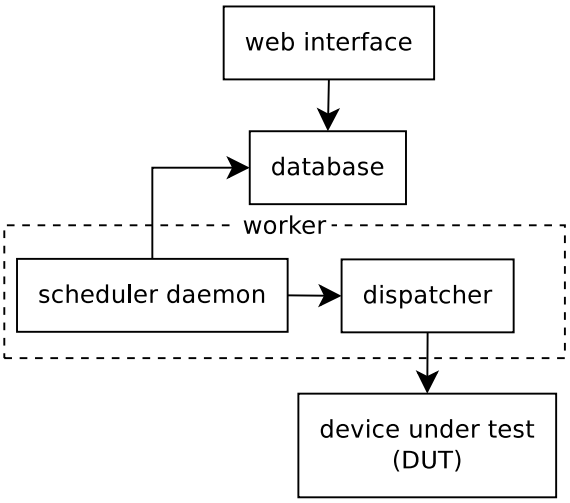


Abbildung 3.5: LAVA Architektur [5]

LAVA die Möglichkeit, als verteiltes System aufgebaut zu werden. Hierzu später mehr in Kapitel 4.

Benutzerschnittstelle

Der Benutzer kann über ein

- Webinterface,
- Kommandozeilen-Tool und
- "XML-RPC[12]" Interface [18]

mit der LAVA-Instanz interagieren. Die Funktionalität ist über alle Schnittstellen die gleiche, so können beispielsweise Testergebnisse und verfügbare DUTs angezeigt werden.

Scheduler-Daemon

Dieser ist für die Verteilung der Testjobs zuständig, dabei geht er wie folgt vor:

1. Er ruft die "Testjob-Queue" zyklisch auf, in ihr werden alle erteilten Testjobs gespeichert.
2. Er reserviert das entsprechende DUTs, welches für den Testjob vorgesehen ist.
3. Er triggert den Dispatchers, damit er mit der Testausführung beginnt.

Dispatcher

Der Dispatcher ist für die eigentliche Testdurchführung verantwortlich. Er managt die serielle Verbindung zum DUT, lädt alle benötigten Dateien wie z.B. den Kernel und führt den Test durch. Am Testende "sammelt" er die Testergebnisse ein.

[12] XML-RPC - Extensible Markup Language remote procedure call, damit kann SW aus der Ferne, über das Internet, ferngesteuert werden.

Testdefinition / Testaufbau

Ein Test besteht in LAVA aus zwei Komponenten der Testjob (Test-Setup) Datei und der Testlogik.

- Test-Setup Datei
  In ihr erfolgt die Definition des Tests, wie zum Beispiel
  - die Festlegung des zu verwendenden DUTs und
  - die Angabe des zu verwenden Linux-Kernels.

- Testlogik
  In ihr wird der Testablauf beschrieben.

Diese Trennung erlaubt eine einfach Testmigration für verschiedene Plattformen, da die gleiche Testlogik für mehrere DUTs verwendet werden kann, mehr in Kapitel 4.

3.4.2 Zusammenfassung

Mit LAVA stellt Linaro ein sehr umfangreiches Testsystem zur Verfügung. Alle Komponenten des Projekts werden unabhängig voneinander, von Linaro Mitarbeitern und der Open Source Community, in der Programmiersprache ”Python” entwickelt. Dies gewährleistet den von Linaro zugesicherten monatlichen Releaszyklus. Der Installationsaufwand von LAVA-Instanzen ist auf ein Minimum beschränkt, da LAVA ab ”Ubuntu 14.10”, bzw. ”Debian Jessie” in den offiziellen Distributions Archiven verfügbar ist.

Durch die Trennung von Testjobs, Testlogiken und HW-spezifischen Dateien wird eine größtmögliche HW-Unabhängigkeit geboten. Die Ausführung der Testfälle geschieht wie bei ”Autotest” und ”Opentest” lokal auf dem DUT. Dazu benötigt LAVA eine serielle Verbindung zum DUT. Ebenso muss das DUT über eine ”Shell” (z.B. die ”Busybox-Shell”) verfügen, da die Test-Ausführungseinheit von LAVA mittels ”Shell-Skript” implementiert wurde.

3.5 Gegenüberstellung der Testsysteme

Es ist schwierig die jeweiligen Testsysteme zu vergleichen, da alle unterschiedliche Implementierungsansätze haben. Dies führt unwiderruflich zu unterschiedlichen Eigenschaften. Mit Tabelle 3.1 wird trotzdem der Versuch unternommen die grundlegenden Eigenschaften zu vergleichen.

Anhand Tabelle 3.1 kann allerdings noch keine Aussage darüber getroffen werden, ob

das jeweilige Testsystem den Anforderungen entspricht. Dazu dient Tabelle 3.2, sie stellt die Systeme den zuvor getroffenen Anforderungen gegenüber. Bei Nichterfüllung eines K.O. - Kriteriums, muss das jeweilige Testsystem ausgeschlossen werden. Bei Nichterfüllung eines Wunschkriteriums, muss im SW-Entwicklungsteam entschieden werden, ob dies zum Ausschluss des Testsystems führt.

Sowohl in Tabelle 3.1 als auch in Tabelle 3.2 wird das LTP Projekt nicht mit aufgeführt. Grund hierfür ist, dass das Projekt kein vollständiges Testsystem darstellt. Es ist eher als Sammlung von vorgefertigten Testfällen zu betrachten. Diese können von jedem Testsystem benutzt werden. Allerdings muss die LTP-Suite für jedes DUT Cross-Kompiliert werden.

Wie in den Tabellen zu erkennen ist, ist LAVA das einzige Testsystem, welches alle K.O. - Kriterien erfüllt. Das Wunschkriterium nach vollständiger Unabhängigkeit können alle aufgeführten Systeme nicht erfüllen. Mit diesem Wissen und nach Absprache mit dem Entwicklungsteam, wurde sich daraufhin geeinigt, dass dieses Wunschkriterium für den jetzigen Zeitpunkt vernachlässigt werden kann. Somit wurde die Entscheidung zugunsten des LAVA-Frameworks getroffen.

	Opentest		Autotest		LAVA	
	Ja	Nein	Ja	Nein	Ja	Nein
Projekt						
Geringer Administrationsaufwand?	☐	☒	☒	☐	☒	☐
Einfache Testjob Erstellung?	☐	☒	☒	☐	☒	☐
Aktuelle Weiterentwicklung?	☐	☐	☒	☐	☒	☐
Weite Verbreitung?	☐	☒	☒	☐	☐	☐
Voraussetzungen						
Cross-Kompilierung nötig?	☒	☐	☐	☒	☐	☒
Sonstige?	☒	☐	☒	☐	☒	☐
Testausführung						
Auf Zielsystem?	☒	☐	☒	☐	☒	☐
Remote vom Host-Rechner aus?	☐	☒	☐	☒	☐	☒
HW-Unabhängigkeit gegeben?	☒	☐	☒	☐	☒	☐
zentrale Speicherung						
Testfälle?	☒	☐	☒	☐	☒	☐
Testjobs?	☒	☐	☒	☐	☒	☐
Testergebnisse?	☒	☐	☒	☐	☒	☐
Darstellung der Testergebnisse						
Webfrontend?	☒	☐	☒	☐	☒	☐
Log-Datei enthalten?	☒	☐	☒	☐	☒	☐
Export möglich?	☒	☐	☒	☐	☒	☐

Tabelle 3.1: Eigenschaften des Testsysteme

	Kriterien	Opentest	Autotest	LAVA
K.O. Kriterien	Wiederverwendbarkeit der Testfälle	✓	✓	✓
	Leichte Adaptierbarkeit der Testfälle auf andere Testhardware	✓ / –	✓	✓
	Einfache Wartung des Testsystems	–	✓	✓
	Automatische Test-Triggerung nach erfolgreichem Jenkins-Buildvorgang	✓	✓	✓
	Speicherung des Testergebnisses mit PASS/FAIL-Auswertung	✓	✓	✓
	Benachrichtigung (Email) nach Testablauf	✓	✓	✓
	Festlegung des Kernels, DTB, Rootfs	✓	–	✓
	Testmöglichkeit von Kernel, DTB, Rootfs	–	–	✓
Wunsch-kriterien	Möglichkeit der Einzeltestdurchführung am Arbeitsplatz	✓	✓	✓
	Minimale DUT Voraussetzung (funktionsfähige(r) serielle Schnittstelle und Netzwerkschnittstelle, Bootloader)	–	–	✓
	Ausführung von Performance- und Stresstests	✓	✓	✓
	Schnittstelle zur autom. Testdokumentationserstellung	✓	✓	✓
	Betriebssystem- und Systemunabhängikeit	–	–	–

Tabelle 3.2: Entscheidungsmatrix

4 Implementierung und Integration des ausgewählten Testsystems

Folgend soll die von Linaro entwickelte ”Linaro Automated Validation Architecure (LAVA)” näher betrachtet werden, um ein tieferes Verständnis[13]über die Arbeitsweise von LAVA zu schaffen. Im Anschluss daran wird dargestellt wie das Testsystem in die bereits vorhandene Architektur integriert wurde.

4.1 LAVA im Detail

4.1.1 Anforderungen

Prinzipiell ist die einzige Voraussetzung, dass das DUT über einen funktionsfähigen Bootloader verfügt. Damit allerdings der gesamte Funktionsumfang von LAVA genutzt werden kann, müssen das DUT und der Bootloader über zusätzliche Anforderungen verfügen[14].

DUT

- Das DUT muss eine serielle Schnittstelle besitzen, damit mit ihm kommuniziert werden kann.
- Es muss einen Ethernetanschluss besitzen, da über diesen der Datenaustausch (z.B. das Laden des Linux-Kernel Images) erfolgt.
- Ein Hardreset[15]muss ohne manuelles Eingreifen möglich sein.

Bootloader

- Der Bootloader muss über die serielle Verbindung gesteuert werden können.
- Der Startvorgang des Bootloader muss unterbrechbar sein. Zur Unterbrechung sendet der Host-PC ein beliebiges Zeichen über die serielle Verbindung an das DUT.
- Damit die Images über das Netzwerk bezogen werden können muss er das ”Trivial File Transfer Protocol (TFTP)” [19] unterstützen.

[13] LAVA ist ein sehr umfangreiches und leistungsfähiges Tool, deshalb kann in dieser Arbeit nicht alles dargestellt werden. Für weitergehende Informationen sei auf die offizielle Dokumentation [5] verwiesen.

[14] Das LAVA Testframework befindet sich derzeit in einer Umbauphase, die grundlegende Mechanismen verändert. Deshalb sei darauf hingewiesen, dass sich die hier aufgeführten Anforderungen bereits auf das neue SW-Design beziehen.

[15] Bei einem Hardreset oder Power-On Reset wird die Stromzufuhr zum DUT für einen kurzen Augenblick unterbrochen.

- Er muss verschiedene Boot-Medien unterstützen, damit möglichst viele Medien getestet werden können.

4.1.2 LAVA Instanzen – Ein verteiltes Testsystem

In Absatz 3.4 wurde bereits kurz auf die Funktionsweise eingegangen, diese soll nun näher betrachtet werden. Wie bereits erwähnt besteht LAVA aus dem LAVA-Server, -Dispatcher und -Scheduler Daemon.

Diese Betrachtungsweise ist für die Erklärung der Skalierbarkeit ungeeignet. Besser ist eine Unterteilung in ”LAVA-Master-Instanz” und ”LAVA-Worker-Instanz”. Die Master-Instanz stellt die Benutzerschnittstellen, eine zentrale Datenbank und die ”LAVA-Log” Dateien zur Verfügung.

- Benutzerschnittstelle
  Die Erklärung der Funktionsweise erfolgte bereits in Absatz 3.4 und soll deshalb nicht näher betrachtet werden.
- Zentrale Datenbank
  In ihr werden die Testjob-Aufträge und -Ergebnisse gespeichert. Außerdem organisiert sie die Worker-Instanzen mit den angeschlossenen DUTs. Als SW kommt ”PostgreSQL” zum Einsatz.
- LAVA-Log
  Diese Log-Dateien dienen der Fehlersuche bei einem Fehlverhalten der LAVA-Software. Sie sind nicht mit den Log-Dateien der Testergebnisse zu verwechseln, die einen detaillierten Testbericht darstellen.

Die Worker-Instanzen stellen den ”Scheduler-Deamon” und den ”Dispatcher” zur Verfügung. Die Funktionsweise dieser beiden SW-Komponenten wurde bereits in Absatz 3.4 erklärt, deshalb wird an dieser Stelle auf eine Erläuterung verzichtet.

Abbildung 4.1 stellt die Beziehung zwischen der Master-Instanz, den Worker-Instanzen und den DUTs dar. Alle LAVA-Instanzen können räumlich voneinander getrennt sein, wobei die Ausdehnung keine Rolle spielt. Allerdings sollte darauf geachtet werden, dass alle Instanzen über eine schnelle Netzwerkanbindung verfügen. Da bei der Testdurchführung ein sehr hoher Daten-Traffic verursacht werden kann, beispielsweise durch das Laden von großen Images. Eine Trennung von Worker und DUT ist nur bedingt möglich, da die Worker-Instanzen einen seriellen Zugriff auf die DUTs benötigen, hierzu später mehr. Neben der Möglichkeit des verteilten Systems, bietet LAVA die Option die Master- und Worker-Instanz in einer Einzel-Instanz zu vereinen (zusammengefasstes System).

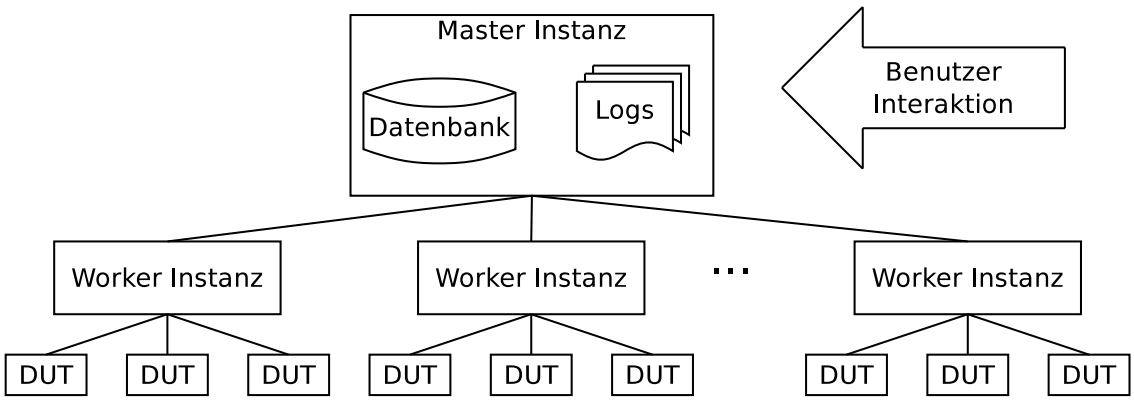


Abbildung 4.1: Aufbau eines verteilen LAVA-Testsystems

Vorteile

- Durch das verteilte System kann das Gesamtsystem schnell erweitert werden. Dies kann nötig sein wenn
  1. alle seriellen Schnittstellen eines Workers belegt sind, oder
  2. die Worker-Auslastung zu hoch ist.
- Alle Daten werden in einer zentralen Datenbank gespeichert.

Nachteile

- Der Erst-Konfigurationsaufwand eines verteilten Systems ist wesentlich höher als der eines zusammengefassten Systems.
- Alle Worker-Instanzen müssen synchronisiert sein, damit die Liste der unterstützen DUTs[16]über alle Worker-Instanzen gleich bleibt.
- Das Austauschen von DUTs von der Worker-Instanz A zu der Worker-Instanz B ist nicht ohne weiteres möglich, da die Worker neben der Liste der unterstützen DUTs eine lokale Liste führen, in der festgehalten wird welche DUTs an ihnen angeschlossen sind.

Vor allem der Vorteil der dynamischen Erweiterung macht das verteilte System zur besseren Wahl. Des weiteren befindet sich, wie schon erwähnt, die LAVA-SW im Umbau. Ist dieser abgeschlossen, wird die Synchronisation der einzelnen Worker-Instanzen überflüssig, da es eine zentrale Liste der unterstützen DUTs gibt.

4.1.3 LAVA Funktionsweise

Aufgrund der Komplexität von LAVA, wird die Erläuterung der Funktionsweise aus verschiedenen Gesichtspunkten betrachtet.

[16] DUTs die an das Testsystem angeschlossen werden können, unabhängig der Worker-Instanz.

Verteiltes System

Nachdem ein Testjob übermittelt wurde wird er in eine Warteschlange (Job-Queue) gestellt, welche Bestandteil der Master-Instanz ist. Die erteilten Testjobs besitzen die Priorität: High, Medium oder Low. Alle Jobs der gleichen Priorität werden nach dem ”First In First Out (FIFO)” Prinzip behandelt. Zuerst werden alle Jobs der höchsten Priorität abgearbeitet, bevor mit der nächst niedrigeren Priorität begonnen wird. Die Worker-Instanzen prüfen diese Queue zyklisch auf neue Testjobs. Ist dies der Fall überprüfen sie ob sie über ein entsprechendes DUT verfügen, um den Test auszuführen. Führt dieses gerade keinen anderen Test aus, wird der Testdurchlauf gestartet. Verfügt der Worker über kein entsprechendes DUT oder ist keins frei versucht er den nächsten Testjob aus der Queue zu bedienen.

Worker-Instanz / LAVA-Dispatcher

Die eigentliche Arbeit eines Workers übernimmt der LAVA-Dispatcher, der den gesamten Testablauf steuert. Wie bereits oben erwähnt, befindet sich das LAVA-Framework derzeit im Umbau. Die folgenden Aufführungen beziehen sich auf das neue SW-Design [6] (Abbildung 4.2, Seite 48).

1. Deploy
   Der Dispatcher bezieht alle benötigen Binaries (Kernel, DTB, rootfs, usw.) von den angegeben Quellen (siehe Abschnitt 4.1.4). Anschließend entpackt er sie wenn nötig und speichert sie in einem temporären Verzeichnis. Die angegebenen Testfälle (siehe Abschnitt 4.1.4) lädt der Dispatcher in ein extra Verzeichnis und ”mounted” sie anschließend in das rootfs unter dem Verzeichnis ”/lava-$hostname/”.
2. Boot
   Zuerst wird die serielle Verbindung zum DUT überprüft. Ist diese in Ordnung wird ein ”Hardreset” durchgeführt damit das DUT neu startet. Jeder LAVA-Dispatcher verfügt über sogenannte DUT-Konfigurationsdateien, damit er weiß wie das jeweilige Zielsystem (DUT) zu booten ist. In diesen werden die Boot-Abfolgen der DUTs beschrieben. Da die Zielsysteme über unterschiedliche Boot-Medien (SD-Card, MMC, Netzwerk, usw.) verfügen, sollte für jedes Medium eine Abfolge existiert. Während des Boot-Vorgangs wartet der Dispatcher auf einen ”String” bzw. eine Zeichenkette (meist ”Hit any key to interrupt autoboot”). Sobald er diese erkennt sendet er ein Zeichen über die serielle Verbindung, damit der Boot-Vorgang unterbrochen wird. Der ”String” und das Zeichen, welches daraufhin gesendet wird, kann in der DUT-Konfigurationsdatei angepasst werden (siehe unten). Diese Funktionsweise zeigt sogleich die ”Haupt-Funktionsweise” von LAVA, denn das gesamte System basiert ausschließlich auf dem ”Parsen” des ”Standard Outputs (STDOUT)” der seriellen Konsole. Nachdem der Boot-Vorgang unterbrochen wurde erwartet der Dispatcher einen ”Bootloader-Prompt”, dieser kann ebenfalls in der DUT-Konfigurationsdatei angepasst werden. Daraufhin

sendet er die zuvor hinterlegte Boot-Abfolge für das entsprechende Boot-Medium. Nachdem alle Boot-Anweisungen übertragen wurden und das Device bootet, wartet der Dispatcher auf den ”Linux Login-Prompt” woraufhin er sich automatisch anmeldet. Diese ”Strings” können ebenfalls angepasst werden.

3. Test
   Nach erfolgreichem Boot-Vorgang, beginnt der Dispatcher mit der Ausführung der Tests. Dazu ruft er eine ”Test-Ausführungseinheit” auf, die sich in dem vorher ”eingehängtem” Testverzeichnis befindet. Diese führt die im Testjob aufgeführten Testfälle, die sich im gleichen Verzeichnis befinden, aus.

Konfigurationsdateien

Wie oben bereits erwähnt werden die HW-spezifischen Eigenschaften der einzelnen DUTs in Konfigurationsdateien festgehalten. In ihnen wird die Kommunikation zum DUT, der Boot-Ablauf und DUT-spezifische Eigenschaften (z.B. ”Linux-Shell Prompt”) festgehalten. Dafür spezifiziert LAVA zwei Dateien.

1. Prozessor-Modul Datei (Listing 4.1)
   In dieser werden die Boot-Abfolgen (Bootloader-Kommandos) definiert.
2. Plattform Datei (Listing 4.2)
   Sie dient der Festlegung der DUT-Kommandos und DUT-spezifischen Eigenschaften. So wird beispielsweise
   - die Seriellen-Schnittstelle festgelegt,
   - das ”Hardreset” Kommando (z.B. Ein Kommando das ein Netzteil fernsteuert) definiert,
   - der DUT-Name unter dem es in LAVA geführt wird bestimmt und
   - die Angabe über das verwendete Prozessor-Modul gemacht.

Die Prozessor-Modul Dateien sind jene, die über alle LAVA-Worker hinweg synchron gehalten werden müssen. Anders als die Plattform Dateien, sie beschreiben ein spezielles DUT (eine Plattform), das an den Worker angeschlossen ist. Durch diese Trennung wird das Hinzufügen einer neuen Plattform, die ein bekanntes Prozessor-Modul einsetzt, vereinfacht (Abbildung 4.3). Parameter der Prozessor-Modul Datei können auch in der Plattform Datei stehen und umgedreht, dies wird aber nicht empfohlen.

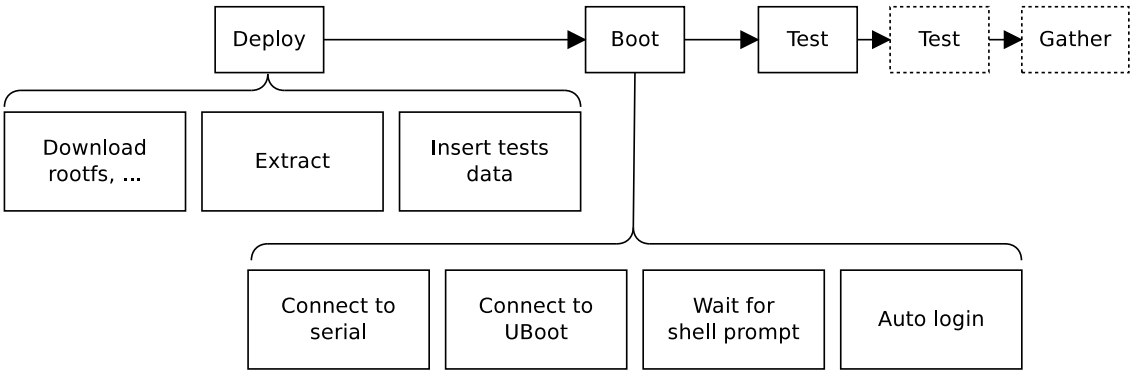


Abbildung 4.2: LAVA Dispatcher Design [6]

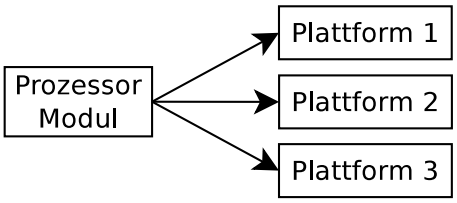


Abbildung 4.3: Zusammenhang der HW-Konfigurationsdateien

Listing 4.1: Beispiel einer Prozessor-Modul Konfigurationsdatei

```
client_type = bootloader

# ***************************** Boot-Commands *******************************
boot_cmds =

boot_cmds_ramdisk =

boot_cmds_nfs =
    setenv autoload no,
    setenv enet_clk_internal 1,
    setenv initrd_high "'0xffffffff'",
    setenv fdt_high "'0xffffffff'",
    setenv kernel_addr_r "'0x42000000'",
    setenv fdt_addr_r "'0x41000000'",
    setenv loadkernel "'tftp ${kernel_addr_r} {KERNEL}'",
    setenv loadfdt "'tftp ${fdt_addr_r} {DTB}'",
    setenv nfsargs "'setenv bootargs root=/dev/nfs rw \
                    nfsroot={SERVER_IP}:{NFSROOTFS},v3,tcp \
                    ip=dhcp console=ttyAPP3,115200'",
    setenv bootcmd "'dhcp; setenv serverip {SERVER_IP}; \
                    run loadkernel; run loadfdt; run nfsargs; \
                    bootz ${kernel_addr_r} - ${fdt_addr_r}'",
    boot

boot_options =
        boot_cmds

[boot_cmds]
default = boot_cmds
```

Listing 4.2: Beispiel einer DUT Konfigurationsdatei

```
1  device_type = tqma28aa
2  hostname = mba28x-tqma28aa-01
3  connection_command = telnet localhost 2000
4
5  bootloader_prompt = >
6
7  master_kernel = file:///home/lavaadmin/test-boot/linuximage
8  master_dtb = file:///home/lavaadmin/test-boot/imx28-mba28.dtb
9  master_nfsrootfs = file:///home/lavaadmin/test-boot/rootfs.tgz
10 #master_ramdisk =
11 master_str = root@MBa28x:~
12 #master_testboot_label =
13 #master_sdcard_label =
```

4.1.4 LAVA-Testjob Aufbau

Bestandteile:

1. Die Daten um einen Test zu Konfigurieren, beschrieben im ”JSON”-Dateiformat (JavaScript Object Notation).
2. Die Befehle die innerhalb des Tests ausgeführt werden, beschrieben im ”YAML”-Dateiformat (YAML Ain’t Markup Language).

JSON-Datei

Dies ist die Testjob-Datei, in ihr werden alle benötigen Angaben festgelegt die für den Testfall nötig sind. Anschließend wird sie an den LAVA-Server übermittelt.

Inhalt:

1. Jeder Testjob beinhaltet die Angabe der Testart (”Health-Check” oder ”User-Test”). ”Health-Checks” werden automatisch in bestimmten Intervallen aufgerufen, was der Sicherstellung dient, dass das DUT einen minimalen Funktionsumfang erfüllt. Schlägt dieser fehl wird das DUT ”Offline” geschaltet und ist für ”User-Tests” nicht mehr verfügbar. ”Health-Checks” besitzen die höchste Priorität.
2. Die Angabe des ”Timeouts” für jede Aktion innerhalb des Tests.
3. Die Festlegung des Test ”Logging-Levels”, womit angegeben wird, wie viele Informationen gespeichert werden. Diese Option entfällt allerdings ab dem neuem LAVA-Dispatcher Design.
4. Damit ein Test bei der späteren Auswertung schneller identifiziert werden kann, muss ihm ein ”Testjobname” zugewiesen werden.
5. Die Pfadangabe der ”unterstützten Dateien” (siehe unten).
6. Alle erforderlichen Parameter, welche für die ”unterstützten Dateien” benötigt werden.
7. Die Angabe der zu verwendenden DUT/DUTs (Plattformen).

8. Die Festlegung des Pfads, wo das Testergebnis anschließend gespeichert werden soll.

Unterstützte Dateien:

1. Dateien die für den Boot- und Testvorgang des DUTs benötigt werden. Dazu gehören die Root Filesystem (rootfs) und Kernel Images sowie die Device Tree Bloobs und Bootloader-Parameter.
2. Dateien die den Test beinhalten. Dies sind die YAML-Dateien, die entweder als Teil eines Repositories, oder als Einzeldatei hinzugefügt werden.

Im wesentlichen gibt es drei Aktionen, die in der JSON-Datei beschrieben werden.

1. ”deploy_linaro_image” bzw. ”deploy_linaro_kernel’:
   Lädt ein komplettes Image welches den Bootloader, Device Tree Blob, Kernel und das rootfs beinhaltet auf das Zielsystem. Sollen diese Angaben von Hand definiert werden, muss die Option ”deploy_linaro_kernel” verwendet werden. Bevor das DUT bootet, wird das ”LAVA Overlay” eingefügt, dies stellt den Zugriff auf die ”LAVA-Test Tools” und die ”Test-Definitionen”, welche in den nachfolgenden ”actions” definiert wurden, sicher.
2. ”lava_test_shell”:
   Bootet das Image und startet den ”lava_test_runner”, welcher die Kommandos aus der YAML-Datei ausführt.
3. ”submit_results”:
   Sammelt die Ergebnisdaten nach der Ausführung aller in der YAML-Datei enthaltenen Befehle. Anschließend sendet der LAVA-Dispatcher (eine Worker-Instanz) einen ”Bundle” mit den Ergebnissen und Metadaten des Testjobs an den LAVA-Server (Master-Instanz), damit er dem angegebenen ”Bundle-Stream” hinzugefügt wird. Die ”Ergebnis-Bundles” können angeschaut, heruntergeladen und in Filtern und Reports benutzt werden.

Die Ausführung der einzelnen ”actions” basiert auf der Reihenfolge, in welcher sie in der JSON-Datei angegeben wurden. In Listing 4.3 wird ein solcher LAVA-Testjob exemplarisch dargestellt.

Listing 4.3: Beispiel eines LAVA-Testjob (JSON-Datei) [5]

```
1  {
2      "health_check": false,
3      "logging_level": "DEBUG",
4      "timeout": 900,
5      "job_name": "kvm-basic-test",
6      "device_type": "kvm",
7      "actions": [
8          {
9              "command": "deploy_linaro_kernel",
10             "parameters": {
```

```
11                 "dtb":
12           "http://git.tqsc.de/LAVA/Last-Builds/MBa28-images/78/imx28-mba28.dtb",
13                 "kernel":
14           "http://git.tqsc.de/LAVA/Last-Builds/MBa28-images/78/linuximage",
15                 "nfsrootfs":
16           "http://git.tqsc.de/LAVA/Last-Builds/MBa28-images/78/root.tgz",
17                 "login_prompt": "TQMa28 login:",
18                 "username": "root"
19             }
20         },
21         {
22             "command": "lava_test_shell",
23             "parameters": {
24                 "testdef_repos": [
25                     {
26                         "git-repo":
27                             "git://git.linaro.org/qa/test-definitions.git",
28                         "testdef":
29                             "ubuntu/smoke-tests-basic.yaml"
30                     }
31                 ],
32                 "timeout": 1800
33             }
34         },
35         {
36             "command": "submit_results",
37             "parameters": {
38                 "stream": "/anonymous/example/",
39                 "server": "http://localhost/RPC2/"
40             }
41         }
42     ]
43 }
```

YAML-Datei

Dies ist die Testdefinitionsdatei und dient der übersichtlichen Beschreibung des ”eigentlichen” Testablaufs. Das YAML-Format ist allerdings nicht ausführungsfähig, deshalb wird es vorher von LAVA in ein Shell-Skript übersetzt, welches anschließend auf das DUT kopiert (in das rootfs ”gemounted”) wird (siehe oben). Jede YAML-Datei enthält Metadaten und Anweisungen.

Metadaten:

1. Zuerst erfolgt die Angabe des Dokumentformats.
2. Anschließend muss eine Kurzbeschreibung, die das Verhalten dieser Datei beschreibt angegeben werden.
3. Zuletzt erfolgt eine ausführliche Beschreibung der Datei.

Wird die YAML-Datei nicht durch ein SCM-System verwaltet, muss zusätzlich

noch eine Versionsangabe der Datei erfolgen. Die notwendigen Angaben können noch durch optionale erweitert werden.

Optionale Metadaten:

1. Die E-Mail Adresse des ”maintainers” der Datei.
2. Eine Liste von Linux-Distributionen, welche die aufgeführten Kommandos unterstützen.
3. Eine Liste von DUTs, von denen angenommen werden kann, dass sie die aufgeführten Kommandos ausführen können.

LAVA unterstützt die Anweisungen ”install”, ”run” und ”parse”. In der ”install” Phase werden alle für den Test benötigten SW-Pakete nachinstalliert (derzeit funktioniert dies nur für ”Debian/Ubuntu” basierte Distributionen). In der ”run” Phase wird der Testablauf beschrieben, was dem Hauptzweck der YAML-Datei entspricht. So können Befehle (”cd”,”echo”, usw.) aufgerufen werden und beliebige Skripte ausgeführt werden, solange der Skript-Interpreter vom DUT unterstützt wird.

Dabei ist darauf zu Achten, dass alle aufgerufenen Kommandos/Befehle auf dem DUT zur Verfügung stehen. Alle Tests laufen in einem eigenem Arbeitsverzeichnis. Wenn die YAML-Datei Teil eines Repositories ist, liegt in diesem Arbeitsverzeichnis die gesamte Repository-Struktur vor.

Ob ein Testfall erfolgreich war oder fehlschlug erkennt LAVA anhand des ”Exit-Status” der ausgeführten Befehle:

- Ist der Exit-Status gleich 0, signalisiert dies einen erfolgreichen Test.
- Wenn er größer 0 ist, signalisiert dies einen fehlerhaften Test.

Für manche Testfälle, oder eigen erstellte Testskripte, ist diese Möglichkeit nicht ausreichend. Da sie einen Erfolg oder Misserfolg über die Standardausgabe mitteilen. Für diesen Fall bietet LAVA einen Parser, der in der ”parse” Phase konfiguriert wird. Dieser erlaubt die Skriptausgabe in pass, fail, skip und unknown zu übersetzen. Darüber hinaus gibt es eine Vielzahl weiterer Einstellungsmöglichkeiten/LAVA-Befehlen die in den Testdefinitionen angegeben werden können.

Beispiele:

- Was soll als Testfall-Ergebnis aufgezeichnet werden und was nicht?
- Soll während des Tests eine Messung (Bsp. Laufzeit) stattfinden?
- Es kann eine temporäre Log-Datei angelegt werden.

Da die Beschreibung jeder einzelnen den Rahmen dieser Arbeit übersteigen würde und derzeit immer noch neue Befehle hinzukommen, wird an dieser Stelle auf die Dokumentation [5] verwiesen. Abschließend wird in Listing 4.4 eine Testde-

finition dargestellt, welche die letzten Aufführungen zusammenfasst. Listing 4.5 zeigt die Ausgabe des Parsers, aus der Testjob Log-Datei. Eine komplette Analyse der Log-Datei wird in Kapitel 5 behandelt. Wie zu erkennen ist, wird ”test 4a” (Listing 4.4, Zeile 23) nicht in der Parser-Ergebnisliste (Listing 4.5) aufgeführt, da sich die beiden Strings zwischen zwei Hochkommas (’ ’) befinden. Dadurch werden die Anführungszeichen durch den ”echo” Befehl mit ausgegeben (Listing 4.5 Zeile 15) und der Parser wurde so eingestellt (Listing 4.4, Zeile 30), dass er diese Ausgabe nicht berücksichtigt.

Listing 4.4: Beispiel einer LAVA-Testdefinition (YAML-Datei) [5]

```
1  metadata:
2      format: Lava-Test Test Definition 1.0
3      name: parser
4      description: "Basic single node test commands"
5      maintainer:
6          - marco.felsch@tq-group.com
7
8      scope:
9          - functional
10
11     devices:
12         - tqma6q
13         - tqma6d
14         - tqma6dl
15         - tqma6s
16         - tqma28aa
17         - tqma28ab
18 run:
19     steps:
20         - echo "test1a:" "pass"
21         - echo "test2a:" "fail"
22         - echo 'test3a:' 'skip'
23         - echo '"test4a:" "unknown"'
24         - lava-test-case echo1 --shell echo "test1b:" "pass"
25         - lava-test-case echo2 --shell echo "test2b:" "fail"
26         - lava-test-case echo3 --shell echo 'test3b:' 'skip'
27         - lava-test-case echo4 --shell echo '"test4b:" "unknown"'
28
29 parse:
30     pattern: "(?P<test_case_id>.*-*):\\s+(?P<result>(pass|fail|skip|unkown))"
```

Listing 4.5: Parser Ausgabe (Testjob Log-Datei)

```
1  <LAVA_DISPATCHER>2015-06-05 11:47:07 AM DEBUG: expect (1197):          \
2          '['<LAVA_TEST_RUNNER>: exiting', <class 'pexpect.EOF'>,        \
3          <class 'pexpect.TIMEOUT'>, '<LAVA_SIGNAL_(\\S+) ([^>]+)>',     \
4          '<LAVA_MULTI_NODE> <LAVA_(\\S+) ([^>]+)>',                     \
5          '<LAVA_LMP> <LAVA_(\\S+) ([^>]+)>',                            \
6          <_sre.SRE_Pattern object at 0x7fc9020a95a0>]'                  \
7  echo LAVA_ACK
8  test1a: pass
```

```
<LAVA_DISPATCHER>2015-06-05 11:47:07 AM DEBUG: expect (1197):           \
        ’[’<LAVA_TEST_RUNNER>: exiting’, <class ’pexpect.EOF’>,         \
        <class ’pexpect.TIMEOUT’>, ’<LAVA_SIGNAL_(\\S+) ([^>]+)>’,      \
        ’<LAVA_MULTI_NODE> <LAVA_(\\S+) ([^>]+)>’,                      \
        ’<LAVA_LMP> <LAVA_(\\S+) ([^>]+)>’,                             \
        <_sre.SRE_Pattern object at 0x7fc9020a95a0>]’
"test4a:" "unknown"
<LAVA_DISPATCHER>2015-06-05 11:47:26 AM INFO: lava_test_shell seems to \
have completed

======
parser
======

Test case                                   Result          Measurement
----------------------------------------- -------- --------------------
test1a                                       PASS
test2a                                       FAIL
test3a                                       SKIP
test1b                                       PASS
echo1                                        PASS
test2b                                       FAIL
echo2                                        PASS
test3b                                       SKIP
echo3                                        PASS
echo4                                        PASS

<LAVA_DISPATCHER>2015-06-05 11:47:26 AM DEBUG: setting status pass
<LAVA_DISPATCHER>2015-06-05 11:47:26 AM DEBUG: finally status pass
<LAVA_DISPATCHER>2015-06-05 11:47:26 AM INFO: [ACTION-E] lava_test_shell is \
finished successfully.
```

4.2 Integration des Testsystems

4.2.1 Überblick über den Gesamtablauf

Wie bereits in Abschnitt 2.2 aufgeführt soll die vorhandene Infrastruktur (Abbildung 4.4, Seite 55) durch ein Testsystem erweitert werden (Abbildung 4.5, Seite 55), wodurch ein geschlossener Kreislauf hergestellt wird. Dieser erweitert die Eigenschaften und das Verhalten des bisherigen Systems.

Erweiterte Eigenschaften:

1. Nach Abschluss eines erfolgreichen Build-Vorgangs, wird automatisch ein Testjob an LAVA gesendet. Anschließend erfolgt die Testdurchführung, vorausgesetzt die entsprechende Zielhardware (DUT) ist verfügbar und bereit.

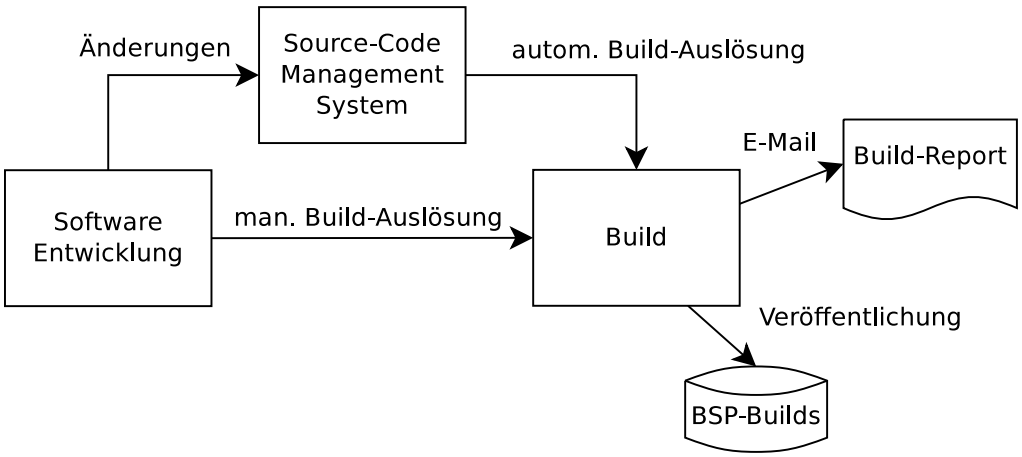


Abbildung 4.4: Softwareentwicklungszyklus vorher

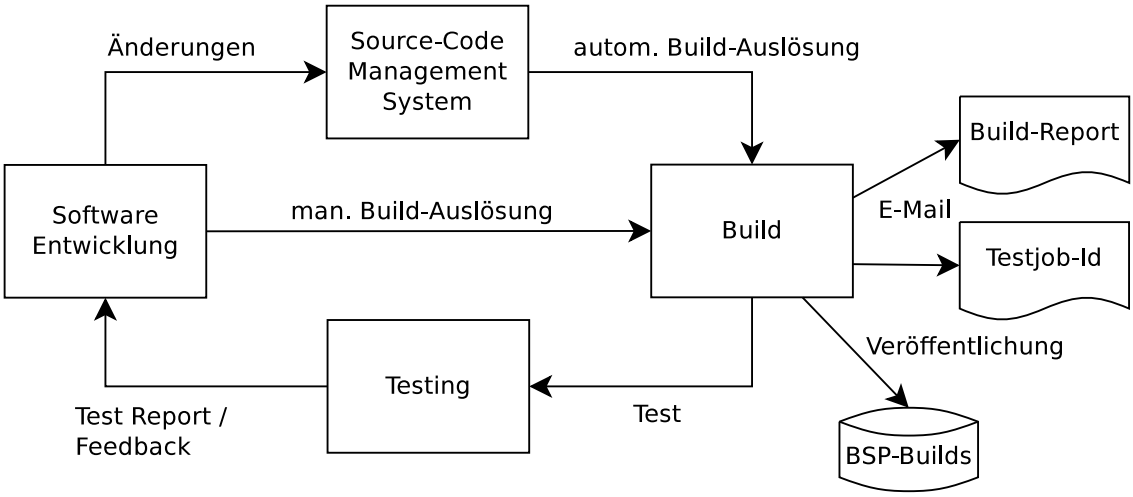


Abbildung 4.5: Softwareentwicklungszyklus danach

2. Nachdem der Testjob übermittelt wurde, sendet der Build-Server eine E-Mail an alle SW-Entwickler, in der die Testjob-ID und die Adresse der LAVA-Server-Instanz (Master-Instanz) enthalten ist. Dadurch ist es ihnen möglich die Testergebnisse mit den entsprechenden Log-Nachrichten zu analysieren.

Weil Abbildung 4.5 den Zusammenhang ausschließlich schematisch betrachtet, wird das Zusammenwirken der einzelnen SW-Komponenten in Abbildung 4.6 dargestellt. Die LAVA-Server Instanz entspricht der oben aufgeführten Master-Instanz und der Jenkins-Server (Abbildung 4.6) der Build-Stage (Abbildung 4.5). Für weitere Erläuterungen über die Aufgaben der einzelnen Instanzen sei auf frühere Kapitel verwiesen.

4.2.2 Voraussetzungen

Damit eine Testinfrastruktur nach Abbildung 4.6 aufgebaut werden kann, müssen zu den bereits aufgeführten Anforderungen (Abschnitt 4.1) zusätzliche getroffen werden.

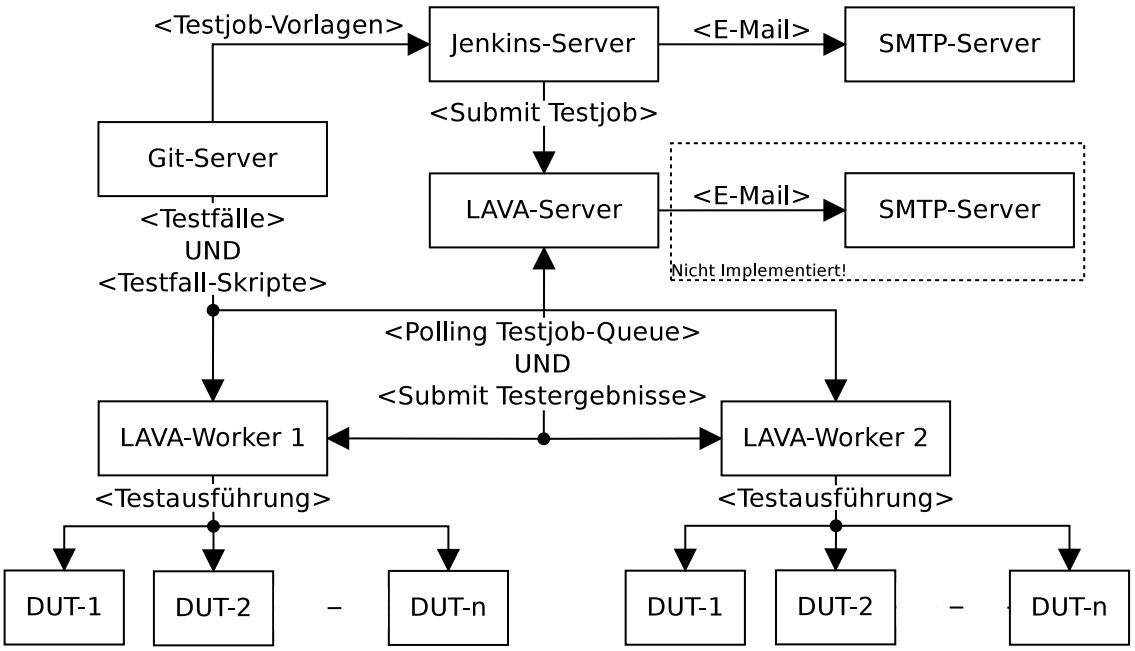


Abbildung 4.6: Überblick Testinfrastruktur

Worker Instanzen

Jede Worker Instanz muss mindestens zwei Netzwerkschnittstellen besitzen. Die erste Schnittstelle ist an das firmeninterne Netzwerk angeschlossen. Mit der zweiten Schnittstelle wird ein eigenes Testnetzwerk aufgebaut, an das die DUT's angeschlossen werden (Abbildung 4.7). Über die "eth0" Schnittstelle bezieht der LAVA-Dispatcher alle relevanten Testdaten und speichert sie temporär lokal zwischen (siehe oben). Weiterhin erfolgt über diese die Kommunikation zwischen den einzelnen LAVA-Instanzen. Über "eth1" und "eth2" werden alle relevanten Daten (z.B. Kernel-Image und DTB) an die DUTs übertragen.

Zugriffsrechte

Der Jenkins-Server benötigt auf dem Git-Server Lese-Rechte, damit er die Testjobs beziehen kann, dazu benutzt er das Secure Shell (SSH) Protokoll. Weiterhin benötigt er entsprechende Rechte, um Testjob-Aufträge an die LAVA-Server Instanz zu erteilen.

Jede LAVA-Worker Instanz benötigt für das Testdefinition-Repository (siehe Abschnitt 4.3) entsprechende Lese-Rechte auf dem Git-Server. Das Repository muss über das git-Protokoll oder über das Hypertext Transfer Protocol (HTTP) erreichbar sein, da die Worker-Instanzen ausschließlich diese beiden Protokolltypen unterstützt.

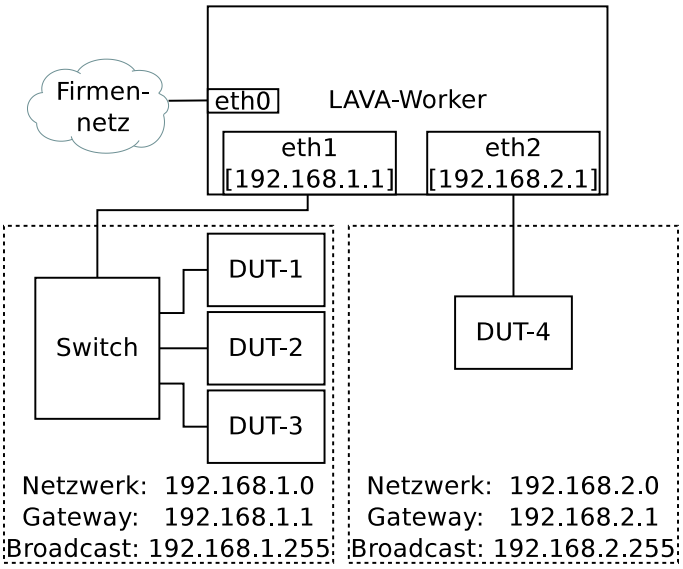


Abbildung 4.7: Beispielkonfiguration Netzwerkschnittstellen

4.2.3 Überblick der Jenkins-Jobs

Um eine schnelle Integration des Testsystems zu gewährleisten werden die vorhanden ”BSP Build-Jobs” durch minimale Anpassungen erweitert und nicht komplett neu entwickelt. Dies erfordert den Aufbau einer so genannten ”Job-Queue” (Abbildung 4.9), was die ”BSP Build-Jobs” von den neuen Jenkins Jobs entkoppelt. Der Informationsaustausch zwischen den einzelnen Jobs geschieht mittels ”Property-Files”, ein Austauschmechanismus der Jenkins-SW. In ihnen sind alle Umgebungsvariablen mit ihren entsprechenden Werten gespeichert, welche für den Folgejob wichtig sind. Damit die Datenintegrität sichergestellt wird, müssen die Jobs geeignet gegeneinander verriegelt werden, dies wird durch das Schlosssymbol (Abbildung 4.9) symbolisiert. Durch die Verriegelung wird sichergestellt, dass ein Job erst wieder gestartet werden kann, wenn der letzte, mit ihm verriegelte Job beendet ist.

Beispiel:
In Abbildung 4.8 wird eine Job-Queue dargestellt, die aus 4 Jobs besteht. ”Job1” kann erst wieder begonnen werden, wenn ”Job2” und ”Job3” beendet sind. Das Ende von ”Job4” spielt hingegen keine Rolle für ”Job1”.

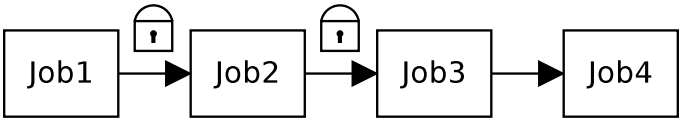


Abbildung 4.8: Beispiel Jenkins Verriegelungsmechanismus

Die Übermittlung der LAVA-Testjobs übernimmt ausschließlich der Jenkins-Server (Abbildung 4.6, Seite 56). Dies hat Vor- und Nachtteile, welche aber nicht näher betrach-

tet werden sollen. In der derzeitigen Implementierung wird ein erfolgreicher ”BSP-Build Job” vorausgesetzt bevor ein Testjob an LAVA übermittelt wird. Dies ist allerdings für den ”normalen” Gebrauch nicht wirtschaftlich, da ein Kompiliervorgang bis zu vier Stunden und länger dauern kann. Deshalb ist vorgesehen, dass ein Jenkins-Job entwickelt wird, der als manueller LAVA-Testjob Trigger fungiert. Mit diesem soll eine einfache LAVA-Testjob Erstellung und Übermittlung, ohne vorangegangenen ”BSP-Build Job”, ermöglicht werden.

Neuer Jenkins-Ablauf (Abbildung 4.9):

0. Startpunkte sind die vorhandenen BSP-Build Jobs. Diese wurden nur minimal erweitert, damit jobabhängige Metadaten für die Folgejobs zur Verfügung gestellt werden können.
1. In der Prepare-Stage werden LAVA-abhängige und DUT-spezifische Variablen gesetzt. Dies ist von besonderer Bedeutung, wenn das vorausgehende BSP-Build Projekt ausführbare Dateien (Binaries) für verschiedene Hardware-Typen erstellt hat.
2. Die Build-Stage wurde wie in Abbildung 4.10 dargestellt implementiert. Sie teilt sich in die Bereiche der automatischen und der manuellen Triggerungen:
   - Automatischer Trigger: Es wird ein LAVA-Job Template, hierzu später mehr in Abschnitt 4.3, aus einem Git-Repository geladen und mit den vorher gesetzten Variablen ausgefüllt.
   - Manueller Trigger: Es werden alle für den LAVA-Testjob benötigen Job-Templates aus einem Git-Repository geladen und zu einem LAVA-Testjob zusammen gestellt.
3. In der Submit-Stage wird der ausgefüllte LAVA-Testjob an den LAVA-Server übermittelt.

4.3 Sicherstellung der Wiederverwendbarkeit

Das Kriterium der Wiederverwendbarkeit war während dieser Arbeit ein K.O.-Kriterium, welches unbedingt zu erfüllen ist.

Trennung von Testjob und Testfall/Testdefinition

Dies ermöglicht es eine Testlogik in verschiedenen Testjobs einzusetzen. Somit kann eine einmal erstellte und getestete Logik in vielen Testjobs (z.B. auf verschiedenen DUTs) zum Einsatz kommen.

Trennung der Testfälle und der HW

Durch die Konfigurationsdateien (Plattform-/ Prozessor-Modul Datei) ist eine schnelle Integration der Testjobs möglich, da in ihnen keine weiteren HW-spezifischen Angaben gemacht werden müssen. Außerdem können DUTs schnell in das Testsystem integriert werden (siehe Punkt 4.1.3 ”Worker-Instanzen”, Seite 45).

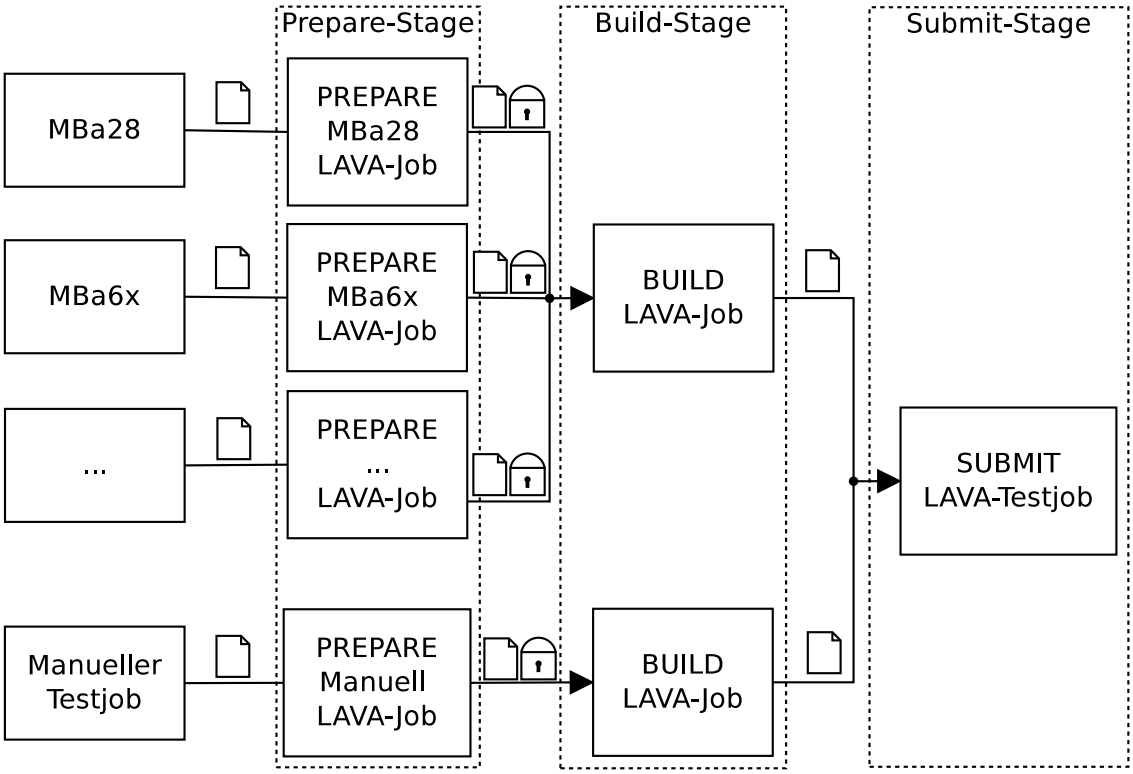


Abbildung 4.9: Jenkins Job-Queue

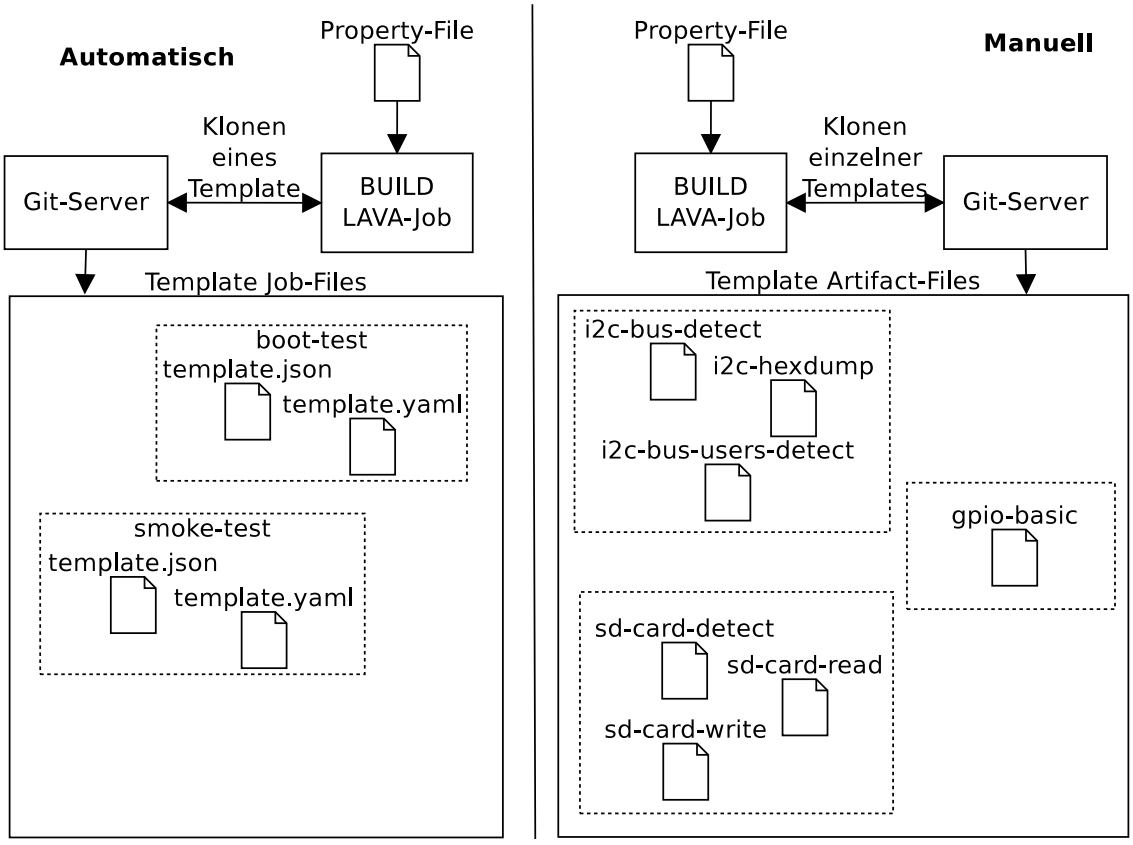


Abbildung 4.10: Build Stage

4.3.1 Erhöhung der Wartbarkeit

Damit neben der Wiederverwendbarkeit auch die Wartbarkeit erhöht wird, wurden zwei Git-Repositories, nach Abbildung 4.11, angelegt. Jedes Repository verfügt über eine ”README-Datei”, welche die Struktur des jeweiligen Repositories beschreibt und Regeln für die Arbeit mit dem Repository festlegt.

Testjob-Repository

- Dient der Verwaltung der LAVA-Testjobs.
- Zugriff auf dieses Repository benötigt nur der Jenkins-Server.
- Im ”script” Verzeichnis befinden sich Skripte, welche Jenkins in bestimmten ”Stages” benötigt.
- Das ”template” Verzeichnis beinhaltet fertige LAVA-Job Templates. Sie liegen im JSON- und YAML-Format vor, da LAVA einen Umstieg von JSON auf YAML vorbereitet.
- Im ”template-artefacts” Verzeichnis liegen ausschließlich YAML-Datein vor, die der Jenkins-Server für die manuelle LAVA-Testjob Triggerung benötigt.

Testdefinitions-Repository

- Dient der Verwaltung der LAVA-Testdefinitionen.
- Ausschließlich die LAVA-Worker Instanzen benötigen Zugriff auf dieses Repository.
- Testdefinitionen die von einer bestimmten Distribution abhängig sind, werden in entsprechenden Unterordnern gespeichert, beispielsweise ”ptxdist”.
- Testdefinitionen die für mehrere Distributionen gleich sind, werden in das ”common” Verzeichnis eingeordnet. Der Aufbau dieses Verzeichnisses ist dem ”sysfs[17]” nachempfunden. Jede Testdefinition-Klasse enthält eine ”*-runall.yaml” Datei, diese führt alle Einzel-Testdefinitionen sequentiell aus.
- Die jeweiligen ”scripts”-Verzeichnisse enthalten die Skripte, die in den Testdefinitionen aufgerufen werden. Die Skript-Namen sind gleich den Testdefinitionsnamen.

Regeln zur Testerstellung

Jeder Test soll nach der UNIX-Philosophie: ”Mache nur eine Sache und mache sie gut.[18]” aufgebaut sein. Dieser Grundsatz soll vor allem bei den Testdefinitionen und den entsprechenden Skripten umgesetzt werden, da sonst die Wiederverwendbarkeit verringert wird. Es ist ratsam diesen Grundsatz auch für die Testjobs anzuwenden, sofern dies möglich ist. Das soll heißen, dass ein Testjob die gleichnamige Testdefinition aufruft. Somit wird die Wartbarkeit und Verständlichkeit erhöht, da schon bei dem Aufruf des jeweiligen Testjobs eindeutig ist, welche/r

[17] Dies ist virtuelles Dateisystem, welches vom Linux-Kernel bereitgestellt wird.

Testdefinition/Testfall ausgeführt wird.

Namenskonvention

- Alle Dateien und Ordner, mit Ausnahme der ”README” Datei, müssen klein geschrieben werden.
- Für Testdefinitionen und Testjobs, die unter den ”bus” und ”class” Verzeichnissen aufgeführt sind lautet der Name wie folgt:

  <DIR_NAME> - <NAME>.*

  Beispiele: i2c-bus-detect.json, led-on.yaml
- Als ”<NAME>” sind aussagekräftige Namen zu wählen.
- Worttrennungen erfolgen durch ein Minus ” - ”.

[18] KISS - Keep it simple, stupid

Test-Jobs Repository

```
/
├── README
├── scripts/
│   ├── post-lava-job.py
│   ├── yaml-to-json.py
│   └── build-lava-job.py
├── templates/
│   ├── boot-test.json
│   ├── boot-test.yaml
│   ├── smoke-test.json
│   ├── smoke-test.yaml
│   ├── ...
│   ├── benchmark/
│   │   └── ...
│   ├── bus/
│   │   ├── i2c/
│   │   │   ├── i2c-bus-detect.json
│   │   │   ├── i2c-bus-detect.yaml
│   │   │   ├── i2c-bus-users-detect.json
│   │   │   │
│   │   │   │
│   │   │   └── ...
│   │   └── pcie/
│   │       └── ...
│   └── class/
│       └── led/
│           ├── led-on.json
│           ├── led-off.yaml
│           └── ...
└── template-artefacts/
    ├── benchmark/
    │   └── ...
    ├── bus/
    │   ├── i2c/
    │   │   ├── i2c-bus-detect.yaml
    │   │   ├── i2c-bus-users-detect.yaml
    │   │   │
    │   │   │
    │   │   └── ...
    │   ├── spi/
    │   │   └── ...
    │   └── ...
    ├── class/
    │   ├── gpio/
    │   │   ├── gpio-out.yaml
    │   │   ├── gpio-in.yaml
    │   │   └── ...
    │   └── led/
    │       ├── led-on.yaml
    │       ├── led-off.yaml
    │       └── ...
```

Test-Definitions Repository

```
/
├── README
├── ptxdist/
│   ├── busybox.yaml
│   ├── kernel-version.yaml
│   ├── i2c ~> ../common/bus/i2c
│   └── scripts/
│       ├── busybox.sh
│       └── ...
├── yocto/
│   └── ...
└── common/
    ├── benchmark/
    │   └── ...
    ├── bus/
    │   ├── i2c/
    │   │   ├── i2c-runall.yaml
    │   │   ├── i2c-bus-detect.yaml
    │   │   ├── i2c-bus-users-detect.yaml
    │   │   │
    │   │   │
    │   │   └── ...
    │   ├── spi/
    │   │   ├── spi-runall.yaml
    │   │   └── ...
    │   └── ...
    ├── class/
    │   └── led/
    │       ├── led-runall.yaml
    │       ├── led-on.yaml
    │       └── ...
    └── scripts/
        ├── benchmark/
        │   └── ...
        ├── i2c/
        │   ├── i2c-bus-detect.sh
        │   ├── i2c-bus-users-detect.sh
        │   └── ...
        ├── led/
        │   ├── led-on.sh
        │   └── ...
        └── ...
```

Abbildung 4.11: Repository Struktur

5 Beispielimplementierung CAN-BUS Test

5.1 Hintergrundinformationen

Der CAN-Bus ist ein Zweidraht-Bus und zählt zu der Familie der Feldbusse. Das CAN Protokoll beschreibt die OSI-Schicht 1 (Physical Layer) und 2 (Data Link Layer). Für eine ausführliche Beschreibung des Busses sei auf die offizielle Dokumentation [20]. Wichtig zu wissen ist, dass es ein Multimaster-Bus ist, der priorisierte Datenpakete verschickt.

Die Konfiguration und Nutzung des CAN-Controllers ist unter Linux sehr intuitiv, da der Linux-Kernel bereits entsprechende CAN-Treiber und einen CAN-Stack namens ”SocketCAN” bereitstellt. Durch ”SocketCAN” wird es dem Benutzer ermöglicht die CAN-Schnittstellen über Sockets anzusprechen, wie es auch beim ”Internet Protokoll (IP)” gemacht wird. Dies ermöglicht die Nutzung von Linux-Tools, wie ”ifup” und ”ifdown”. Darüber hinaus wird von den meisten Linux-Distributionen ein CAN-Tool-Set angeboten, das Programme zur Kommunikation, zum Debuggen und zum Konfigurieren der CAN-Schnittstelle bereitstellt.

5.2 Versuchsbeschreibung

Im Versuch sollen zwei DUTs über ein CAN-Netzwerk miteinander kommunizieren. Dazu wird die CAN-Schnittstelle-1 des ersten DUTs mit der CAN-Schnittstelle-1 des zweiten DUTs verbunden (Abbildung 5.2, Seite 67). Mit dem Versuch soll die Funktionalität der CAN-Schnittstellen der beiden DUTs sichergestellt werden. Die Versuchsdurchführung ist auf beiden DUTs gleich, deshalb wird sie nur einem beschrieben.

Versuchsdurchführung:

1. CAN_INIT:
   In diesem Schritt wird die CAN-Schnittstelle initialisiert. So erfolgt die Festlegung der Bitrate, CAN-Schnittstelle und die CAN-Nachrichtennummer (CAN-ID) auf die ”gehört” werden soll.
2. SYNC:
   Diese Phase dient der Sicherstellung, dass die beiden DUTs ihre Init-Phase abgeschlossen haben.
3. CAN_SEND:
   In dieser Phase erfolgt die Nachrichtenübertragung. Dazu wird die CAN-Nachricht mit ihrer entsprechenden CAN-Nachrichtennummer, die CAN-Schnittstelle auf der

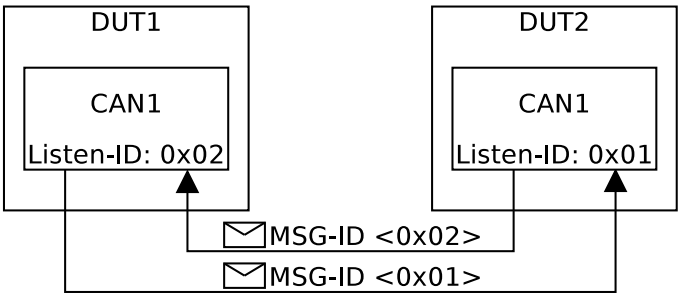


Abbildung 5.1: Versuchsablauf CAN-Test

gesendet werden soll sowie die Nachrichtenanzahl festgelegt. Anschließend erfolgt die Datenübertragung.

4. SYNC:
   Mit der zweiten Synchronisationsphase wird sichergestellt, dass alle DUTs ihre Nachrichten übertragen haben.
5. CAN_RESULT:
   Abschließend erfolgt die Auswertung der Testdurchführung.

Mit Abbildung 5.1 wird der Ablauf nochmals grafisch dargestellt.

5.3 Versuchsaufbau

Verwendete HW

- 1x MBa28x-Plattform Rev. 103 mit TQMa28-AA Prozessor Modul Rev. 200
- 1x MBa6x-Plattform Rev. 0200 mit TQMa6s-AB Prozessor Modul Rev. 0103

Verwendete Schnittstellen und Einstellungen

- Schnittstelle:
  Auf beiden DUTs kam die CAN1 Schnittstelle zum Einsatz.
- Einstellung:
  Beide CAN-Schnittstellen müssen mit einem 120 Ohm Widerstand terminiert werde. Dies erfolgt mittels der On-Board Widerstände, die über entsprechende Schalter AN/AUS geschaltet werden können.

Testjob

Er wurde manuell mittels des Web-Interfaces erstellt und übermittelt. LAVA bietet zwei verschiedene Testjob-Arten an Single-Node Tests und Multi-Node Test. Single-Node Tests werden auf einem DUT ausgeführt und Multi-Node-Tests auf mehreren.

Da der Aufbau eines LAVA-Testjobs bereits betrachtet wurde soll nur auf die Besonderheit des Multi-Node Testjobs eingegangen werden, der komplette Testjob

ist in Anhang A, Listing A.1, auf Seite 78 aufgeführt. Die Besonderheit ist statt nur einem DUT, eine ganze Gruppe anzugeben (Listing 5.1). Jede Gruppe kann aus einem oder mehreren DUTs bestehen (Angabe unter ”counts”), weiterhin kann jeder Gruppe eine sogenannte Rolle (”role”) zugewiesen werden. Über diese ist es möglich ”actions” einzelnen Gruppen zuzuordnen (Listing 5.2). Alle weiteren Testjobangaben sind gleich der, der Single-Node Tests.

Listing 5.1: Multi-Node Testjob DUT-Gruppen

```
"device_group": [
{
    "role": "can-dut1",
    "count": 1,
    "device_type": "tqma28aa"
},
{
    "role": "can-dut2",
    "count": 1,
    "device_type": "tqma6s"
}],
```

Listing 5.2: Multi-Node Testjob DUT-Gruppen Aktionen

```
{
    "command": "boot_linaro_image",
    "parameters":
    {
        "test_image_prompt": "root@TQMa28:~",
        "role": "can-dut1"
    }
},
{
    "command": "boot_linaro_image",
    "parameters":
    {
        "test_image_prompt": "root@MBa6x:~",
        "role": "can-dut2"
    }
},
```

Testdefinition

Der Ablauf des CAN-Tests wurde bereits oben beschrieben, Listing 5.3 stellt diesen in einer LAVA-Testdefintion dar. Wie zu erkennen ist werden drei Shell Skripte aufgerufen, die jeweils eine bestimmte Aufgabe erfüllen. Manche Skripte erwarten Aufrufparameter, diese können in der Testdefinition mit Default-Werten versehen und ”Live” im Testjob überschrieben werden, wie in Anhang Anhang A Listing A.1, auf Seite 78 zu erkennen ist. Die Skripte wurden so aufgebaut, dass sie unabhängig[19] von der LAVA-Infrastruktur sind und auch ohne LAVA aufgerufen

[19] Es werden keine LAVA-spezifischen Shell-Befehle in den Skripten aufgerufen.

werden können (Listing A.2 - A.4, Anhang A.2). Die Skriptausgabe erfolgt auf der Standardausgabe in der formatierten Form:

```
<Test-Name>: <fail/pass>
```

Dies ermöglicht eine einfache Übersetzung durch den LAVA-Parser. Die in Listing 5.3 aufgeführten ”lava-sync” Aktionen dienen der Synchronisation der DUT-Gruppen und sind LAVA-spezifische Kommandos. Die Verwendung LAVA-spezifischer Kommandos innerhalb der Testdefinitionen sind zulässig, da diese nur für das LAVA-System benutzt werden. Anders ist es hingegen bei den Skripten, die innerhalb der Testdefinition aufgerufen werden. Diese sollten möglichst keine LAVA-spezifischen Kommandos enthalten, da sie sonst nur noch in Verbindung mit dem LAVA-System nutzbar wären.

Listing 5.3: CAN-Bus Testdefinition

```
1  metadata:
2      format: Lava-Test Test Definition 1.0
3      name: can
4      description: "Basic CAN-Bus test with two nodes"
5      maintainer:
6          - marco.felsch@tq-group.com
7
8      os:
9
10     scope:
11         - functional
12
13     devices:
14         - tqma6q
15         - tqma6d
16         - tqma6dl
17         - tqma6s
18         - tqma28aa
19         - tqma28ab
20 params:
21     DUT_BITRATE: 1000000
22     DUT_MSG: 0xff
23     DUT_MSG_ID: 0x10
24     DUT_MSG_LOOP: 10000
25     DUT_CAN_PORT: can0
26     DUT_LISTEN_ID: 0x05
27
28
29 run:
30     steps:
31       - lava-test-case multinode-role-output --shell lava-role
32       - "cd ptxdist/scripts/can"
33       - "./can-transceiver-init.sh -i $DUT_LISTEN_ID  \
34                                    $DUT_BITRATE $DUT_CAN_PORT"
35       - lava-sync can_ready
```

```
36        - "./can-transceiver-start.sh -i $DUT_MSG_ID -l \
37                                    $DUT_MSG_LOOP $DUT_CAN_PORT $DUT_MSG"
38        - lava-sync can_transmitted
39        - "./can-transceiver-evaluate-result.sh"
40
41  parse:
42      pattern:
43          "(?P<test_case_id>.*-*):\\s+(?P<result>(pass|fail|skip|unkown))"
```

Versuchsaufbau

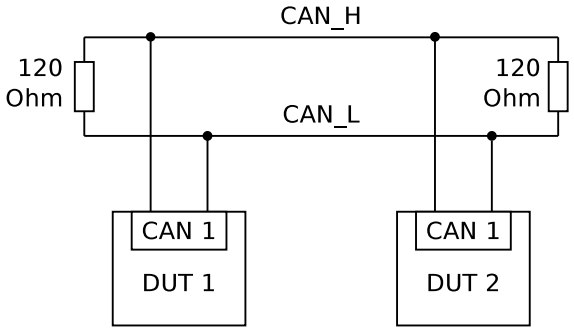


Abbildung 5.2: Versuchsaufbau CAN-Test

5.4 Versuchsauswertung

Wie bereits erwähnt übersetzt LAVA die Testjob-Datei von dem YAML-Format in ein ausführbares Shell-Skript (Anhang A.2 - Listing A.5, Seite 89), welches von der LAVA-Testausführungseinheit gestartet wird. LAVA bietet die Möglichkeit alle relevanten Testausgaben als Textdatei herunterzuladen (Anhang A.2 - Listing A.6, Seite 90). Weiterhin stellt LAVA alle Testergebnisse übersichtlich in einer Tabelle im Web-Frontend dar. Um die Arbeitsweise von LAVA einmal darzustellen wurden aus der Log-Datei die entsprechenden Teile separiert und folgend in ”Phasen” dargestellt.

Phase 1

LAVA testet ob eine serielle Verbindung zum DUT hergestellt werden kann.

Listing 5.4: Test serielle Verbindung

```
1  <LAVA_DISPATCHER>2015-07-10 01:34:40 PM INFO: Attempting to connect to
       device using: telnet localhost 2000
2  <LAVA_DISPATCHER>2015-07-10 01:34:40 PM DEBUG: expect (10): ’[’Connection
        closed by foreign host\\.’, ’Connected\\.\r’, ’Data Buffering
       Suspended\\.’, <class ’pexpect.TIMEOUT’>]’
3  Trying ::1...
4  Trying 127.0.0.1...
```

```
5  Connected to localhost.
6  Escape character is '^]'.
7
8  ser2net port 2000 device /dev/ttyS7 [115200 N81] (Debian GNU/Linux)
9
10
11
12 OSELAS(R)-Busybox ROOTFS-0010 / tqma28-0112-rc1
13
14 ptxdist-2013.12.0/2015-05-19T09:51:49+0200
15
16
17
18 TQMa28 login: <LAVA_DISPATCHER>2015-07-10 01:34:50 PM INFO: Matched <
       class 'pexpect.TIMEOUT'> which means all-good
19 <LAVA_DISPATCHER>2015-07-10 01:34:50 PM DEBUG: [ACTION-B] Multi Node test
       !
20 <LAVA_DISPATCHER>2015-07-10 01:34:50 PM DEBUG: [ACTION-B] target_group is
        (71416b28-17f8-429d-8206-2bc7b60a6c30).
21 <LAVA_DISPATCHER>2015-07-10 01:34:50 PM DEBUG: lmp modules default init
       data is []
22 <LAVA_DISPATCHER>2015-07-10 01:34:50 PM DEBUG: lmp modules final init
       data is []
```

Phase 2

Der LAVA-Dispatcher bezieht alle benötigten Daten (Kernel, DTB, rootfs) von den angegeben Quellen und bildet anschließend eine "md5" und "sha256" Checksumme.

Listing 5.5: Download aller Images

```
1  <LAVA_DISPATCHER>2015-07-10 01:34:50 PM INFO: [ACTION-B]
       deploy_linaro_kernel is started with {u'login_prompt': u'TQMa28 login
       :', u'dtb': u'http://git.tqsc.de/LAVA/Last-Builds/MBa28-images/78/
       imx28-mba28.dtb', u'username': u'root', u'nfsrootfs': u'http://git.
       tqsc.de/LAVA/Last-Builds/MBa28-images/78/root.tgz', u'kernel': u'http
       ://git.tqsc.de/LAVA/Last-Builds/MBa28-images/78/linuximage'}
2  <LAVA_DISPATCHER>2015-07-10 01:34:50 PM DEBUG: Attempting to set
       deployment data
3  <LAVA_DISPATCHER>2015-07-10 01:34:50 PM DEBUG: About to download http://
       git.tqsc.de/LAVA/Last-Builds/MBa28-images/78/linuximage to the host
4  <LAVA_DISPATCHER>2015-07-10 01:34:50 PM INFO: Downloading image: http://
       git.tqsc.de/LAVA/Last-Builds/MBa28-images/78/linuximage
5  <LAVA_DISPATCHER>2015-07-10 01:34:50 PM INFO: md5sum of downloaded
       content: cd5fceca0f73ae9e027f42183c99e995
6  <LAVA_DISPATCHER>2015-07-10 01:34:50 PM DEBUG: sha256sum of downloaded
       content: 2
       bec2a64021ac6edb99e39b99bc0e64a2f926b8f7f441fd3509f202aa901c6de
7  <LAVA_DISPATCHER>2015-07-10 01:34:50 PM DEBUG: About to download http://
       git.tqsc.de/LAVA/Last-Builds/MBa28-images/78/imx28-mba28.dtb to the
       host
```

```
8  <LAVA_DISPATCHER>2015-07-10 01:34:50 PM INFO: Downloading image: http://
       git.tqsc.de/LAVA/Last-Builds/MBa28-images/78/imx28-mba28.dtb
9  <LAVA_DISPATCHER>2015-07-10 01:34:50 PM INFO: md5sum of downloaded
       content: 3978e3da1176a7c3618b6bd11ad9a0e4
10 <LAVA_DISPATCHER>2015-07-10 01:34:50 PM DEBUG: sha256sum of downloaded
       content:
       c4fb4ec0c81f71bf3480dc7c0e1a8cd58e7f510c7b6e6a14b1c2add887aaaf03
11 <LAVA_DISPATCHER>2015-07-10 01:34:50 PM DEBUG: About to download http://
       git.tqsc.de/LAVA/Last-Builds/MBa28-images/78/root.tgz to the host
12 <LAVA_DISPATCHER>2015-07-10 01:34:50 PM INFO: Downloading image: http://
       git.tqsc.de/LAVA/Last-Builds/MBa28-images/78/root.tgz
13 <LAVA_DISPATCHER>2015-07-10 01:34:51 PM INFO: md5sum of downloaded
       content: 35d89eb5fec0055c8c0b59568856c0ae
14 <LAVA_DISPATCHER>2015-07-10 01:34:51 PM DEBUG: sha256sum of downloaded
       content:
       af6fa19d34856bdea46fbd2c4f6e9daae8a9793a32ca35217ec03294a413cced
15 <LAVA_DISPATCHER>2015-07-10 01:34:51 PM WARNING: Attempting to extract
       tarball with --strip-components=1
16 <LAVA_DISPATCHER>2015-07-10 01:34:51 PM DEBUG: Executing on host : 'u'
       nice tar --selinux --strip-components=1 -C /var/lib/lava/dispatcher/
       tmp/tmpi0OTQr/tmpxvoCOf -xaf /var/lib/lava/dispatcher/tmp/tmpi0OTQr/
       root.tgz''
17 <LAVA_DISPATCHER>2015-07-10 01:34:51 PM DEBUG: Executing on host : 'u'rm
       /var/lib/lava/dispatcher/tmp/tmpi0OTQr/root.tgz''
18 <LAVA_DISPATCHER>2015-07-10 01:34:51 PM DEBUG: Executing on host : '['sh
       ', '-c', u'file /var/lib/lava/dispatcher/tmp/tmpi0OTQr/.linuximage']'
19 <LAVA_DISPATCHER>2015-07-10 01:34:51 PM INFO: Attempting to set boot
       command as bootz
20 <LAVA_DISPATCHER>2015-07-10 01:34:51 PM DEBUG: Undefined u_load_addrs or
       z_load_addrs. Three values required!
21 <LAVA_DISPATCHER>2015-07-10 01:34:51 PM DEBUG: setting status pass
22 <LAVA_DISPATCHER>2015-07-10 01:34:51 PM DEBUG: finally status pass
23 <LAVA_DISPATCHER>2015-07-10 01:34:51 PM INFO: [ACTION-E]
       deploy_linaro_kernel is finished successfully.
```

Phase 3

Es wird die "boot_linaro_image" Aktion ausgeführt. Diese Aktion hätte auch weggelassen werden können, da jede "lava_test_shell" Aktion das DUT neu startet. Allerdings kann nur in dieser Phase der "Shell-Prompt" festgelegt werden. Dies ist nötig, da er nicht mit den "LAVA Default-Werten" übereinstimmt.

Listing 5.6: Hardreset des DUTs

```
1  <LAVA_DISPATCHER>2015-07-10 01:34:51 PM INFO: [ACTION-B]
       boot_linaro_image is started with {u'test_image_prompt': u'
       root@TQMa28:~'}
2  <LAVA_DISPATCHER>2015-07-10 01:34:51 PM INFO: Boot the test image
3  <LAVA_DISPATCHER>2015-07-10 01:34:51 PM INFO: Booting the test image.
       Attempt: 1
4  <LAVA_DISPATCHER>2015-07-10 01:34:51 PM DEBUG: expect (1): '['.+', <class
        'pexpect.EOF'>, <class 'pexpect.TIMEOUT'>]'
```

```
5  <LAVA_DISPATCHER>2015-07-10 01:34:51 PM DEBUG: expect (1): '['.+', <class
       'pexpect.EOF'>, <class 'pexpect.TIMEOUT'>]'
6  <LAVA_DISPATCHER>2015-07-10 01:34:52 PM INFO: Perform hard reset on the
       system
7  <LAVA_DISPATCHER>2015-07-10 01:34:54 PM DEBUG: expect (120): 'Hit any key
       to stop autoboot'
8  b
9
10 Hit any key to stop autoboot:  3 <LAVA_DISPATCHER>2015-07-10 01:35:07 PM
       DEBUG: send (delay_ms=0): b
```

Anschließend beginnt LAVA damit die Boot-Abfolge zu übertragen, dabei ist die Arbeitsweise von LAVA sehr gut erkennbar (das Parsen der Standardausgabe).

Listing 5.7: Übertragung der Boot-Abfolge

```
1  <LAVA_DISPATCHER>2015-07-10 01:35:07 PM INFO: Loading boot_cmds from
       device configuration
2  <LAVA_DISPATCHER>2015-07-10 01:35:07 PM DEBUG: boot_cmds(after
       preprocessing): ['setenv autoload no', 'setenv enet_clk_internal 1',
       "setenv initrd_high '0xffffffff'", "setenv fdt_high '0xffffffff'", "
       setenv kernel_addr_r '0x42000000'", "setenv fdt_addr_r '0x41000000'",
        u"setenv loadkernel 'tftp ${kernel_addr_r} tmpi0OTQr/.linuximage'",
       u"setenv loadfdt 'tftp ${fdt_addr_r} tmpi0OTQr/imx28-mba28.dtb'", u"
       setenv nfsargs 'setenv bootargs root=/dev/nfs rw nfsroot
       =192.168.1.1:/var/lib/lava/dispatcher/tmp/tmpi0OTQr/tmpxvoCOf,v3,tcp
       ip=dhcp console=ttyAPP3,115200'", "setenv bootcmd 'dhcp; setenv
       serverip 192.168.1.1; run loadkernel; run loadfdt; run nfsargs; bootz
        ${kernel_addr_r} - ${fdt_addr_r}'", 'boot']
3  <LAVA_DISPATCHER>2015-07-10 01:35:07 PM DEBUG: expect (150): '>'
4     0
5
6  TQMa28 U-Boot > <LAVA_DISPATCHER>2015-07-10 01:35:07 PM DEBUG: send (
       delay_ms=0): setenv autoload no
7  <LAVA_DISPATCHER>2015-07-10 01:35:08 PM DEBUG: send (delay_ms=0):
8
9  <LAVA_DISPATCHER>2015-07-10 01:35:08 PM DEBUG: expect (150): '>'
10 setenv autoload no
11
12 TQMa28 U-Boot > <LAVA_DISPATCHER>2015-07-10 01:35:08 PM DEBUG: send (
       delay_ms=0): setenv enet_clk_internal 1
13 <LAVA_DISPATCHER>2015-07-10 01:35:37 PM DEBUG: expect (120): 'Linux
       version'
14 boot
15
16 [    0.000000] Booting Linux on physical CPU 0x0
17 [    0.000000] Linux version 3<LAVA_DISPATCHER>2015-07-10 01:35:47 PM
       DEBUG: expect (150): 'TQMa28 login:'
18
19
20 OSELAS(R)-Busybox ROOTFS-0010 / tqma28-0112-rc1
21
```

```
22 ptxdist-2013.12.0/2015-06-04T17:42:48+0200
23
24
25
26 TQMa28 login: <LAVA_DISPATCHER>2015-07-10 01:36:17 PM DEBUG: send (
       delay_ms=0): root
27 <LAVA_DISPATCHER>2015-07-10 01:36:22 PM INFO: System is in test image now
```

Phase 4

In dieser Phase wird der CAN-Test von der Test-Ausführungseinheit aufgerufen.

Listing 5.8: CAN-Test

```
1  linaro-test [rc=0]# <LAVA_DISPATCHER>2015-07-10 01:36:29 PM DEBUG:
       Checking for vm-group host
2  <LAVA_DISPATCHER>2015-07-10 01:36:29 PM DEBUG: send (delay_ms=0): /lava-
       mba28x-tqma28aa-01/bin/lava-test-runner /lava-mba28x-tqma28aa-01
3  <LAVA_DISPATCHER>2015-07-10 01:36:32 PM DEBUG: send (delay_ms=0):
4
5  <LAVA_DISPATCHER>2015-07-10 01:36:32 PM DEBUG: expect (1200): '['<
       LAVA_TEST_RUNNER>: exiting', <class 'pexpect.EOF'>, <class 'pexpect.
       TIMEOUT'>, '<LAVA_SIGNAL_(\\S+) ([^>]+)>', '<LAVA_MULTI_NODE> <LAVA_
       (\\S+) ([^>]+)>', '<LAVA_LMP> <LAVA_(\\S+) ([^>]+)>', <class 'pexpect
       .TIMEOUT'>]'
6  /lava-mba28x-tqma28aa-01/bin/lava-test-runner /lava-mba28x-t
7
8  qma28aa-01
9  /lava-mba28x-tqma28aa-01
10 <LAVA_TEST_RUNNER>: started
11 <LAVA_TEST_RUNNER>: looking for installation work in /lava-mba28x-
       tqma28aa-01/lava-test-runner.conf-86
12 <LAVA_TEST_RUNNER>: save hardware/software context info...
13 <LAVA_TEST_RUNNER>: looking for work in /lava-mba28x-tqma28aa-01/lava-
       test-runner.conf-86
14 <LAVA_TEST_RUNNER>: running 0_can under lava-test-shell...
15 <LAVA_SIGNAL_STARTRUN can 885be734-bf15-4bc3-ac92-5a4dcb1fbd2c>
16 <LAVA_DISPATCHER>2015-07-10 01:36:34 PM DEBUG: Received signal <STARTRUN>
17 <LAVA_DISPATCHER>2015-07-10 01:36:34 PM DEBUG: send (delay_ms=0): echo
       LAVA_ACK
```

Phase 5

Abschließend erfolgt das "Einsammeln" der Testergebnisse und die Übertragung dieser an die LAVA-Master Instanz, wie bereits in Kapitel 4 beschrieben.

Listing 5.9: CAN-Test Ergebnisse

```
===
can
===

Test case                                   Result
------------------------------------------ --------
multinode-role-output                         PASS
can-init                                      PASS
can-start                                     PASS
can-eval-res                                  PASS

<LAVA_DISPATCHER>2015-07-10 01:36:47 PM DEBUG: setting status pass
<LAVA_DISPATCHER>2015-07-10 01:36:47 PM DEBUG: finally status pass
<LAVA_DISPATCHER>2015-07-10 01:36:47 PM INFO: [ACTION-E] lava_test_shell
    is finished successfully.
<LAVA_DISPATCHER>2015-07-10 01:36:47 PM DEBUG: Finalizing child process
    group with PID 3383
<LAVA_DISPATCHER>2015-07-10 01:36:48 PM INFO: Passing results bundle to
    LAVA Coordinator.
<LAVA_DISPATCHER>2015-07-10 01:36:48 PM DEBUG: sending result bundle
```

6 Zusammenfassung und Ausblick

6.1 Zusammenfassung

Einführend erfolgte eine Analyse des Ist-Zustandes des bisherigen Testverfahrens. Diese zeigte auf, dass das bisherige Testverfahren nur bedingt geeignet ist, da keine reproduzierbare Aussage über die Funktionsfähigkeit eines Software-Standes getroffen werden kann. Anschließend erfolgte in Absprache mit allen Beteiligten die Definition des Soll-Zustandes, welcher in den Tabellen 2.1 und 2.2 auf den Seiten 11 und 12 festgehalten wurde. Nachfolgend wurde ein Überblick auserwählter formaler Testmethoden erstellt, der die Testerstellung erleichtern und vereinheitlichen soll[20]. Darauf erfolgte eine Kurzvorstellung der bereits auf dem Markt vorhanden Testsysteme. Der anschießende Vergleich aller vorgestellten Systeme ist in Tabelle 3.1 auf Seite 40 festgehalten. Abschließend wurden die Systeme den Anforderungen gegenübergestellt, woraus die Tabelle 3.2 auf Seite 41 hervorging. Es hat sich herausgestellt, dass LAVA das einzige System ist, welches alle Anforderungen erfüllt. Infolgedessen wurde LAVA beispielhaft implementiert, damit sichergestellt wird, dass das System mühelos in die vorhandene Infrastruktur eingebunden werden kann[21]. Zum Schluss wurde ein Testfall für den CAN-Bus implementiert. Dieser soll die Vorgehensweise der Testerstellung demonstrieren.

Im Zuge der Analyse vorhandener Testsysteme, wurde die LTP und LTP-DDT Testsuite, in die Linux-Distribution ”PTXdist” eingepflegt. Dies erlaubt einen schnellen Einstieg in das Erstellen von Testfällen, da allein die LTP-Suite über 300 vorbereitete Testfälle beinhaltet [12].

Durch das neue Testsystem wurde eine Infrastruktur geschaffen, die über das einfache Ausführen von Shell-Skripten hinaus geht. Mit der neuen Infrastruktur ist es nun möglich Regressionstests, wie sie in Abschnitt 2.4 vorgestellt wurden, durchzuführen. Mit der Ausnahme das ein Vergleich der Testergebnisse nicht stattfindet, da das Ergebnis der Testfälle durch das aufgerufene Testfallskript bestimmt wird. Trotz alledem kann es als automatischer Regressionstest betrachtet werden, wodurch die Wirtschaftlichkeit und die Aussagekraft der Testergebnisse gesteigert wird, da automatisch nach jeder Softwareänderung ein kompletter Testdurchlauf durchgeführt wird. Damit wird weiterhin

[21] Damit möglichst alle Testfälle erfasst werden, wird ausdrücklich empfohlen einer dieser Methoden bei der Testerstellung zu verwenden. Sie können aber auch schnell zu einem großem Overhead führen, weshalb abgewogen werden muss, ob eine formale Beschreibung sinnvoll ist.

[21] Während dieser Phase wurde eine Dokumentation erstellt, die eine spätere reale Implementierung vereinfachen soll.

sichergestellt, dass nicht nur die Komponenten getestet werden, die von der Änderung direkt betroffen sind.

Aufgrund der begrenzten Zeit konnte noch kein umfangreiches Testfall-Repository aufgebaut werden. Ebenso ist die manuelle Testjob-Triggerung noch nicht möglich. Ein Verbesserungsvorschlag für die reale Implementierung wäre, dass der Jenkins-Server und die LAVA-Server Instanz auf dem gleichen Computer ausgeführt werden. Dies hat den Vorteil, dass der Testjob direkt übermittelt werden kann und nicht erst über das Netzwerk geschickt werden muss.

Vorteile der neuen Testinfrastruktur:

- Eine einheitliche und immer gleichbleibende Testdurchführung.
- Es wird automatisch nach jedem erstellten BSP eine Testdurchführung gestartet.
- Testergebnisse werden für spätere Analysen gespeichert und können anschließend exportiert werden. Dies eröffnet die Möglichkeit einer automatischen Testdokumentationserstellung.
- Die Testergebnisse werden übersichtlich in einem Web-Frontend dargestellt. Durch dieses ist es ebenfalls möglich, die Ergebnisse nach eigenen Bedürfnissen zu filtern.
- Wurde das Prinzip von LAVA einmal verstanden, ist das Hinzufügen neuer DUTs sehr einfach.
- Für ad hoc Tests können die Skripte benutzt werden, welche sonst in den Testfällen aufgerufen werden.
- Durch die Möglichkeit eigene Erweiterungen für LAVA zu schreiben, kann das System genau an die eigenen Bedürfnisse und Wünsche angepasst werden.
- Linaro bietet für das LAVA-Framework eine professionelle Wartung an.

6.2 Ausblick

6.2.1 Plattformunabhängigkeit

Mit LAVA wurde ein Testsystem eingeführt, dass vor allem für ARM-SoC's in Verbindung mit Linux geeignet ist. Das System basiert darauf die Standardausgabe zu "parsen", deshalb sollte es grundsätzlich möglich sein das System auch für Mikrocontroller einzusetzen. Dafür müssen folgende Bedingungen für den Mikrocontroller gegeben sein:

1. Er muss über mindestens eine RS232-Schnittstelle verfügen.
2. Auf dem Mikrocontroller läuft eine minimale Software, welches per RS232 Befehle entgegennimmt und entsprechend ausführt.

Die erste Bedingung kann als formal betrachtet werden, da heutige Mikrocontroller fast alle über eine RS232-Schnittstelle verfügen. Schwieriger ist es die zweite Bedingung zu erfüllen. Hierfür können die folgenden Lösungsmöglichkeiten in Betracht gezogen werden:

- Entwicklung einer eigene Applikation, die dies erfüllt. Diese Lösung sollte auf einem Betriebssystem basieren, damit die Hardware abstrahiert werden kann und die Wiederverwendbarkeit erhöht wird.
- Die Verwendung eines Bootloaders, womit Skripte ausgeführt werden können.
- Verwendung einer proprietären Lösung, wobei dies die Gefahr birgt, dass diese Lösung möglicherweise nicht in LAVA integriert werden kann.

Eine solche Lösung bietet die Firma KOZIO mit ihrer Software VTOS-Scan [21]. Das Softwarepaket umfasst ein Host-Tool, welches für die Interaktion mit dem DUT verantwortlich ist und ein Target-Tool, welches die Befehle des Hosts entgegennimmt und auf dem DUT ausgeführt wird. VTOS-Scan kann somit als eigenständiges Tool betrachtet werden. Bevor die Testdurchführung begonnen werden kann, muss das Target-Tool mittels JTAG-Programmer in den Onboard-Speicher des DUTs programmiert werden. Mit VTOS-Scan können beispielsweise GPIO, SPI-Bus und der I2C-Bus getestet werden [22]. Die Testergebnisse können anschließend exportiert werden.

6.2.2 LAVA Erweiterungen

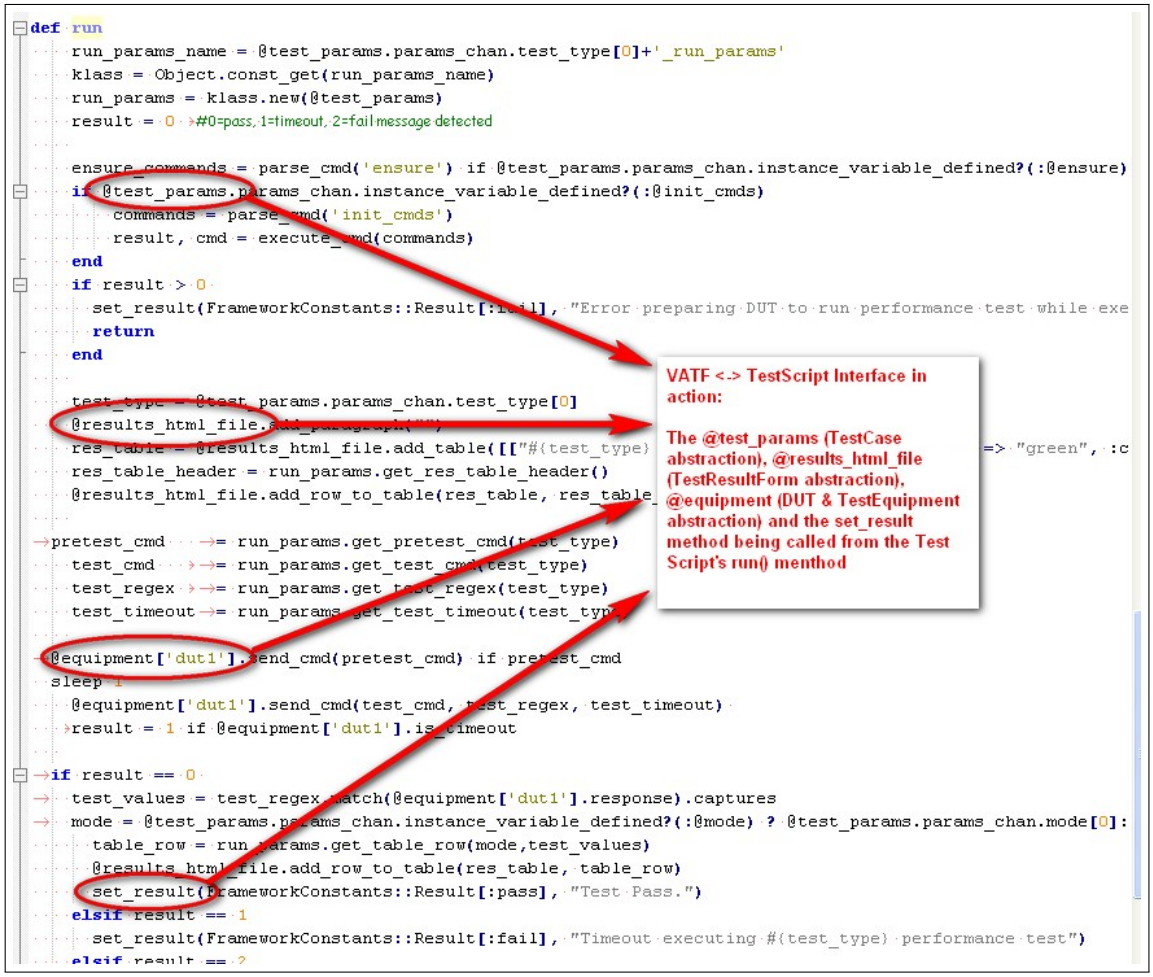
Wie bereits erwähnt, kann LAVA mittels Erweiterungen nach den eigenen Wünschen angepasst werden. Eine Idee ist die Entwicklung einer Erweiterung, die es ermöglicht aus den Testergebnissen fertige Testdokumentationen zu erstellen. Dies vereinfacht den BSP-Release Zyklus um einen weiteren Schritt und es wird eine einheitliche Dokumentationsstruktur eingeführt, die über alle Projekte hinweg gleich bleibt.

6.2.3 Testhardware-Gegenstelle

Das bisherige System setzt darauf auf, dass alle Zielsysteme (Targets), die an den Worker-Instanzen angeschlossen sind, zugleich DUTs sind. Dies ist für einfache Boot-Tests vollkommen ausreichend, bei komplexeren Tests kann diese Tatsache allerdings zu Problemen führen. Soll beispielsweise ein Bus-Test, wie in Kapitel 5 beschrieben, durchgeführt werden, ist es schwieriger auftretende Fehler zu lokalisieren. Da die Fehlerquelle sowohl auf einem, oder auch auf beiden DUTs existieren kann. Besser ist es eine bekannte HW-Gegenstelle zu Entwickeln, die über alle zu testenden Schnittstellen verfügt. Diese müssen natürlich im Voraus getestet wurden sein, damit eine korrekte Funktionsweise sichergestellt werden kann.

Anhang A: Quellcode

A.1 Opentest - VATF Beispiel Testskript



A.2 LAVA

Listing A.1: Multi-Node CAN-Bus Testjob

```
{
    "actions": [
    {
        "command": "deploy_linaro_kernel",
        "parameters":
        {
            "dtb":
        "http://git.tqsc.de/LAVA/Last-Builds/MBa28-images/78/imx28-mba28.dtb",
            "kernel":
        "http://git.tqsc.de/LAVA/Last-Builds/MBa28-images/78/linuximage",
            "nfsrootfs":
        "http://git.tqsc.de/LAVA/Last-Builds/MBa28-images/78/root.tgz",
            "login_prompt": "TQMa28 login:",
            "username": "root",
            "role": "can-dut1"
        }
    },
    {
        "command": "deploy_linaro_kernel",
        "parameters":
        {
            "dtb":
        "file:///home/lavaadmin/images/imx6dl-mba6x.dtb-tqma6x-3-16",
            "kernel": "file:///home/lavaadmin/images/linuximage",
            "nfsrootfs": "file:///home/lavaadmin/images/root.tgz",
            "login_prompt": "MBa6x login:",
            "username": "root",
            "role": "can-dut2"
        }
    },
    {
        "command": "boot_linaro_image",
        "parameters":
        {
            "test_image_prompt": "root@TQMa28:~",
            "role": "can-dut1"
        }
    },
    {
        "command": "boot_linaro_image",
        "parameters":
        {
            "test_image_prompt": "root@MBa6x:~",
            "role": "can-dut2"
        }
    },
    {
        "command": "lava_test_shell",
        "parameters":
```

```
50          {
51              "testdef_repos": [
52              {
53                  "git-repo": "git://192.168.169.89/test-definitions",
54                  "testdef": "ptxdist/can.yaml",
55                  "parameters":
56                  {
57                      "DUT_BITRATE": "500000",
58                      "DUT_MSG": "0xbb",
59                      "DUT_MSG_ID": "0x05",
60                      "DUT_MSG_LOOP": "10",
61                      "DUT_CAN_PORT": "can1",
62                      "DUT_LISTEN_ID": "0x10"
63                  }
64              }],
65              "role": "can-dut1"
66          }
67      },
68      {
69          "command": "lava_test_shell",
70          "parameters":
71          {
72              "testdef_repos": [
73              {
74                  "git-repo": "git://192.168.169.89/test-definitions",
75                  "testdef": "ptxdist/can.yaml",
76                  "parameters":
77                  {
78                      "DUT_BITRATE": "500000",
79                      "DUT_MSG": "0xaa",
80                      "DUT_MSG_ID": "0x10",
81                      "DUT_MSG_LOOP": "10",
82                      "DUT_CAN_PORT": "can1",
83                      "DUT_LISTEN_ID": "0x05"
84                  }
85              }],
86              "role": "can-dut2"
87          }
88      },
89      {
90          "command": "submit_results",
91          "parameters":
92          {
93              "server": "http://tqsc-lava-server-1.tqsc.de/RPC2/",
94              "stream": "/public/team/ci/BSP/MBa28/"
95          }
96      }],
97      "device_group": [
98      {
99          "role": "can-dut1",
100         "count": 1,
101         "device_type": "tqma28aa"
```

```
    },
    {
        "role": "can-dut2",
        "count": 1,
        "device_type": "tqma6s"
    }],
    "logging_level": "DEBUG",
    "job_name": "manual-can-test",
    "timeout": 900
}
```

Listing A.2: CAN-Init Shell Skript

```
#!/bin/sh
#
# Set up a can inerface to send and receive data


# TODO(mfelsch@tqsc.de): remove printdbg()



readonly EXIT_FAILURE=2 # Ash failure exit-status
readonly EXIT_SUCCESS=0
readonly PWD=$(pwd)

baud=""          # can interface bitrate
iface=""         # can interface
id="3"           # message id to listen to

err() {
    echo "[$(date +'%Y-%m-%dT%H:%M:%S%z')]: $@" >&2
}


help() {
    echo "Usage: "$(basename "$0")" [OPTIONS] \
                <BAUDRATE (Hz)> <CAN_INTERFACE>"
    echo "Set up the CAN interface to transmit and receive can messages"
    echo ""
    echo "Options:"
    echo "========"
    echo "     -h          this message"
    echo "     -i          message id to listen to (default = 3)"
    echo ""
}

#printdbg() {
#    echo "baud:       ${baud}"
#    echo "iface:      ${iface}"
#    echo "listen-id:  ${id}"
#}

parse_option() {

    # avoid problems with multiple calling
    local h i opt OPTIND

    while getopts "hi:" opt
    do
        case "${opt}" in
            h)
                help
                exit ${EXIT_FAILURE}
```

```
52                  echo "help"
53                  ;;
54              i)
55                  id=${OPTARG}
56                  ;;
57              ?)
58                  help
59                  err "Unexpect option"
60                  exit ${EXIT_FAILURE}
61                  ;;
62          esac
63      done
64      return ${OPTIND}
65  }
66
67  parse_arguments() {
68
69      # avoid wrong arguments
70      if [ ${#} -lt 1 ]
71      then
72          help
73          err "To less arguments: ${#}"
74          exit ${EXIT_FAILURE}
75      elif [ ${#} -gt 2 ]
76      then
77          help
78          err "To much arguments: ${#}"
79          exit ${EXIT_FAILURE}
80      fi
81
82      # set baudrate
83      if [ "${1}" = "" ]
84      then
85          help
86          err "The Baudrate must be set greater than zero"
87          exit ${EXIT_FAILURE}
88      else
89          baud="${1}"
90      fi
91
92      # set can interface
93      if [ "${2}" != "" ]
94      then
95          for ifaces in $(ifconfig -a | grep can | cut -f1 -d' ')
96          do
97              if [ "${ifaces}" = "${2}" ]
98              then
99                  iface="${ifaces}"
100             fi
101         done
102         if [ "${iface}" = "" ]
103         then
```

```
104             help
105             err "wrong interface ${2}"
106             exit ${EXIT_FAILURE}
107         fi
108     else
109         help
110         err "wrong interface"
111         exit ${EXIT_FAILURE}
112     fi
113 
114 #   printdbg
115 }
116 
117 can_init() {
118     canconfig ${iface} bitrate ${baud}
119     canconfig ${iface} start
120     candump ${iface} -d --filter=${id}:0xffff
121 }
122 
123 main() {
124     local argstart
125 
126     parse_option "${@}"
127     argstart=${?}
128     shift $((argstart-1))   # align the program arguments
129     parse_arguments "${@}"
130     can_init
131     echo "pass"
132     exit ${EXIT_SUCCESS}
133 }
134 
135 main "$@"
```

Listing A.3: CAN-Start Shell Skript

```
#!/bin/sh
#
# Start transmission on can bus


# TODO(mfelsch@tqsc.de): remove printdbg()


readonly EXIT_FAILURE=2 # Ash failure exit-status
readonly EXIT_SUCCESS=0
readonly PWD=$(pwd)

iface=""         # can interface
msg_loop="100"   # message counter
id="2"           # id for message to send
msg=""           # message to send

err() {
    echo "[$(date +'%Y-%m-%dT%H:%M:%S%z')]: $@" >&2
}


help() {
    echo "Usage: "$(basename "$0")" [OPTIONS] <CAN_INTERFACE> <CAN_MSG>"
    echo "<CAN_MSG> can consist of up to 8 bytes given separated list"
    echo ""
    echo "Options:"
    echo "========"
    echo "     -h       this messag"
    echo "     -i       message id (default = 2)"
    echo "     -l COUNT send message COUNT times (default: COUNT = 100)"
    echo ""
}

#printdbg() {
#    echo "iface:      ${iface}"
#    echo "loop:       ${msg_loop}"
#    echo "message:    ${msg}"
#    echo "message-id: ${id}"
#}

parse_option() {

    # avoid problems with multiple calling
    local h i l opt OPTIND

    while getopts "f:hi:l:" opt
    do
        case "${opt}" in
            h)
                help
```

```
52                  exit ${EXIT_FAILURE}
53                  ;;
54              i)
55                  id="${OPTARG}"
56                  ;;
57              l)
58                  msg_loop="${OPTARG}"
59                  ;;
60              ?)
61                  help
62                  err "Unexpect option"
63                  exit ${EXIT_FAILURE}
64                  ;;
65          esac
66      done
67      return ${OPTIND}
68  }
69
70  parse_arguments() {
71
72      # set can interface
73      if [ "${1}" != "" ]
74      then
75          for ifaces in $(ifconfig -a | grep can | cut -f1 -d' ')
76          do
77              if [ "${ifaces}" = "${1}" ]
78              then
79                  iface="${ifaces}"
80              fi
81          done
82          if [ "${iface}" = "" ]
83          then
84              help
85              err "wrong interface ${1}"
86              exit ${EXIT_FAILURE}
87          fi
88      else
89          help
90          err "wrong interface"
91          exit ${EXIT_FAILURE}
92      fi
93
94      # set message. error if message greater than 8 or less than 1
95      if [ ${#} -gt 9 ] || [ ${#} -lt 2 ]
96      then
97          help
98          err "wrong message"
99          exit ${EXIT_FAILUER}
100     else
101         while [ ${#} -gt 1 ]
102         do
103             msg="${msg} ${2}"
```

```
104             shift 1
105         done
106     fi
107
108 #    printdbg
109 }
110
111 can_start() {
112     cansend ${iface} -i ${id} --loop=${msg_loop} ${msg}
113 }
114
115 main() {
116     parse_option "$@"
117     argstart=${?}
118     shift $((argstart-1))
119     parse_arguments "${@}"
120     can_start
121     echo "pass"
122     exit ${EXIT_SUCCESS}
123 }
124
125 main "$@"
```

Listing A.4: CAN-Auswertung Shell Skript

```
#!/bin/sh
#
# Evaluate the can transceiver test

readonly EXIT_FAILURE=2 # Ash failure exit-status
readonly EXIT_SUCCESS=0
readonly PWD=$(pwd)

err() {
    echo "[$(date +'%Y-%m-%dT%H:%M:%S%z')]: $@" >&2
}


help() {
    echo "Usage: "$(basename "$0")" [OPTIONS]"
    echo "Read the /proc/net/can/stats to evaluate the resaults"
    echo ""
    echo "Options:"
    echo "========"
    echo "     -h          this message"
    echo ""
}

parse_option() {

    # avoid problems with multiple calling
    local h opt OPTIND

    while getopts "h" opt
    do
        case "${opt}" in
            h)
                help
                exit ${EXIT_FAILURE}
                echo "help"
                ;;
            ?)
                help
                err "Unexpect option"
                exit ${EXIT_FAILURE}
                ;;
        esac
    done
    return ${OPTIND}
}

main() {
    parse_option "${@}"

    rx_frames=$(cat /proc/net/can/stats \
                                        | grep RXF \
```

```
52                                          | sed 's/^ *//g' \
53                                          | cut -f1 -d' ')
54      tx_frames=$(cat /proc/net/can/stats \
55                                          | grep RXMF \
56                                          | sed 's/^ *//g' \
57                                          | cut -f1 -d' ')
58
59      if [ ${rx_frames} -ne ${tx_frames} ]
60      then
61          echo "fail"
62          exit ${EXIT_FAILURE}
63      else
64                  echo $(cat /proc/net/can/stats)
65          echo "pass"
66          exit ${EXIT_SUCCESS}
67      fi
68  }
69
70  main "$@"
```

Listing A.5: ”Kompiliertes” Shell-Skript

```
###default parameters from yaml###
DUT_LISTEN_ID='5'
DUT_MSG_LOOP='10000'
DUT_MSG='255'
DUT_MSG_ID='16'
DUT_BITRATE='1000000'
DUT_CAN_PORT='can0'
######
###test parameters from json###
DUT_LISTEN_ID='0x10'
DUT_MSG_LOOP='10'
DUT_MSG_ID='0x05'
DUT_BITRATE='500000'
DUT_CAN_PORT='can1'
DUT_MSG='0xbb'
######
set -e
export TESTRUN_ID=can
cd /lava-mba28x-tqma28aa-01/tests/0_can
UUID=‘cat uuid‘
echo "<LAVA_SIGNAL_STARTRUN $TESTRUN_ID $UUID>"
#wait for an ack from the dispatcher
read
lava-test-case multinode-role-output --shell lava-role
cd ptxdist/scripts/can
./can-transceiver-init.sh -i $DUT_LISTEN_ID $DUT_BITRATE $DUT_CAN_PORT
lava-sync can_ready
./can-transceiver-start.sh -i $DUT_MSG_ID -l $DUT_MSG_LOOP \
                                            $DUT_CAN_PORT $DUT_MSG
lava-sync can_transmitted
./can-transceiver-evaluate-result.sh
echo "<LAVA_SIGNAL_ENDRUN $TESTRUN_ID $UUID>"
#wait for an ack from the dispatcher
read
```

Listing A.6: CAN-Test Ergebnis Log

```
<LAVA_SIGNAL_STARTRUN can 885be734-bf15-4bc3-ac92-5a4dcb1fbd2c>
<LAVA_SIGNAL_STARTTC multinode-role-output>
can-dut1<LAVA_SIGNAL_ENDTC multinode-role-output>
<LAVA_SIGNAL_TESTCASE TEST_CASE_ID=multinode-role-output RESULT=pass>
can1 bitrate: 500000, sample-point: 0.875
can1 state: ERROR-ACTIVE
can-init: pass
<LAVA_SYNC_DEBUG  preparing Thu Jan  1 00:01:30 UTC 1970>
<LAVA_SYNC_DEBUG  started Thu Jan  1 00:01:30 UTC 1970>
<LAVA_MULTI_NODE> <LAVA_SYNC can_ready>
<LAVA_SYNC_DEBUG  finished Thu Jan  1 00:01:31 UTC 1970>
<LAVA_SYNC_DEBUG  finished Thu Jan  1 00:01:31 UTC 1970>
<LAVA_SYNC_DEBUG  starting to wait Thu Jan  1 00:01:31 UTC 1970>
<LAVA_SYNC_DEBUG  save message to /tmp/lava_multi_node_cache.txt Thu Jan  1
    00:01:35 UTC 1970>
<LAVA_SYNC_DEBUG  waiting over Thu Jan  1 00:01:35 UTC 1970>
interface = can1, family = 29, type = 3, proto = 1
can-start: pass
<LAVA_SYNC_DEBUG  preparing Thu Jan  1 00:01:35 UTC 1970>
<LAVA_SYNC_DEBUG  started Thu Jan  1 00:01:35 UTC 1970>
<LAVA_MULTI_NODE> <LAVA_SYNC can_transmitted>
<LAVA_SYNC_DEBUG  finished Thu Jan  1 00:01:36 UTC 1970>
<LAVA_SYNC_DEBUG  finished Thu Jan  1 00:01:36 UTC 1970>
<LAVA_SYNC_DEBUG  starting to wait Thu Jan  1 00:01:36 UTC 1970>
<LAVA_SYNC_DEBUG  save message to /tmp/lava_multi_node_cache.txt Thu Jan  1
    00:01:40 UTC 1970>
<LAVA_SYNC_DEBUG  waiting over Thu Jan  1 00:01:40 UTC 1970>
10 transmitted frames (TXF) 20 received frames (RXF) 20 matched frames (RXMF)
    100 % total match ratio (RXMR) 0 frames/s total tx rate (TXR) 0 frames/s
    total rx rate (RXR) 100 % current match ratio (CRXMR) 0 frames/s current
    tx rate (CTXR) 0 frames/s current rx rate (CRXR) 100 % max match ratio (
    MRXMR) 10 frames/s max tx rate (MTXR) 10 frames/s max rx rate (MRXR) 1
    current receive list entries (CRCV) 2 maximum receive list entries (MRCV)
can-eval-res: pass
<LAVA_SIGNAL_ENDRUN can 885be734-bf15-4bc3-ac92-5a4dcb1fbd2c>
```

Anhang B: Dokumentationen

B.1 Auszug Test-Dokumentation

Software-Test-Dokument TQMa6x BSP

4.6.6 I2C

Der Test unter U-Boot wurde mit TQMa6S durchgeführt. Da Kernel und RootFS für alle Module identisch sind, wird der Test nicht mit anderen Modultypen wiederholt.

Tabelle 29: I2C – Status

FT/BM	Unterpunkt	Bearbeiter	Ergebnis
FT	Erkennung I2C Devices U-Boot	Niebel	OK
FT	Funktionstest	Niebel	OK

4.6.6.1 Erkennung I2C Devices U-Boot

Tabelle 30: I2C - Funktionstest

Testcase	● I2C3 (i2c dev 0) Auslesen der Device-Adressen im U-Boot
	`TQMa6x[MBa6x] U-Boot > i2c dev 0`
	`Setting bus to 0`
	`TQMa6x[MBa6x] U-Boot > i2c probe`
	`Valid chip addresses: 08 48 49 50 57 68`
	`TQMa6x[MBa6x] U-Boot >`

Erwartung	Adressen 08, 48, 49, 50, 57 und 68 vorhanden
Ergebnis	OK

Testcase	● I2C1 (i2c dev 1) Auslesen der Device-Adressen im U-Boot
	`TQMa6x[MBa6x] U-Boot > i2c dev 1`
	`Setting bus to 1`
	`TQMa6x[MBa6x] U-Boot > i2c probe`
	`Valid chip addresses: 18`

Erwartung	Adresse 18 vorhanden
Ergebnis	OK

4.6.6.2 Funktionstest

Der Funktionstest wird durch folgende Testfälle für Devices an den I2C Bussen implizit abgedeckt:

- EEPROM
- **RTC**
- Audio
- Temperatursensoren

4.6.7 RS232

Der Funktionstest für ttymxc1 wird implizit durch andere Tests abgedeckt. Ein Test der Modemleitungen findet dabei nicht statt. Andere Schnittstellen werden nicht getestet.

Tabelle 27: RS232 – Status

FT/BM	Unterpunkt	Bearbeiter	Ergebnis
FT	Vorhandensein Devices Linux	Niebel	OK
FT	**Funktionstest U-Boot**	Niebel	OK
FT	Funktionstest Linux	Niebel	OK

4.6.7.1 Vorhandensein Devices Linux

Test erfolgte mit TQMa6Q. Andere Module nutzen gleichen Kernel und gleiches RootFS.

Tabelle 31 Vorhandensein Devices

Testcase	Vorhandene Device nodes für serielle Schnittstellen abfragen
	`root@MBa6x:/ ls -al /dev/ttymxc*`
	`crw-------  1 root  root  207,  17 Jan  9 18:05 /dev/ttymxc1`
	`crw-------  1 root  root  207,  18 Jan  9 17:46 /dev/ttymxc2`
	`crw-------  1 root  root  207,  19 Jan  9 17:46 /dev/ttymxc3`
	`crw-------  1 root  root  207,  20 Jan  9 17:46 /dev/ttymxc4`

Erwartung	ttymxc1 … ttymxc4 vorhanden
Ergebnis	OK

4.6.7.2 Funktionstest U-Boot

Tabelle 32 Funktionstest RS232-Boot

Testcase	Bedienung U-Boot über Terminal
Erwartung	keine Fehler
Ergebnis	OK

4.6.7.3 Funktionstest Linux

Tabelle 33 Funktionstest RS232 Linux

Testcase	Bedienung Kernel über Terminal	
Erwartung	keine Fehler	
Ergebnis	OK	

Literaturverzeichnis

[1] Prof. Dr. U. Schneider. Vorlesung Betriebssysteme; Thema: Architekturen, 2013.

[2] Peter Liggesmeyer. *Software-Qualität: Testen, Analysieren und Verifizieren von Software*. Spektrum Akademischer Verlag, 8 2002.

[3] Opentest. http://arago-project.org/wiki/index.php/Opentest, 2015. [Online; Stand 12. Mai 2015].

[4] Autotest. https://autotest.github.io/, 2014. [Online; Stand 12. Mai 2015].

[5] Linaro Automated Validation Architecture (LAVA). https://validation.linaro.org/, 2015. [Online; Stand 12. Mai 2015].

[6] Neil Williams, Remi' Duraffort. LAVA Dispatcher refactoring – Learning from the old Preparing for the new, 2015. [Online; Stand 02. Juli 2015].

[7] TQ Group GmbH. http://www.tq-group.com, 2015. [Online; Stand 21. Juli 2015].

[8] Device-Tree. http://www.devicetree.org, 2015. [Online; Stand 21. Juli 2015].

[9] Linus Torvalds. Linux Kernel Version: mainline: 4.2-rc3 - https://www.kernel.org, 2015. [Online; Stand 21. Juli 2015].

[10] Git. Free and Open-Source Distributed Version Control System - https://www.git-scm.com, 2015. [Online; Stand 21. Juli 2015].

[11] Jenkins CI. An extensible open source continuous integrations server - https://jenkins-ci.org/, 2015. [Online; Stand 21. Juli 2015].

[12] Linux Test Project (LTP). http://linux-test-project.github.io, 2015. [Online; Stand 12. Mai 2015].

[13] Linux Test Project Device Driver Test (LTP-DDT). http://processors.wiki.ti.com/index.php/LTP_DDT, 2015. [Online; Stand 12. Mai 2015].

[14] STAF. Software Testing Automation Framework – http://staf.sourceforge.net, 2015. [Online; Stand 16. Juni 2015].

[15] Testlink. Open Source Test Management – http://testlink.org, 2015. [Online; Stand 16. Juni 2015].

[16] John Admanski, Steve Howard. Autotest - Testing the Untestable, 2009.

[17] Python Software Foundation. https://www.python.org, 2015. [Online; Stand 26. Juli 2015].

[18] XML-RPC.Com. http://xmlrpc.scripting.com, 2011. [Online; Stand 12. Mai 2015].

[19] IETF - Network Working Group, K. Sollins. TFTP Protocol Rev.2 https://www.ietf.org/rfc/rfc1350.txt, 1992. [Online; Stand 26. Julie 2015].

[20] Robert Bosch GmbH. CAN Specification Version 2.0, 1991. [Online; Stand 08. Juli 2015].

[21] Kozio, Inc. Verification and Test OS (VTOS) Scan - http://www.kozio.com/vtos-scan, 2015. [Online; Stand 28. Juli 2015].

[22] Kozio, Inc. Datenblatt VTOS Scan - http://www.kozio.com/datasheets/Kozio_VTOS_Scan_DS.pdf, 2015. [Online; Stand 13. Juni 2015].

Erklärung

Hiermit erkläre ich, dass ich meine Arbeit selbstständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und die Arbeit noch nicht anderweitig für Prüfungszwecke vorgelegt habe.

Stellen, die wörtlich oder sinngemäß aus Quellen entnommen wurden, sind als solche kenntlich gemacht.

Mittweida, 29. Juli 2015