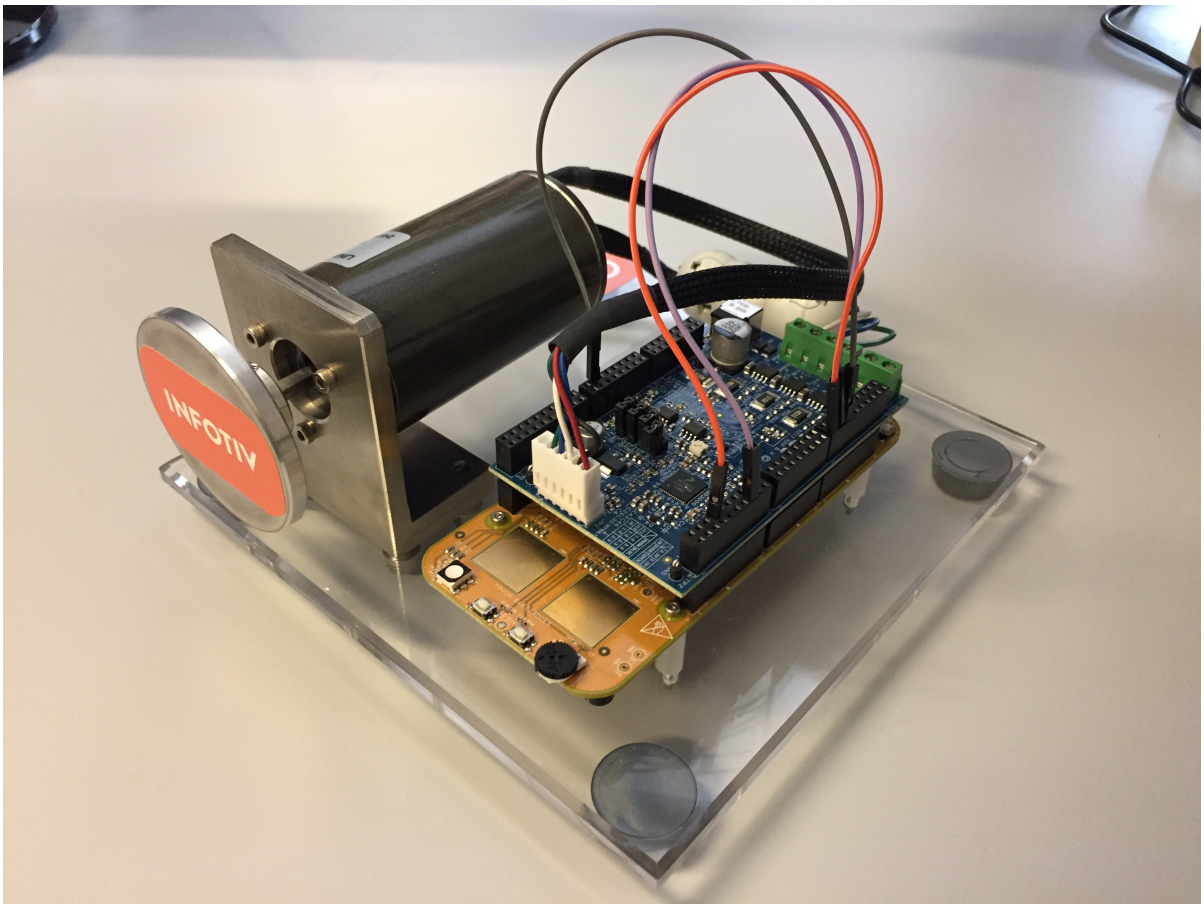
CHALMERS



Testautomatisering för motorstyrenhet

Examensarbete i Data- och Informationsteknik

MATTIAS GUSTAFSSON
GUSTAF LINDQVIST

Institutionen för data- och informationsteknik
CHALMERS TEKNISKA HÖGSKOLA
GÖTEBORGS UNIVERSITET
Göteborg, Sverige 2019

EXAMENSARBETE

Testautomatisering för motorstyrenhet

MATTIAS GUSTAFSSON
GUSTAF LINDQVIST

Institutionen för data- och informationsteknik

CHALMERS TEKNISKA HÖGSKOLA
GÖTEBORGS UNIVERSITET

Göteborg, Sverige 2019

Testautomatisering för motorstyrenhet
MATTIAS GUSTAFSSON
GUSTAF LINDQVIST

Examinator: Jonas Duregård

Institutionen för data- och informationsteknik
Chalmers tekniska högskola
Göteborgs universitet
SE-412 96 Göteborg
Sverige
Telefon: +46 (0)31-772 1000

Institutionen för data- och informationsteknik
Göteborg, Sverige 2019

Testautomatisering för motorstyrenhet

MATTIAS GUSTAFSSON
GUSTAF LINDQVIST
Institutionen för data- och informationsteknik, Chalmers tekniska högskola

Examensarbete

Sammanfattning

Avsikten med detta projekt var att ta fram en automatisk testkedja, och att utveckla funktionalitet, för en styrenhet till en elmotor. Syftet med kedjan är att göra det möjligt att starta tester från datorer som inte är anslutna till testutrustningen och att spara tid genom att automatisera vissa testmoment. Arbetet har utförts i samarbete med konsultbolaget Infotiv AB och all utveckling har skett på deras kontor i Göteborg.

Projektet har resulterat i att utveckling av ny funktionalitet för motorstyrenheten nu kan utföras på ett mer effektivt sätt. Detta eftersom alla utvecklare som arbetar med den nu kan starta tester som kommer köras i en dedikerad testmiljö från sina personliga datorer. Denna testautomatisering har utvecklats med hjälp av automationsservern Jenkins. Testerna har implementerats i Robot Framework som är ett ramverk för acceptanstestning, vilket innebär den typen av testning som verifierar att programvaran möter kundens krav.

Nyckelord: Testautomatisering, Jenkins, Robot Framework, Acceptanstestning

Abstract

The intent of this project was to produce an automated testchain, and to develop functionality, for a motor control unit for an electric motor. The purpose of the testchain is to make it possible to run tests from computers that are not connected to the testing equipment and to save time by automating some testing procedures. The work has been done in collaboration with the consulting company Infotiv AB and all development was performed at their offices in Gothenburg.

The project has resulted in that developing new functionality for the motor control unit can now be done more efficiently. The reason for this is that all developers working on the unit can now start tests that will run in a dedicated testing environment, from their own personal computers. This test automation has been developed with the help of the automation server Jenkins. The tests have been implemented in Robot Framework which is a framework for acceptance testing, which means the type of testing that verifies that the software meets the customer specifications.

Keywords: Test automation, Jenkins, Robot Framework, Acceptance Testing

Förord

Examensarbetet har utförts som ett avslutande moment för utbildningen Datateknik 180 hp, högskoleingenjör, på Chalmers tekniska högskola i Göteborg. Arbetets omfattning är 15 hp och det har utförts på halvtid under 20 veckor på vårterminen 2019, vid institutionen för Data- och Informationsteknik. Projektet möjliggjordes under HARM-mässan på Lindholmen Science Park där gruppen fick kontakt med Magnus de Canésie, konsultchef på Infotiv AB. Efter en tid besökte gruppen företagets kontor för ett möte, där ett antal förslag på examensarbeten presenterades och detta arbete valdes ut.

Vi vill tacka Infotiv AB för att vi har fått möjligheten att arbeta med detta examensarbete. Vi vill speciellt tacka Magnus de Canésie som gjorde examensarbetet möjligt, Tommy Jansson för handledning och Carl-Johan Tiger för god teknisk rådgivning inom projektet.

Ett stort tack går även till vår handledare på Chalmers, Sandro Stucki, som kommit med många goda råd under projektets gång och varit till stor hjälp vid utformningen av rapporten.

Mattias Gustafsson och Gustaf Lindqvist, Göteborg, Juni 2019

Innehåll

1 Inledning

Denna rapport inleds med en genomgång av projektets bakgrund och syfte, för att därefter gå igenom relevanta tekniker och tekniska begrepp följt av en beskrivning av projektets arbetsmetod. Resterande kapitel tar upp vad som konstruerats, vilka resultat denna konstruktion lett till och en diskussion kring dessa. Rapporten avslutas med ett slutsatskapitel.

1.1 Bakgrund

Projektet genomfördes i samarbete med konsultföretaget Infotiv. Testavdelningen på företaget, Infotiv Test & Development (ITD), jobbar bland annat med att utveckla och leverera testsystem till fordonsindustrin. Avdelningen ville ta fram ett enklare modellsystem som bygger på samma principer som de verkliga produkterna. Avsikten var att denna modell skulle kunna användas för att demonstrera för kunder hur testsystemen som utvecklas på företaget fungerar.

Modellen är en vidareutveckling av en existerande testplattform. Denna använde sig innan projektets start av två kretskort som testobjekt men dessa ansågs inte vara lämpliga att utveckla ny funktionalitet för och dessutom saknade de relevans för arbetsmarknaden i Göteborg. Dessa byttes därför ut mot en styrenhet för en elmotor.

Testplattformen kommer fortsätta utvecklas av anställda på Infotiv efter projektets slut.

1.2 Syfte

Uppdraget gick ut på att framställa en helt automatisk testkedja för ett testobjekt. Detta för att göra det snabbare och enklare för programmerare att testa sin kod och för att göra testerna mer konsekventa då de alltid körs i samma miljö, alltså samma operativsystem och samma version av programmen som används. En annan del av uppdraget var att skriva kod till styrenheten för att göra det möjligt att styra elmotorn som denna är kopplad till. Syftet med testkedjan var att kunna använda den för att köra relevanta tester för styrenheten.

I slutet av projektet förväntades testkedjan vara uppsatt samt kod för styrning av elmotorn vara klar. Meningen var att så fort ny kod skickas upp till Git-servern skulle den genomgå en rad tester. Kedjan var tänkt att bestå av fyra delar. Den första var att kunna kompilera den kod som precis laddats upp. Efter det skulle enhetstester för koden köras. Därefter skulle koden laddas upp till testobjektet. Till sist skulle en testsvit baserad på Robot Framework köras. Om något steg i kedjan misslyckades så skulle inte nästkommande steg köras. Det skulle lätt kunna upptäckas var i kedjan det uppstår fel. Både loggfiler och ett grafiskt gränssnitt i Jenkins skulle visa informationen. Den sista delen i kedjan, testsviten i Robot Framework, var tänkt att innehålla tester för funktionalitet så som att till exempel sätta ett specifikt varvtal till elmotorn eller tända lysdioderna med en viss färg. Det skulle vara enkelt att se vilka av testerna i denna svit som lyckades och vilka som inte gjorde det.

1.3 Mål

I början av projektet såg målen ut som följer:

- Visualisera testresultat med hjälp av Robot Framework. Testresultaten ska kunna läsas i textform eller kunna representeras grafiskt i Jenkins webbgränssnitt.
- Utveckla testfall i Robot Framework för att verifiera implementerad funktionalitet hos testobjektet. Det vill säga om någon till exempel laddar upp ny kod där ett specifikt varvtal kan sättas ska Robot Framework snabbt kunna avgöra om det lyckades eller inte.
- Använda ett testautomatiseringsverktyg (Jenkins) för att automatiskt köra relevanta testfall vid incheckning av kod till Git. Först och främst skall Jenkins testa så att koden går att kompilera, därefter skall enhetstester utföras, koden ska laddas upp till testobjektet och sist i kedjan startas acceptanstesterna. Om något av stegen i ledet misslyckas kommer efterföljande ej köras.
- Kunna styra varvtal på elmotorn via CAN-meddelanden.
- Ta fram en CAN-databasfil med relevanta meddelanden för att styra elmotorn. Dessa meddelanden ska åtminstone kunna starta och stoppa motorn samt sätta den till ett visst varvtal.

1.4 Avgränsningar

- Endast mjukvara som berör testobjektet och testplattformen kommer implementeras.
- Hårdvaran som används kommer inte modifieras.

2 Teknisk bakgrund

Detta kapitel utgör en redogörelse för tekniker, verktyg, koncept och tekniska begrepp som är av betydelse för projektets genomförande.

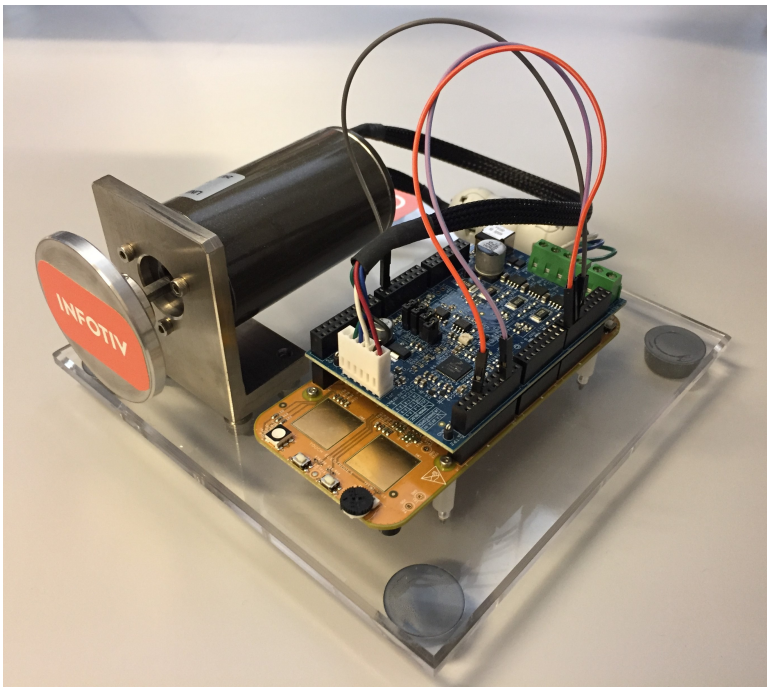
2.1 Mikrokontroller

Projektets testobjekt är en ARM-baserad mikrokontroller som ska programmeras för att implementera en motorstyrenhet till en borstlös likströmsmotor. För att programmera testobjektet används "S32 Design Studio" som är en integrerad utvecklingsmiljö från tillverkaren av mikrokontrollern.

2.2 Borstlös likströmsmotor

Projektets testobjekt, mikrokontrollern, styr en borstlös likströmsmotor. Denna typ av motor har enligt Yedamale [1] flera fördelar jämfört med en klassisk likströmsmotor, bland annat ökad livslängd, högre verkningsgrad och ett bättre förhållande mellan storleken på motorn och vridmomentet som den kan leverera. Tre centrala beståndsdelar är rotorn, statorn och Hall-sensorerna. Rotorn består av permanenta magneter och på statorn sitter parvisa kopparlindningar placerade i en stjärnformation. Motorn roterar när när lindningarna strömsätts i en viss sekvens. För att kunna avgöra rotorns position och därigenom veta vilka lindningar som behöver strömsättas för att den ska rotera används tre Hall-sensorer. Dessa avgör rotorns placering genom att mäta styrkan på magnetfälten [1].

En bild på mikrokontrollern och motorn kan ses i figur 2.1.



Figur 2.1: Motorn som ska styras och mikrokontrollern som ska implementera en motorstyrenhet.

2.3 CAN-protokollet

I detta projektet används bussprotokollet *Controller-Area Network* (CAN) för att skicka kommandon till styrenheten. Det uppfanns 1986 och var från början tänkt att användas till bilar men används idag till allt från fordon till skrivare. Alla noder i ett CAN nätverk är sammankopplade till en gemensam buss. På denna buss är 0-signaler dominanta vilket leder till att 1-signaler är recessiva. Varje nod i nätverket har ett ID nummer som avgör hur högt noden prioriteras framför andra noder i nätverket. Ett högt ID nummer har låg prioritet, ett lågt har hög. På grund av att noder prioriteras är CAN-protokollet lämpligt att använda i realtidssystem. Exempelvis i ett fordon ska en nod med ansvar för bromsar ha högre prioritet än en nod som ansvarar för radion.

En av de stora fördelarna med CAN är dess pålitlighet. När en nod skickar sitt meddelande kontrollerar den löpande vad som skickas ut på bussen. Detta jämförs med vad som borde skickas ut. Om det mot förmodan skulle skulle skilja sig, kommer ett fel registreras. När ett meddelande har skickats, skickas även en kontrollsumma. De mottagande noderna kommer registrera ett fel om den inte skulle stämma. Det finns räknare som håller reda på hur många fel varje nod har upptäckt. Om en mottagande nod ständigt upptäcker fel på grund av att den är trasig, kommer den tillslut inte kunna registrera fel längre, utan meddelandet kommer godkännas. Anledningen till detta är att meddelande ska kunna skickas och inte konstant bli invaliderade av fördärvade noder [2].

2.4 JSON

Automatiseringsservern som används i detta projekt, Jenkins, skriver ut statusutskrifter i ett filformat som kallas *Javascript Object Notation* (JSON). JSON är ett textbaserat datautbytesformat. Formatet har två typer av strukturer: objekt och fält, och fyra grundläggande typer: null, strängar, nummer och booleska variabler [3].

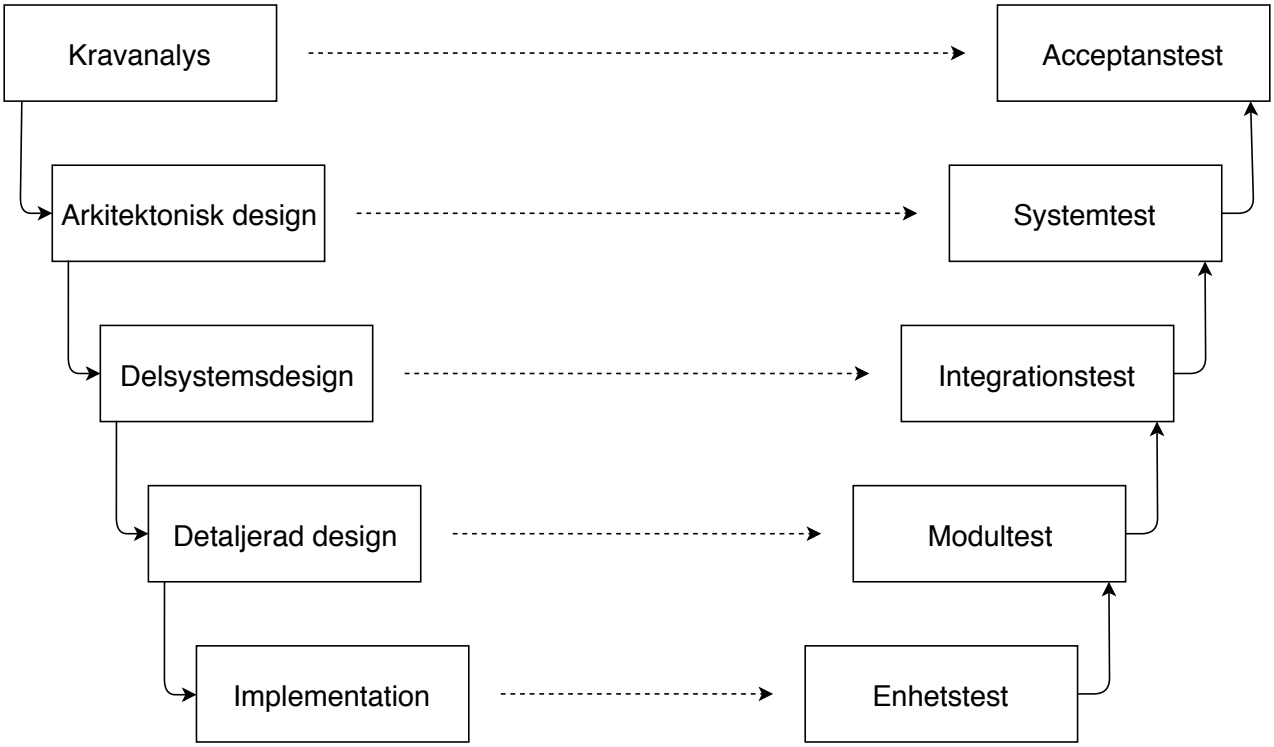
2.5 Testning

Testning utgjorde en central del av detta projektet eftersom meningen var att framställa en automatiskt testkedja. Detta underkapitel redogör för viktiga begrepp och verktyg inom området testning som projektet använder sig av.

2.5.1 Acceptanstestning

Acceptanstestning är enligt Ammann och Offutt [4] en nivå av testning där programvara bedöms med hänseende till användarkraven. Ett tidigt steg i mjukvaruutvecklingen är ett genomföra en kravanalys och acceptanstestningen är utformad för att svara på om den färdiga mjukvaran lyckas möta de behov som har identifieratsi denna kravanalys. Kort kan alltså acceptanstestning beskrivas som en nivå av testning som avgör ifall mjukvaran faktiskt gör vad användaren vill.

I V-modellen (se figur 2.2), som beskriver vilka mjukvaruutvecklingssteg olika testnivåer vanligtvis relateras till, brukar acceptanstestning placeras på den översta nivån. Den kan alltså påbörjas så fort systemtestningen och testen från de övriga nivåerna är klara. Dock rekommenderas att acceptanstesten implementeras i samband med att motsvarande utvecklingssteg genomförs [4].



Figur 2.2: V-modellen

2.5.2 Ceedling

Ceedling är ett byggverktyg för C-projekt som används för att underlätta testdriven utveckling. Det används för att sammanfoga funktionalitet från verktygen Cmock, Unity och CException [5]. I detta projekt används verktyget för att implementera och exekvera enhetstesterna för testobjektet.

2.5.3 Nyckelordsdriven testning

Laukkanen [6] menar att en begränsning med datadriven testning är att alla testfall liknar varandra och att det krävs programmering för att lägga till nya testfall. Detta går att lösa med hjälp av nyckelordsdriven testning där inte bara testdata utan också instruktioner för vad som ska göras med testdatan skickas med i testfilerna. Instruktionerna kallas för *nyckelord* och dessa kan sättas samman för att skapa nya testfall även av någon som inte har programmeringsbakgrund. Nyckelorden kan implementera olika programmeringsnivåer. De nyckelord som implementerar funktioner på låg nivå kan sedan användas för att bygga upp funktioner på högre nivåer [6]. Detta projekt använder sig av nyckelord på hög nivå för att utföra acceptanstester. Detta för att göra testfallen lätta att förstå och designa. I figur 2.3 ses några exempel på nyckelord som används i detta projekt.

2.5.4 Gherkin-syntax

Acceptanstesterna som skrivits i Robot Framework använder sig av en syntax som kallas "*Gherkin*". Kortfattat innebär det att man har tre steg i varje test. Det första är "*Given*", det andra är "*When*" och det sista är "*Then*". Given är något som är givet, exempelvis att motorn i ett fordon redan är igång. When är något som görs, kanske att en knapp trycks in. Det sista

steget, Then, beskriver ett utfall som borde ske [7]. I figur 2.3 kan man se hur gruppen har använts sig av denna syntax för att testa en lysdiod. Givet att lysdioden är släckt från början, och när en array med värden matas in ska sedan samma värden kunna läsas ut. Stämmer detta kommer testet godkännas. På samma figur kan även vissa nyckelord ses och dessa används ungefär som funktioner av testfallen. Till exempel “LED is turned off”, där ska alla värderna på lysdioden vara 0, vilket innebär att inget rött, grönt eller blått ljus kommer lysa. Det vill säga, inget ljus alls.

```
*** Test Cases ***

Test RED
    Given LED is turned off
    When LED is set to ${253} ${5} ${5}
    Then color should be ${253} ${5} ${5}

Test BLUE
    Given LED is turned off
    When LED is set to ${5} ${253} ${5}
    Then color should be ${5} ${253} ${5}

Test GREEN
    Given LED is turned off
    When LED is set to ${2} ${2} ${253}
    Then color should be ${2} ${2} ${253}

*** Keywords ***
LED is turned off
    Set LED color           ${0}    ${0}    ${0}

LED is set to ${r} ${g} ${b}
    Set LED color           ${r}    ${g}    ${b}

Color should be ${r} ${g} ${b}
    LED should be           ${r}    ${g}    ${b}
```

Figur 2.3: Exempel på nyckelord och den syntax som används vid acceptanstestningen.

2.6 Testautomatisering

Detta underkapitel förklar vilka verktyg och tekniker projektet använde sig av för att möjliggöra automatisk testning.

2.6.1 Jenkins

Syftet med Jenkins är automatisering av byggning och testning av kod. Det är skrivet i Java och är dessutom väldigt lätt att bygga ut tack vare stöd för plugins [8]. Ett av dessa plugins som kommer att användas är ett som visar ”pipelines”, det vill säga arbeten som är kedjekopplade.

2.6.2 Automatisk skriptkörning

Enligt Chacon [9] finns det många versionshanteringssystem som erbjuder ett sätt att automatiskt starta skript efter att vissa händelser inträffar. I programmet som projektet använder sig av för versionshantering, Git, kallas denna funktionalitet för hakar. Det finns två olika typer av hakar: de som används på serversidan och de som används på klientsidan. Händelser som dessa kan reagera på kan till exempel vara att en användare vill checka in kod systemet, sammanfoga två grenar eller synkronisera sin lokala programkod med den som finns på versionshanteringsservern. I den lokala kopian av ett Git-förvar finns alltid en dold underkatalog med namnet ".git" och denna innehåller i sin tur en katalog med namnet "hooks". I denna katalog placeras skripten som användaren vill installera och starta automatiskt. Vilken händelse ett skript ska starta efter bestäms av dess filnamn. Ifall användaren till exempel vill att ett skript ska startas efter att två grenar sammanfogats, anges filnamnet "post-merge". Vilka händelser som finns tillgängliga anges i dokumentationen [9]. En för projektet viktig detalj för hur hakar i Git fungerar på klientsidan är att de inte automatiskt kopieras över till resten av användarna efter att de installerats. Den som vill använda sig av ett visst skript måste därför själv installera det lokalt på sin egen maskin [9].

2.6.3 Robot Framework

På den officiella hemsidan för Robot Framework går att läsa att det är ett ramverk för acceptanstestning och acceptanstestdriven utveckling. Testerna skrivs med hjälp av nyckelord vilket gör dem lättlästa även för medarbetare utan programmeringsbakgrund. När testerna körs från kommandoraden genereras en rapport och en log i HTML- och XML-format. Ramverket utvecklades till en början vid Nokia Networks men är idag öppen källkod som sponsras av Robot Framework Foundation. Det används bland annat av Nokia, ABB och Finnair [10]. I detta projekt används Robot Framework för acceptanstestningen av testobjektet.

2.7 Kontinuerlig integration

Kontinuerlig integration är enligt Meyers [11] ett antal principer som utvecklare kan använda sig av för att motverka problem med att programvara fungerar i vissa miljöer men inte i andra. All kod måste förvaras i ett versionshanteringssystem. När någon checkar in ny kod i systemet ska detta automatiskt upptäckas och tester ska startas för att avgöra om de nya ändringarna fungerar med den existerande programvaran. Dessa tester görs alltid i samma miljö oavsett vem som har checkat in ändringarna. På så sätt undviks tidskrävande problem med att programvaran fungerar på en viss dator men inte på en annan eftersom alla i utvecklingsgruppen utvecklar mot exakt samma system. För att åstadkomma detta krävs det att programvaran byggs ihop automatiskt så att testen kan köras [11]. Detta kan göras med någon form av automatiseringsserver som till exempel Jenkins vilket detta projekt använder sig av.

Vidare menar Meyers att en annan viktig princip för effektiv kontinuerlig integration är att alla medlemmar i utvecklingsgruppen checkar in sina ändringar till projektets versionshanteringssystems huvudgren ofta. Hur ofta detta ska göras kan skilja sig från projekt till projekt men en grundregel är minst dagligen. Syftet med detta är att uppmuntra till att regelbundet kontrollera så att nyskriven kod fungerar tillsammans med den existerande mjukvaran. Ifall detta inte görs uppstår en risk för att utvecklarna jobbar i var sin lokal gren under längre perioder vilket kan leda till problem när grenarna ska slås samman [11].

2.8 PXI-plattformen

PXI-systemet som används i projektet är tillverkat av National Instruments. PXI (PCI Extensions for Instrumentation) är enligt National Instruments en PC-baserad plattform för mät- och automationssystem. Plattformen är uppbygd på samma sätt som en vanlig persondator med ett chassi, nätaggregat, kylning och en kommunikationsbuss. Beroende på vilka behov ett projekt har kan olika moduler installeras för att utföra olika typer av mätningar eller tester. Modulerna kan sedan kontrolleras med en inbyggd styrenhet eller en extern dator. Om en extern dator används så kan modulerna styras med hjälp av National Instruments programvara Veristand [12]. Detta projekt använder sig främst av en modul på PXI-platformen som låter chassit ansluta till, och kommunicera med, en CAN-buss. Programvaran Veristand styr modulerna och används för att skicka och ta emot CAN-meddelanden.

2.8.1 Hardware-in-the-loop-dator

Modulerna på PXI-plattformen kontrolleras i detta projekt av en extern dator. Denna kallas hardware-in-the-loop-datorn och den antas alltid vara ansluten till PXI-plattformen och testobjektet. Detta eftersom den automatiserade testkedja som projektet tar fram förlitar sig på att enhets- och acceptanstester för testobjektet alltid kan köras via PXI-plattformen. En integrerad utvecklingsmiljö för motorstyrenheten finns installerad på HIL-datorn och eftersom datorn är ansluten till PXI-plattformen utgör den innan projektes början den mest effektiva platsen för att utveckla mjukvara för motorstyrenheten. Detta eftersom bara datorer som är direkt anslutna till PXI-plattformen har möjlighet att köra tester på testobjektet ifall det inte finns någon automationskedja.

3 Metod

Sättet att arbeta delades upp i 3 olika delar. Den första innefattade att läsa på om hur ett visst problem kan lösas, den andra var experimenterande och den tredje var storskalig implementering och verifiering. Till förfogande hade gruppen 3 datorer, ett testsystem, ett testobjekt och flera olika programvaror.

3.1 Kunskapssamlande

Många av delarna som innefattades i projektet hade gruppmedlemmarna aldrig tidigare arbetat med. Till exempel Jenkins och Robot Framework. I dessa fall var det nödvändigt att samla på sig grundläggande kunskaper om mjukvaran. Kunskaper som förklarar varför den ska användas, hur den ska användas och vad den kan tillföra. Insamlingen kom från många olika källor. En av de mest pålitliga källorna var dokumentation som skickas med mjukvaran. Även guider på olika sajter och forum var till hjälp. Om någon eller några medarbetare på kontoret hade erfarenhet med verktyget kunde det vara lämpligt att ställa frågor till dessa. I de fall ingen av de föregående metoderna var tillräckliga kunde videoguider studeras.

Vissa av delarna i projektet var dock saker som projektdeltagarna hade arbetat med tidigare, såsom C- och Pythonprogrammering. Tanken var att projektgruppen skulle jobba vidare på en redan etablerad kodbas för exempelvis motorstyrenheten. I detta fall hade en medarbetare på Infotiv redan börjat skriva kod vilket gjorde det nödvändigt att sätta sig in i vad personen hade gjort och hur dennes kod var uppbyggd. Vid behov tillfrågades även denna person om det skulle vara något som inte var helt klart.

3.2 Experimenterande

När tillräckligt mycket information hade ackumulerats började experimentstadiet. För att kunna komma igång snabbt gjordes dessa experiment så små och enkla som möjligt. Dessutom var de helt avskilda från resten av projektet. Dessa experiment var till för att se om projektdeltagaren hade förstått hur mjukvaran används. När experimentet fått det resultat som förväntades gick processen vidare till nästa steg.

Ett exempel skulle kunna vara att bara sätta upp en lokal Jenkins-server och skriva några få testarbeten. Testerna skulle kunna vara att få en annan dator i samma nätverk att utföra ett jobb som exempelvis att hämta ner senaste versionen av ett projekt som ligger på Git. Detta utan någon interaktion från människor.

3.3 Implementering och verifiering

I detta sista steg skalades det tidigare experimentet upp och integrerades med hela projektet. Exemplet i föregående stycke skulle kunna integrerats med hela projektutvecklingen, så att en annan dator automatiskt laddade ner den senaste versionen av programkoden och genomförde tester på den varje gång ny kod synkroniserades med versionhanteringsservern. Ett standardtest var att kontrollera att det var möjligt att kompilera koden. Verifieringen utgjordes bland annat av enhetstester. Dessa var till för att se att en ny version av projektet inte hade orsakat ett fel i en funktion som tidigare hade fungerat. Om problem uppstod drogs koden tillbaka och författaren blev tvungen att se över programkoden och uppdatera den.

3.4 Kontinuerliga möten

För att säkerställa att projektet höll rätt kurs och för att hålla deltagarna synkroniserade med varandra var avsikten att i projektets inledningsskede hålla möten med alla deltagare och projektledaren två gånger per vecka. På dessa möten diskuterades hur arbetet hade fortlöpt sedan senaste mötet, ifall någon hade stött på några problem med sina uppgifter och vad var och en planerade att jobba med härnäst. När projektmedlemmarna började känna sig bekväma i sina roller och projektet hade gått framåt enligt plan minskades antalet möten per vecka.

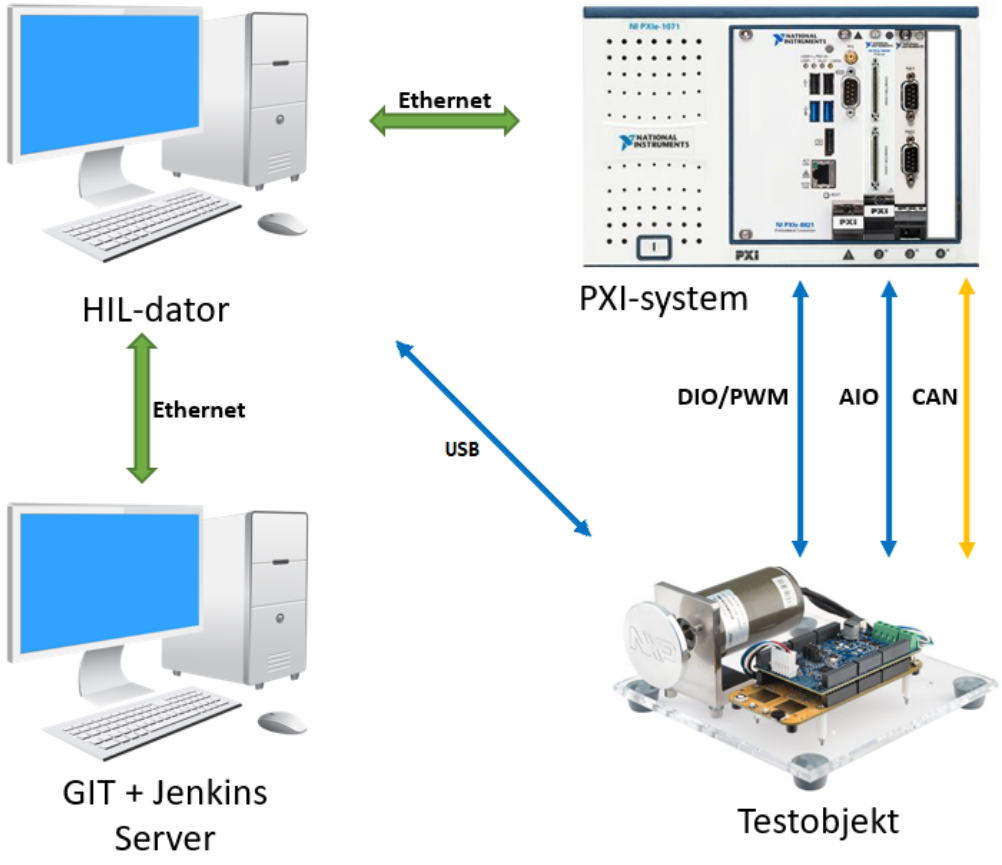
3.5 Implementeringen av kontinuerlig integration

För att hålla huvudgrenen uppdaterad använde sig detta projekt av ett arbetsflöde där ny funktionalitet utvecklades i separata grenar som sammanfogades med huvudgrenen när den nya funktionaliteten var klar. Det var viktigt att varje gren inte innehöll för mycket funktionalitet eftersom det då gick för lång tid innan huvudgrenen blev uppdaterad. När en ny funktion skulle implementeras skapades först en beskrivning av vad den skulle göra. Denna beskrivning lades i ett system som var separat från versionshanteringssystemet och som fanns tillgängligt för alla projektmedlemmar. Beskrivningen tilldelades ett nummer och utvecklaren som skulle arbeta med funktionen checkade ut en ny gren med detta nummer som grennamn i versionshanteringssystemet. Innan implementeringen påbörjades skulle utvecklaren enligt principen för testdriven utveckling skriva tester för funktionen och köra dessa för att se att de inte lyckades. Funktionen implementerades därefter och ansågs vara klar när testerna gått igenom. Innan den nya koden kunde sammanfogas med huvudgrenen behövde integrationstester skrivas. När dessa blev godkända kunde funktionaliteten sammanfogas med resten av programvaran. En fördel med att jobba på detta sätt var att det alltid fanns en stabil huvudgren med funktionalitet som när som helst skulle kunna visas upp för en intressent som till exempel produktägare eller projektledare.

4 Systemkonstruktion

Detta projekt är en vidareutveckling av ett existerande projekt på Infotiv och därför fanns viss funktionalitet implementerad från start.

Systemet som projektet bygger på består av 4 huvudsakliga delar: en server avsedd för att köra Git och Jenkins, en HIL-dator, ett PXI-system samt ett testobjekt. Servern har alla datorer inom nätverket tillgång till, HIL-datorn är kopplad till PXI-systemet via ethernet-anslutning och testobjektet är i sin tur kopplat till PXI-systemet via CAN. Testobjektet är dessutom kopplat till HIL-datorn via USB. Hela kopplingskedjan kan ses i figur 4.1.



Figur 4.1: Projektets beståndsdelar.

Vid projektets början har en CAN-databas redan påbörjats. Denna ska under projektets gång vidareutvecklas med bland annat stöd för motorstyrning. Det finns funktionalitet för att kontrollera motorstyrenhetens lysdioder, för att skicka och ta emot CAN-meddelanden och för att hantera avbrott.

För acceptanstestningen finns dessutom ett bibliotek av funktioner skrivna i Python påbörjat och ett färdigställt exempel för hur tester ska skrivas i Robot Framework. Biblioteket ska utvidgas med mer funktionalitet. Enhetstester för programkod skriven i C finns redan implementerade.

4.1 Git- och Jenkins-server

Git och Jenkins kommer vid projektets slut inte ligga på samma server men har här slagits ihop för enkelhets skull. Git ska i detta projektet användas för att versionshantera mjukvara till både HIL-datorn och testobjektet. Servern för Git fanns redan innan projektet startade och gruppen har inte behövt lägga ner något arbete på denna. Desto mer tid har lagts ner på Jenkins-servern. I dagsläget startas en Jenkins-server på en av gruppens bärbara datorer, tanken är dock att den ska ligga på en separat server-dator. Syftet med Jenkins-servern är att fördela arbetsuppgifter till andra datorer som kallas slav-noder. Som det ser ut i dagsläget finns endast en slav-nod, HIL-datorn.

Alla datorer i nätverket har möjlighet att starta arbete på Jenkins-servern. Det kräver dock tre saker: att användaren har konfigurerat Git korrekt på den egna datorn, att det finns ett skript (så kallad hake, se kaptiel 2.6.2) på rätt plats i datorns filsystem samt att användaren skriver en särskild nyckelfras i sina commit-meddelanden.

4.1.1 Jenkins-pipeline

Projektets Jenkins-pipeline är den kedja av Jenkins-arbeten som utgör testautomatiseringen. Arbetena körs sekventiellt under förutsättning att föregående arbete lyckades. Om ett arbete i kedjan misslyckas avbryts exekveringen av pipelinen. Nedan följer en beskrivning av samtliga arbeten i den ordning de exekveras av Jenkins.

Starta pipelinen och checka ut de senaste ändringarna

Testautomatiseringen är tänkt att användas när någon i utvecklargruppen vill köra samtliga acceptans- och enhetstester. Användaren av kedjan måste då se till att rätt Git-hake installerats på dess personliga dator. Haken reagerar på att användaren synkroniserar kod med Git-servern och varje gång detta sker körs ett enkelt bash-skript. Det första skriptet gör är att kontrollera ifall nyckelfrasen “RUNTESTS” finns med i commit-meddelandet för den senaste commiten. Ifall nyckelfrasen inte finns med så avslutas skriptet. Testkedjan behöver alltså inte köras vid varje synkronisering. Har användaren däremot angett nyckelfrasen i det senaste commit-meddelandet så skrivs namnet på Git-grenen som användaren står i som en sträng till en variabel. Skriptet startar därefter det första jobbet i pipelinen och skickar med namnet på grenen som parameter till Jenkins.

Det första Jenkins-jobbet har som uppgift att hämta koden som användaren vill testa. Till att börja med måste därför rätt gren checkas ut på HIL-datorn. Detta kräver att det inte finns några osparade ändringar eftersom Git då inte kommer låta HIL-datorn byta från grenen som den står på till grenen som innehåller koden som användaren vill testa. Inledningsvis löstes detta genom att alltid återställa koden på HIL-datorn till hur den såg ut vid den senast sparade ändringen. På så vis raderades eventuella osparade ändringar och grenbytet kunde därmed genomföras. Problemet med detta tillvägagångssätt är att den som använder sig av testautomationskedjan riskerar att radera arbete för någon som har valt att arbeta direkt vid HIL-datorn istället för att använda sin personliga arbetsdator. Under projektets gång har det visat sig att vissa arbetsuppgifter går snabbare att utföra direkt på HIL-datorn eftersom denna är direkt kopplad till PXI-systemet. För att fortfarande ha kvar möjligheten att kunna utveckla direkt på HIL-datorn implementerades en lösning där osparade ändringar inte raderas utan istället läggs på en stack med hjälp av ett kommando som i Git kallas för “stash”. På

detta sätt kan grenbytet göras utan risk för att arbete går förlorat.

När den senaste koden som ska testas har hämtats till HIL-dator är det första Jenkins-arbetet klart och pipelinen fortsätter till nästa.

Kompilering

Kompileringssteget i pipelinen kompilerar projektkoden. Under utvecklingsarbetet görs detta i S32 Design Studio vilket kräver input från utvecklaren. För att automatisera detta steg körs istället kompilatorn direkt från kommandoraden av Jenkins.

Enhetstestning

När kompileringen är klar startas enhetstesterna. Dessa körs med hjälp av verktyget Ceedling. Testerna har implementerats av en medarbetare på Infotiv, inte av projektgruppen. Ceedling körs med hjälp av kommandotolken Bash och utskriften från verktyget kan ses i figur 4.2.

Som synes i figur 4.2 finns flera moduler: “test_CAN.c”, “test_LED.c”, “test_interrupts.c” och så vidare.

Varje klass innehåller ett antal testfall som alla körs varje gång Ceedling startas, förutsatt att inget av dem är satta till att ignoreras. Ett av testfallen i “test_LED.c” är till exempel att kontrollera att lysdioden faktiskt lyser med blått ljus när den ska.

Mjukvarunedladdning

Ifall kompileringen lyckas och enhetstesterna går igenom måste projektkoden köras på testobjektet så att acceptanstesterna kan utföras. För att automatisera detta behöver en debug-server och en debug-klient köras på HIL-datorn. Även detta görs under utvecklingsarbetet från S32 Design Studio men både servern och klienten går även att köra direkt från kommandoraden. Jenkins ställer sig därmed i katalogen på HIL-datorn där S32 Design Studio installerats och kör debug-klienten.

Kring exekveringen av debug-servern uppstod dock problem eftersom servern är en process utan ett tydligt slut. Ifall den startas av Jenkins kommer Jenkins därför inte veta när debuggingen är avslutad eftersom serverprocessen kommer att köra fram till att den manuellt avbryts. När en viss tid gått efter att jobbet startades kommer det därför att avslutas och markeras som misslyckat vilket i sin tur avbryter hela testkedjan. Lösningen på problemet blev att låta debug-servern köra konstant på HIL-datorn. Detta gör att Jenkins själv inte behöver starta processen och därmed inte heller behöver en klarsignal från den för att kunna markera ett jobb som lyckat.

Acceptanstestning i Robot Framework

Automatiseringskedjan sista steg. För att köra testerna används Robot Framework och om detta Jenkins-jobb blir markerat som godkänt så har alla steg i kedjan lyckats och alla tester gått igenom. Dessa tester är skrivna med Gherkin-syntaxen för att det ska vara lätt för människor utan programmeringskunskaper att kunna förstå de olika testfallen. Robot-filen fungerar ungefär som ett abstraktionslager. Det är dock fullt möjligt att köra tester helt utan Robot Framework. All kommunikation med PXI-systemet sker via Veristand och det går att styra Veristand med hjälp av ett applikationsprogrammeringsgränssnitt. Olika anrop från applikationsprogrammeringsgränssnittet används i en python-fil för att till exempel säga åt

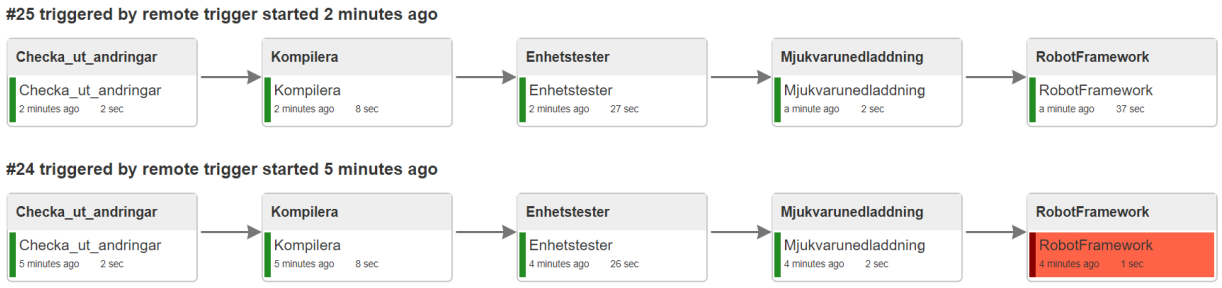
```
Test 'test_CAN.c'
-----------------
Running test_CAN.out...

Test 'test_CAN_messages.c'
--------------------------
Running test_CAN_messages.out...

Test 'test_LED.c'
-----------------
Running test_LED.out...

Test 'test_interrupts.c'
------------------------
Running test_interrupts.out...

Test 'test_main_utilities.c'
----------------------------
Running test_main_utilities.out...

Test 'test_timer.c'
-------------------
Running test_timer.out...

--------------------
OVERALL TEST SUMMARY
--------------------
TESTED:  50
PASSED:  50
FAILED:   0
IGNORED:  0
```

Figur 4.2: Utskrift från Ceedling-tester.

PXI-systemet att skicka ut ett CAN-meddelande som säger åt testobjektet att lysa grönt istället för blått. Denna python-fil används som ett bibliotek för Robot Framework-testfallen. De testfall som finns i projektet testar funktionalitet för att släcka lysdioden och för att sätta den till rött, grönt och blått sken. PXI-systemet skickar CAN-meddelanden kontinuerligt och testobjektet skickar kontinuerligt tillbaka CAN-meddelanden.

För att se detaljer om utfallet för varje steg i Jenkins-pipelinen och för att se testresultaten kan användaren logga in på Jenkins-servern för att se en grafisk representation av pipelinen enligt figur 4.3.

Varje del i översiktsbilden är klickbar. För att till exempel se utfallet av testresultaten för acceptanstestningen kan användaren klicka på det steget i kedjan och får då en vy över testresultaten



Figur 4.3: En lyckad körning samt en misslyckad körning.

som i figur 4.4. Resultaten av de tre första stegen i pipelinen och resultatet av enhetstesterna syns dessutom direkt i kommandotolken. Detta åstadkoms genom att hakskriptet anropar Jenkinsjobben och får tillbaka varje jobbs statusutskrift som en JSON-fil. Dessa utskrifter innehåller information om just det arbetet, till exempel om det håller på att byggas eller ej. En sådan utskrift kan ses i figur 4.5. Hakskriptet letar efter strängen "building":false", och när det hittat den skriver det ut den senaste statusutskriften i kommandotolken. Hade det istället skrivit ut utskriften direkt hade det fått tag i en gammal sådan, innan det nya bygget hunnit bli klart. Tack vare att resultatet från enhetstesterna skrivs ut direkt i kommandotolken slipper användaren logga in på Jenkins-servern.

```
Building remotely on HIL_dator in workspace C:\...
[RobotFramework] cmd \c call C:\...

robot TestLED.robot

==============================================================================
TestLED :: Simple robot file for testing that the LED on testobject is wor...
==============================================================================
Test RED                                                              | PASS |
------------------------------------------------------------------------------
Test BLUE                                                             | PASS |
------------------------------------------------------------------------------
Test GREEN                                                            | PASS |
------------------------------------------------------------------------------
TestLED :: Simple robot file for testing that the LED on testobject   | PASS |
3 critical tests, 3 passed, 0 failed
3 tests total, 3 passed, 0 failed
==============================================================================
Output:   output.xml
Log:      log.html
Report:   report.html
Finished: SUCCESS
```

Figur 4.4: Testresultat av acceptanstestningen.

```
{
    "_class":"hudson.model.FreeStyleBuild",
    "actions":[
        {
            "_class":"hudson.model.CauseAction",
            "causes":[
                {
                    "_class":"hudson.model.Cause.UpstreamCause",
                    "shortDescription":"Started by upstream project",
                    "upstreamBuild":53,
                    "upstreamProject":"checkout_changes",
                    "upstreamUrl":"job/checkout_changes/"
                }
            ]
        },
        {

        },
        {

        }
    ],
    "artifacts":[

    ],
    "building":false,
    "description":null,
    "displayName":"#82"
}
```


Figur 4.5: Statusutskrift från ett Jenkins-jobb, i form av en JSON-fil. Nedtill i figuren syns strängen “building”:false” som hakskriptet letar efter.

4.2 Testobjektet

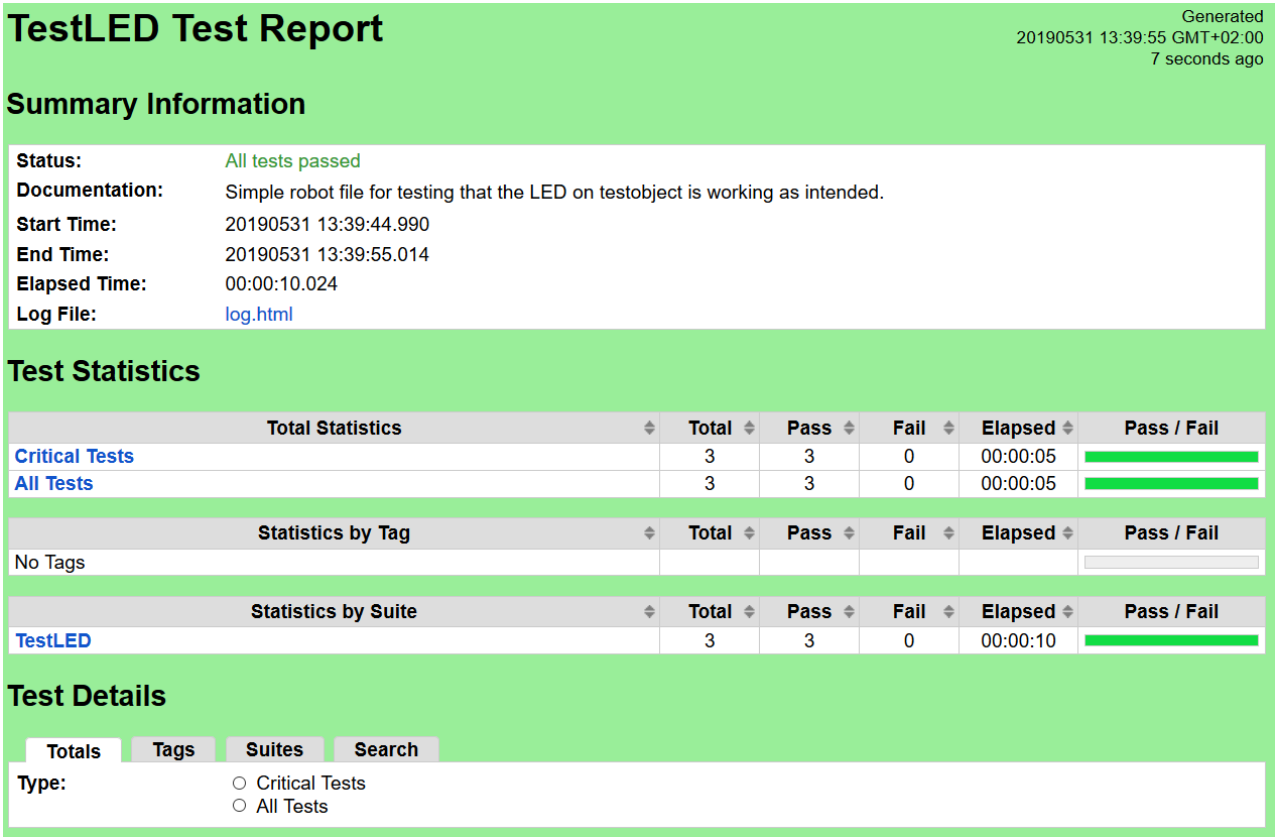
Testobjektet är en styrenhet kopplad till en borstlös likströmsmotor. Initialt var planen att kunna styra denna motor med hjälp av CAN-meddelanden. Detta har dock inte uppnåtts och därför har målen reviderats något. Testobjektet kan ta emot CAN-meddelanden för att styra en lysdiod som sitter på kretskortet och det kan även skicka CAN-meddelanden för att tala om vilka värden som lysdioden har. Dessa meddelanden innehåller tre heltalsvariabler i datafältet: en för intensiteten för röd färg, en för grön och en för blå. Detta system kallas RGB (Red green blue). Meddelanden har dessutom ett ID-nummer och planen var att ett specifikt nummer skulle vara för att styra motorn och ett annat för att styra lysdioden.

5 Resultat

Detta kapitel redogör för vilka resultat projektet levererat.

5.1 Visualisering

Testresultat från automatiseringskedjan presenteras både i textform (se figur 4.2 samt figur 4.4) och grafiskt (se figur 4.3). Så fort en användare pushar upp kod till Git för testning, via exempelvis Bash, kommer information så småningom dyka upp i användarens kommandotolk. Om kedjan lyckades med de föregående stegen körs enhetstesterna via Ceedling. Resultaten för dessa skrivs ut direkt i kommandotolken, där det går att se hur många tester som lyckades och hur många som eventuellt inte lyckades. Förutsatt att dessa tester gick igenom kommer det efter en stund skrivas ut resultat från acceptanstesterna, det vill säga Robot Framework-testerna. Dessa visas också i kommandotolken och där visas varje enskilt test som en låda där de antingen blev stämplade med "FAIL" eller "PASS". Sammanfattningen av dessa visas längst ner i utskriften som kan ses i kommandotolken. Robot Framework sparar även undan rapporter från testkörningarna i form av en textfil, en html-fil och ett xml-dokument. Figur 5.1 visar hur en rapport sparad i html-kod ser ut när den öppnas i en webbläsare. Om det istället är önskvärt att se förloppet grafiskt kan användaren logga in på Jenkinsservern. Där går det att övergripande se nuvarande körning av testkedjan och vilka steg som passerat (gröna) och vilket steg som kanske misslyckats (rött).



TestLED Test Report

Generated
20190531 13:39:55 GMT+02:00
7 seconds ago

Summary Information

Status:	All tests passed
Documentation:	Simple robot file for testing that the LED on testobject is working as intended.
Start Time:	20190531 13:39:44.990
End Time:	20190531 13:39:55.014
Elapsed Time:	00:00:10.024
Log File:	log.html

Test Statistics

Total Statistics	Total	Pass	Fail	Elapsed	Pass / Fail
Critical Tests	3	3	0	00:00:05	
All Tests	3	3	0	00:00:05	

Statistics by Tag	Total	Pass	Fail	Elapsed	Pass / Fail
No Tags					

Statistics by Suite	Total	Pass	Fail	Elapsed	Pass / Fail
TestLED	3	3	0	00:00:10	

Test Details

Totals | Tags | Suites | Search

Type: ○ Critical Tests ○ All Tests

Figur 5.1: Testresultatrapport genererad av Robot Framework.

5.2 Testfall

Projektruppen har implementerat en testsvit i Robot Framework. Denna innefattar endast tester för lysdioden. Testerna kontrollerar att lysdioden kan lysa i vissa färger. Motorstyrenheten skickar kontinuerligt tillbaka CAN-meddelanden till PXI-enheten och dessa meddelanden innehåller information om lysdiodens nuvarande värden. Om dessa värden överensstämmer med de som skickades från PXI-enheten till motorstyrenheten så godkänns testerna.

5.3 Testautomation

Genom att ange ett nyckelfras i commit-meddelandet i kommandotolken, när ny funktionalitet ska synkroniseras med versionshanteringsservern kan en användare starta en Jenkins-pipeline. Denna pipeline kontrollerar automatiskt att koden kan kompileras, att enhets- och acceptanstesterna går igenom och att koden kan överföras till motorstyrenheten. Med denna lösning behöver den som till exempel vill köra acceptanstesterna inte befinna sig på plats vid HIL-datorn.

6 Diskussion

Detta kapitel innehåller en diskussion kring hur väl projektet anses ha lyckats möta vart och ett av de delmål som redovisades i kapitel 1.3. Det avslutas med förslag till fortsatt utveckling samt en reflektion kring projektets miljöpåverkan.

6.1 Visualisering av testresultat

Projektgruppen kom tillsammans med handledare på Infotiv fram till att endast delar av testkedjan behövde presenteras. De viktigaste delarna är enhetstester och acceptanstester så det beslutades att de delarna skulle visas. Till en början skrevs all information ut till användaren och denna inkluderade utskrift från både kompilatorn och S32 Design Studio när mjukvaruneddladdning till styrenheten skedde. Då dessa delar inte är lika intressanta som de andra valdes att inte presentera dem för användaren. Detta gjorde även att det blev avsevärt mindre text att gå igenom vilket underlättar.

I kapitel 5.1 nämndes det också att det är möjligt att direkt logga in på Jenkins-servern för att grafiskt kunna se förloppet av automationskedjan. Detta motverkar dock delvis syftet med arbetet då det kräver att användaren loggar in på servern och måste vänta på kedjan, istället för att den meddelar användaren när relevanta delar är klara.

6.2 Testfall i Robot Framework

Testsviten skulle varit avsevärt mycket större om funktionalitet för motorstyrning hade implementerats. Motorn hade gått att testa på fler sätt än bara lysdioden. Det hade kunnat innefatta tester så som att starta motorn, stoppa den, sätta den i ett visst varvtal, ändra rotationsriktning och så vidare.

När gruppen insåg att den inte skulle lyckas få igång motorn (se kap 6.4) diskuterades det hur testsviten skulle kunna utökas på andra sätt. En metod hade varit att utföra fler tester för olika färger på lysdioden. Detta ansågs dock inte tillföra något då det bara hade inneburit samma tester som innan fast med andra parametrar.

En annan idé var att få lysdioden att blinka i ett angivet tidsintervall, för att sedan testa om den blinkar i rätt hastighet. Denna idé kom dock fram sent i projektet och det hanns helt enkelt inte med att få in den funktionen i koden för styrenheten.

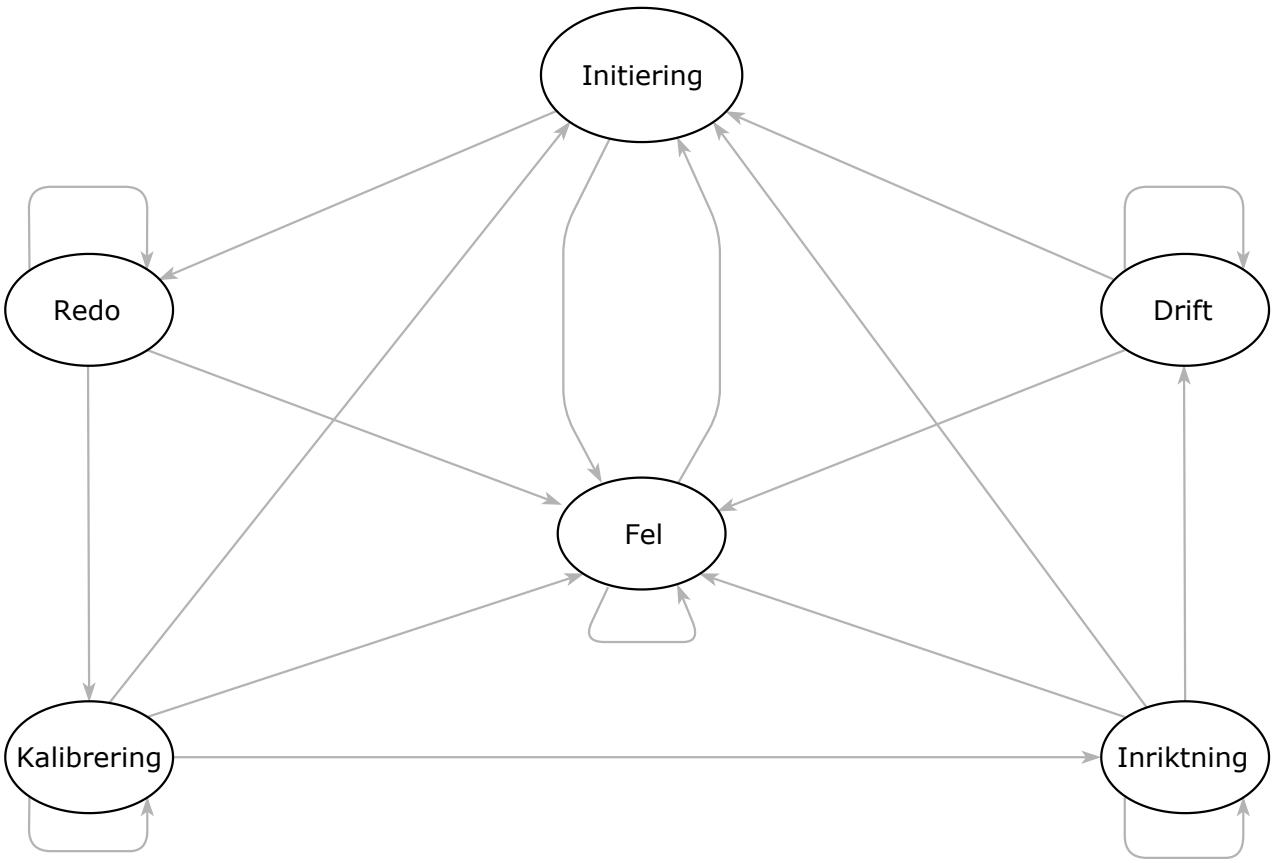
6.3 Automatisk testning

Detta delmål anses vara uppnått i och med att den som använder sig av testkedjan kan använda nyckelfrasen för att automatiskt starta samtliga tester. Den nuvarande lösningen kräver dock att varje användare manuellt installerar ett krokskript på sin maskin första gången den använder sig av nyckelfrasen. En bättre lösning hade varit att det fanns möjlighet att installera skriptet automatiskt för alla användare som klonar projektet från versionshenteringssystemet.

6.4 Styra varvtal via CAN

Under projektets gång lades mycket tid på att försöka nå detta delmål. Tillvägagångssättet var att analysera ett exempelprojekt från tillverkaren till kretskortet som motorstyrenheten skulle implementeras på. Detta projekt innehöll bland annat funktionalitet för motorstyrning och avsikten var att studera exempelprojektet för att försöka förstå hur funktionalitet för att till exempel styra varvtal skulle kunna byggas. Till exempelprojektet fanns även ett tillhörande dokument som beskrev hur det implementerats.

Det uppstod dock problem när lösningarna för motorstyrning från exempelprojektet skulle sammanfogas med den programkod för motorstyrenheten som redan fanns färdigskriven innan projektet påbörjades. Denna programkod innehöll ingen funktionalitet för att styra själva motorn men det fanns stöd för att till exempel skicka och ta emot CAN-meddelanden, styra lysdioder på kretskortet och hantera hårdvaruavbrott. Avsikten var att separera lösningarna för motorstyrning i exempelprojektet och implementera dem i det redan existerande projektet. Denna process visade sig dock vara mer komplex än vad projektgruppen hade räknat med. Exempelprojektet var uppbyggt kring en tillståndsmaskin med sex tillstånd (se figur 6.1). För att till exempel starta motorn behövde motorstyrenheten gå genom tillstånden för initiering, kalibrering, inriktning och drift. Det redan existerande projektet innehöll däremot ingen tillståndsmaskin och därför behövde lösningarna för motorstyrning separeras från tillståndsmaskinen för att kunna användas. Lösningarna för motorstyrning visade sig dock ha många kontaktpunkter med tillståndsmaskinen och att frigöra funktioner från den var komplext.



Figur 6.1: Exempelprojektets tillståndsmaskin.

Exempelprojektet från kretskortstillverkaren utgjorde dessutom ett realtidssystem vilket gjorde det svårt att analysera projektet under körning. När en brytpunkt sattes ut i debugverktyget och exekveringen av exempelprojektet stannade vid denna punkt hamnade systemet i ett feltillstånd. Detta visade sig bero på att programmet behövde kontinuerliga uppdateringar för motorns tillstånd, till exempel information om i vilken position rotorn för närvarande befann sig i. Ifall programmet inte fick dessa uppdateringar genererades ett tidsberoende avbrott som satte systemet i ett felläge. Detta gjorde det svårt att få en klar bild av vilka steg programmet gick igenom under körning för att styra motorn.

Av ovanstående anledningar lyckades projektgruppen inte nå delmålet. Det är möjligt att för mycket tid ägnades åt att försöka analysera exempelprojektet. I dokumentet som tillhörde detta projekt fanns en beskrivning av teorin för hur en borstlös likströmsmotor ska drivas. Projektgruppen ansåg sig inte ha tillräckliga förkunskaper för att implementera denna teori på den tid som avsatts för delmålet. Möjligtvis hade det dock varit bättre att fokusera på någon del av denna teori och att försöka implementera denna del på egen hand. Till exempel hade delmålet kanske kunnat omarbetats till att bara kunna skicka någon form av en sekvens av signaler till motorn för att se att den faktiskt reagerar, snarare än att försöka uppnå komplett motorstyrning.

6.5 CAN-databasfil

Tanken var att lägga till fler CAN-meddelanden i CAN-databasen som ligger på HIL-datorn. Dessa meddelanden skulle starta, stoppa och ändra hastigheten på elmotorn. Eftersom gruppen inte lyckades styra elmotorn så valde den att inte lägga tid på databasen då den inte skulle bidra med något till automatiseringskedjan. Dock skulle en vidareutveckling av CAN-databasen kunna komma till nytta för andra personer som väljer att arbeta med elmotorn. Får en framtida grupp igång motorn skulle den utvidgade CAN-databasen vara värdefull. Mjukvaran till styrenheten känner i dagsläget endast igen meddelanden med ett ID-nummer. Om det gemensamt inom projektet bestäms vad meddelanden för styrning av motorn skall ha för ID-nummer kan detta läggas till i databasen. Det meddelandet skulle kunna ha samma uppbyggnad som styrningen av lysdioden, alltså tre variabler i datafältet. Men istället för att avgöra intensiteten på röd, grön och blå, så skulle det till exempel kunna vara starta/stoppa motorn, höj/sänk hastigheten och ange rotationsriktning.

6.6 Förslag till framtida utveckling

I och med att detta projekt misslyckades med att uppnå vissa av delmålen blir en naturlig fortsättning att jobba vidare med de delar som inte färdigställdes. Ett möjligt framtida projekt skulle kunna vara ett som endast fokuserar på implementering av funktionalitet för motorstyrenheten och på att bygga ut CAN-databasfilen. I ett sådant projekt skulle deltagarna få mer tid på sig att sätta sig in i hur motorn kan styras på ett optimalt sätt. Detta skulle medföra att motorstyrningen skulle kunna produceras från grunden och inte behöva förlita sig på det exempelprojekt som diskuterades i kapitel 6.4.

Vad gäller den automatiska testkedjan skulle en lösning där varje användare inte manuellt behöver installera krokskriptet på sin personliga arbetsdator kunna tas fram. Eftersom kedjan i dagsläget är ett pågående arbete förutsätter den dessutom att programvaror den är beroende av redan körs på servern när den används. Det hade varit användbart ifall en framtida version

kunde hantera en avsaknad av någon av programvarorna och meddela en användare om vad den behöver göra för att använda sig av testkedjan. Nuvarande version kräver dessutom att en debug-serverprocess alltid körs på HIL-dator. I en framtida implementering hade denna server kunnat startats endast vid behov.

6.7 Miljö och etik

Tanken bakom automatiseringskedjan och styrenheten är att möjliggöra för att på liten skala kunna testa en motor för ett fordon. Istället för att behöva utföra tester i ett riktigt fordon med en stor motor kan istället en liten variant av motorn testas på plats. Detta har flera fördelar, bland annat miljömässiga. Personalen slipper transportera sig till annan plats vilket gör att eventuella utsläppsgenererande transporter undviks. Dessutom sparas även energi (el, bensin) in då en liten testmotor förbrukar avsevärt mindre än en fullskalig bilmotor.

7 Slutsatser

Det huvudsakliga syftet med projektet var att ta fram en automatisk testkedja. Gruppen har lyckats ta fram en kedja som byter till den aktuella grenen i Git, kompilerar koden, kör relevanta enhetstester, laddar ner koden till styrenheten och kör acceptanstester, helt utan mänsklig interaktion. Utfallet av dessa tester presenteras i textform för användaren, sparas i loggar och kan även ses på Jenkinsservern. Detta underlättar för den arbetande då tid inte behöver ägnas åt att den egna datorn ska genomföra testerna, utan testerna körs på en server i nätverket. Då är det istället möjligt att fortsätta programmera medan testerna körs, vilket i längden kommer spara tid.

7.1 Implementering av motorstyrning

Delmålet som bestod i att implementera motorstyrning blev aldrig genomfört (se kapitel 6.4). Det ägnades mycket tid åt att försöka lösa problemen eftersom projektgruppen ansåg att det var centralt för projektet men trots det levererades aldrig något resultat. Båda medlemmarna i projektgruppen ansåg i ett tidigt stadie av arbetet med delmålet att de saknade den tekniska bakgrund som hade behövts för att implementera motorstyrningen inom tidsramen för projektet. Vad som borde ha gjorts annorlunda är projektgruppen borde ha kontaktat sin tekniska handledare tidigare i processen och startat en dialog om att antingen omarbeta delmålet eller släppa det helt. På så sätt hade tiden kunnat använts mer effektivt.

Referenser

[1] P. Yedamale, Brushless dc (bldc) motor fundamentals, *Microchip Technology Inc*, **20**, 3–15, 2003, accessed 2019-04-24. URL: http://electrathonoftampabay.org/www/Documents/Motors/Brushless%5C%20DC%5C%20(BLDC)%5C%20Motor%5C%20Fundamentals.pdf.

[2] A. Van Herrewege, D. Singelee och I. Verbauwhede, "Canauth-a simple, backward compatible broadcast authentication protocol for can bus", *ECRYPT Workshop on Lightweight Cryptography*, hämtad: 2019-05-06, vol. 2011, 2011. URL: https://www.esat.kuleuven.be/cosic/publications/article-2086.pdf.

[3] D. Crockford, "The application/json media type for javascript object notation (json)", tekn. rapport, 2006, hämtad: 2019-05-31.

[4] P. Ammann och J. Offutt, *Introduction to Software Testing*, Andra upplagan. Cambridge University Press, 2016, s. 22–23, 388.

[5] (7 juli 2011). Ceedling. hämtad: 2019-05-29, Throw The Switch, URL: http://www.throwtheswitch.org/ceedling.

[6] P. Laukkanen, Data-driven and keyword-driven test automation frameworks, *Master's thesis. Helsinki University of Technology*, 2006, hämtad: 2019-05-29.

[7] G. Lucassen, F. Dalpiaz, J. M. E. van der Werf, S. Brinkkemper och D. Zowghi, "Behavior-driven requirements traceability via automated acceptance tests", *2017 IEEE 25th International Requirements Engineering Conference Workshops (REW)*, hämtad: 2019-05-29, IEEE, 2017, s. 431–434. DOI: 10.1109/REW.2017.84.

[8] V. Armenise, "Continuous delivery with jenkins: Jenkins solutions to implement continuous delivery", *Proceedings of the Third International Workshop on Release Engineering*, hämtad: 2019-04.25, DOI: 10.1109/RELENG.2015.19, IEEE Press, 2015, s. 24–27. URL: https://dl.acm.org/citation.cfm?id=2820701.

[9] S. Chacon och B. Straub, *Pro git*. Apress, 2014, hämtad: 2019-05-08.

[10] (30 juli 2016). Robot framework. hämtad: 2019-05-12, Robot Framework Foundation, URL: https://robotframework.org/.

[11] M. Meyer, Continuous integration and its tools, *IEEE software*, **31**, nr 3, 14–16, 2014, hämtad: 2019-04-25, DOI: 10.1109/MS.2014.58. URL: https://ieeexplore.ieee.org/document/6802994.

[12] (19 mars 2019). Introduction to the pxi architecture. hämtad: 2019-05-06, National Instruments, URL: http://www.ni.com/sv-se/innovations/white-papers/14/introduction-to-the-pxi-architecture.html.