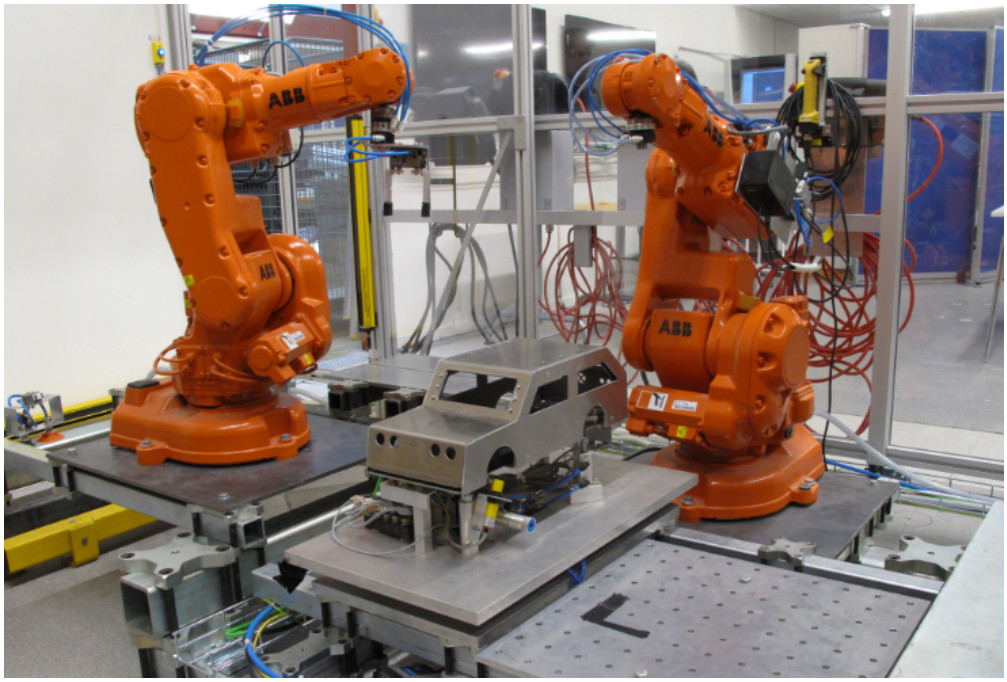
CHALMERS



Flexibel utveckling av automatiserade produktionssystem

Modulärt styrkoncept och automatiskt genererad kod

Kandidatarbete inom civilingenjörsprogrammet

Matilda Anulf
Pontus Johansson
Simon Kardell
Erik Norén

Institutionen för Signaler och System
Chalmers Tekniska Högskola
Göteborg, Sverige 2015

Kandidatarbete 2015: SSYX02–15–16

Flexibel utveckling av automatiserade produktionssystem

Modulärt styrkoncept och automatiskt genererad kod

Matilda Anulf

Pontus Johansson

Simon Kardell

Erik Norén

CHALMERS

Institutionen för Signaler och System
Avdelningen för Automation
Chalmers Tekniska Högskola
Göteborg, Sverige, 2015

Flexibel utveckling av automatiserade produktionssystem
Modulärt styrkoncept och automatiskt genererad kod
Matilda Anulf
Pontus Johansson
Simon Kardell
Erik Norén

Handledare: Kristofer Bengtsson, Institutionen för Signaler och System
Examinator: Knut Åkesson, Institutionen för Signaler och System

Kandidatarbete 2015: SSYX02–15–16
Institutionen för Signaler och System
Avdelningen för Automation
Chalmers Tekniska Högskola
412 96 Göteborg

Omslag: Produktionssystemet i Production System Laboratory.

Typeset in LaTeX
Tryckt av institutionen för Signaler och System
Göteborg, Sverige, 2015

Flexibel utveckling av automatiserade produktionssystem
Modulärt styrkoncept och automatiskt genererad kod
Matilda Anulf, Pontus Johansson, Simon Kardell, Erik Norén
Instutionen för Signaler och System
Chalmers Tekniska Högskola

Sammandrag

Automationsgraden inom produktion har varit på stark tillväxt under senare år. I takt med att automationsgraden växer blir även utvecklingsteamen större. För att få en tydligare struktur över koden samt att minska kostnaderna för utvecklingen i form av kortare utvecklingstider och mindre felsökning, använder sig företag av tydliga och strikta standarder under utvecklingen. Dessa standarder leder ofta till att vissa segment inom koden förekommer med hög frekvens och utgör ofta ett icke kognitivt och repetitivt arbete under utvecklingen.

Den här rapporten innefattar konstruktion och utvärdering av ett koncept för styrning av ett automatiserat produktionssystem. Konceptet berör primärt utveckling av en standard som ska sammanlänka Volvo Car Corporations nuvarande standard för PLC-logik med den forskning och utveckling som bedrivs på Chalmers Tekniska Högskola. Konceptet innefattar även robotprogrammering samt framtagning av en extern utvecklingsmiljö för PLC-logik, där frekvent förekommande kodsegment genereras automatiskt i syfte att reducera utvecklingstiden i ett projekt.

Resultatet av detta projekt är en modulär och överskådlig standard för PLC-logik. Konceptet har blivit verifierat i ett produktionssystem där repetitiva kodsegment har genererats automatiskt. Empiriska tester visar att utvecklingstiden har minskat med 65 % där den automatiska kodgenererngen har implementerats.

Nyckelord: Automation, PLC, Standard, Robotik, Automatisk kodgenerering, Automatiserade produktionssystem

Flexible Development of Automated Production Systems
Modular Concept for Control of Automated Production Systems and Automatic Code Generation
Matilda Anulf, Pontus Johansson, Simon Kardell, Erik Norén
Department of Signals and Systems
Chalmers University of Technology

Abstract

The level of automation within production has grown over the past years. Due to the growth of automation, the development teams have also grown larger. To get a strict structure of the code and to minimize costs in form of reduced development time and less troubleshooting, companies uses a strict standard under the development process. These standards often lead to certain segments which are repetitive within the code and represent a non-cognitively and repetitive work during the development process.

This report includes the design and evaluation of a concept for control of an automated production system. The concept will primarily focus on the development of a standard that will link Volvo Car Corporations current PLC-standard with the research conducted at Chalmers University of Technology within the field. The concept also includes robot programming and the development of an external programming environment for PLC logic, where frequent code can be automatically generated in order to reduce the development time of an project.

The result of this project is a standard for PLC logic that is both modular and intuitive. The concept has been verified in a production system where repetitive code segments have been automatically generated. Empirical tests show that the development time has been reduced by 65 % when the automatic code generating concept was implemented.

Keywords: Automation, PLC, Standard, Robotic, Automatic Code Generation, Automated Production System

Förord

Denna rapport är en del av kandidatarbetet *Flexibel utveckling av automatiserade produktionssystem* som genomfördes vid Chalmers Tekniska Högskola vårterminen 2015. Vi skulle vilja rikta ett stort tack till följande personer för hjälp under kandidatarbetets gång:

Kristofer Bengtsson för handledning, hjälp och stöd genom hela projektet.

Per Nyqvist för hjälp med olika programtekniska delar, felsökning i labbet samt utbildning i robotik.

Hans Sjöberg för hjälp vid montering av PLC.

Jan Bragee och *Reine Nohlborg* för assistans vid tillverkning av ny hårdvara.

Andreas Jonsson och *John Selander* på Volvo för intressant guidning och för PLC-kod.

Matilda Anulf, Pontus Johansson, Simon Kardell, Erik Norén
Göteborg, Juni 2015

Innehåll

Tabell 1: *Ordlista*

Ordlista	**Förklaringslista**
ABB	Företag som tillverkar industrirobotar
C#	Objektorienterat programmeringsspråk
DAG	Directive Acyclic Graph
FBD	Function Block Diagrams
HMI	Human Machine Interface
IL	Instruction List
KUKA	Företag som tillverkar industrirobotar
PLC	Programmable logic controller
PSL	Production System Laboratory
RFID	Radio Frequency Identification
SFC	Sequentiall Function Chart
SOP	Sequence of Operations
ST	Structured Text
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
XML	Extensible Markup Language

1

Introduktion

I dagens globaliserade samhälle är konkurrensen inom industrin större än någonsin, vilket bidrar till att företagen strävar efter att minska sina kostnader [1]. Detta kan åstadkommas på många sätt. Några exempel är massproduktion, effektivisering av produktionen, minskning av ställtider samt snabb anpassning till marknadens snabbt växlande behov och krav [1, 2].

Ett automatiserad produktionssystem är i de flesta fall det mest effektiva alternativet vid massproduktion, detta tack vare maskiners precision, snabbhet och kontinuitet [3]. Automatiserade produktionssystem har även sedan sin införsel på marknaden spelat en betydande roll för den ökade massproduktionen [4].

I takt med att produktionssystemen blir alltmer automatiserade ökar därför antalet maskiner och dess kringutrustning. För att de ska kunna fungera tillsammans behövs något som styr dem på en övergripande nivå. Många maskiner i ett produktionssystem styrs därför av en PLC, *Programmable Logic Controller*, som är en typ av dator som ofta används inom industrin för styrning av olika processer [5].

PLC har använts under många år inom industrin men trots det har få ändringar gjorts i sättet de programmeras på. Det sker mycket forskning om PLC och olika sätt att implementera PLC-program. Då industrin i regel är konservativ angående förändringar hinner de inte med i utvecklingen [6].

Inom de flesta företag förekommer så många styrprogram att det krävs standarder för att alla delar ska kunna fungera smidigt ihop, vare sig det är standarder som vuxit fram med tiden eller sådana företaget aktivt infört [7]. Den standard som finns på Volvo Car Corporation innefattar ett regelverk hur PLC-programmen ska utformas med krav på till exempel struktur, namnsättning, adresser, kommunikation och Human Machine Interface [8]. Fortsättningsvis kommer Volvo Car Corporation att benämnas som enbart Volvo.

På företag som Volvo hyrs ofta kompetens in för att skriva PLC-programmen givet en viss standard som Volvo skapat [9]. I PLC-programmen är det viktigt med återanvändning av kod och därför har de inhyrda kompetenserna tillgång till ett bibliotek med kodkomponenter vid utveckling av nya PLC-program [9]. Det är även viktigt att PLC-programmen är lättförstådda så att operatörer kan felsöka koden, vilket avspeglar sig i både struktur och namnsättning hos Volvo. Arbetet att utforma styrlogik och PLC-program är generellt väldigt tidskrävande och ineffektivt och därav har forskning bedrivits för att effektivisera arbetet [10]. Det som i synnerhet kan förbättras är just återanvändning av repetitiva kodsegment.

I PSL, *Production System Laboratory*, på Chalmers Tekniska Högskola finns ett produktionssystem styrd av en PLC som används för utbildnings- och forskningssyften. Många kandidatarbeten har utförts i cellen med avseende att optimera den ur olika perspektiv som energi och tid [11].

1.1 Syfte

Projektet syftar till att utvärdera och föreslå hur ett koncept från närliggande forskning som bedrivits på Chalmers Tekniska Högskola kan implementeras i Volvos nuvarande standard. För att nå dit kommer de båda mjukvarustrukturerna att behöva anpassas för att därefter skapa en ny standard. Det ska sedan undersökas om möjligheterna att automatiskt generera repetitiv kod för att spara tid i utvecklingprojekt. Konceptet innehållande en ny standard och automatiskt genererad kod ska sedan implementeras och verifieras i PSL. För att detta ska vara möjligt kommer även produktionscellen att behöva anpassas. Resultatet av implementeringen kommer vara ett mjukvarukoncept som ska kunna fungera som en demonstrator för att påvisa potentialen i Chalmers Tekniska Högskolas operationskoncept samt inom automatisk kodgenerering för industrin.

2

Automatiserade produktionssystem

Robotarnas intåg i industrin har bidragit till alltmer automatiserade produktionssystem. Detta innebär att det är framförallt självgående elektromekaniska produkter, som robotar och transportband, som tillverkar produkter. Produktion med automatiserade produktionssystem har blivit regel snarare än undantag inom framförallt massproduktion [3]. Produktionssystemen är därför numera datorstyrda och det måste då finnas en dator med tillhörande mjukvara i närheten av cellen [12].

Mjukvarukostnaden för en tillverkningsprocess kan uppgå till 30 % av den totala kostnaden av tillverkningsprocessen [13]. För att minimera denna kostnad kan företag inom industrin investera i mjukvara som både är lätt att förändra och snabb att implementera [2]. Även fast detta redan är känt av de flesta företag förblir fortfarande mjukvaran specifikt designad efter en produkt [13]. Det som vanligen kontrollerar ett automatiserat produktionssystem är en PLC, som är en typ av styrenhet. Anledningen till att PLC:n oftast används för ändamålet är för att den är robust, har lång livslängd och är mycket flexibel [14].

Detta kapitel ger en övergripande introduktion i hur automatiserade produktionssystem samt dess komponenter fungerar. Det kommer beskrivas hur en PLC fungerar samt en introduktion till dess olika programmeringsspråk. Även operationer och dess sekvensiella beteende kommer att behandlas då detta är viktigt för att få en förståelse för konceptet som utvecklats inom projektet. För att beskriva ett produktionssystems olika tillstånd samt hur olika operationer relaterar till varandra kommer även grafer att introduceras.

2.1 PLC

Tekniken med PLC utvecklades under 70-talet för att kontrollera maskiner och således möjliggöra en datorstyrd och säker produktion. En PLC används även flitigt utanför produktion och industriella sammanhang som exempelvis dörrar i bussar och trafikljus för både tåg och bilar. En PLC utför sina instruktioner i cykler, så kallade scancykler [13]. Den utför detta genom att läsa av digitala och analoga signaler, som vanligen kommer från olika knappar och sensorer. Signalerna behandlas därefter med hjälp av matematiska och logiska uttryck som programmeraren valt. Den processade signalen går därefter till utgången som blivit definierad där den kan vara inkopplad till olika lampor, motorer eller liknande [15, 14].

Koden i en PLC återanvänds ofta genom att vara komprimerade i block som anropas. För att få bättre överblick och lättare kunna återanvända kod kan blocken sparas i bibliotek som kan anropas från övriga delar av programmet [9].

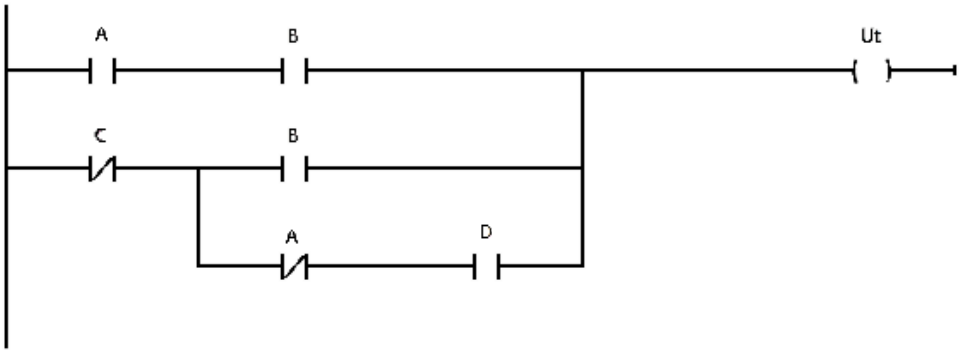
2.1.1 Programmeringsspråk för PLC

Det finns som tidigare nämnts olika programmeringsspråk för en PLC. Detta avsnitt behandlar *Ladderlogik*, IL (*Instruction list*), FBD (*Function Block Diagram*) och SFC (*Sequential Function Chart*). Ett viktigt ord i detta avsnitt är Network, som är en slinga där en eller flera insignaler bearbetas och bildar en eller flera utsignaler [6].

2.1.1.1 Ladderlogik

Ladderlogik är ett grafiskt programmeringsspråk för att illustrera logiska uttryck, se ekvation (2.1) och figur 2.1 för jämförelse av ett logiskt uttryck skrivet i ladderlogik och med boolesk algebra. Ett sätt att illustrera relälogiken som PLC:n ersatte är med ladderlogik. Stommen för ladderlogiken består av två vertikala linjer och minst en horisontell sladd, även kallad rung eller Network. Den vänstra vertikala linjen symboliserar strömförsörjaren och den högra vertikala linjen symboliserar jorden som inte alltid illustreras i alla programmeringsmiljöer. För att det ska bli en sluten krets krävs då att strömförsörjaren och jorden blir ihopkopplade. På detta Network sätts därefter olika kontakter som i figur 2.1 motsvarar ett booleskt uttryck Ut. Kontakten utgör en operator som sluter kretsen då dess operand, även kallat variabel, blir sann. Det finns även kontakter som kan användas för negerade variabler samt sluta kretsen i pulser vid upp- och neråtgående flank för operanden.

$$Ut := (A \wedge B) \vee (\neg C \wedge (B \vee (\neg A \wedge D))) \tag{2.1}$$



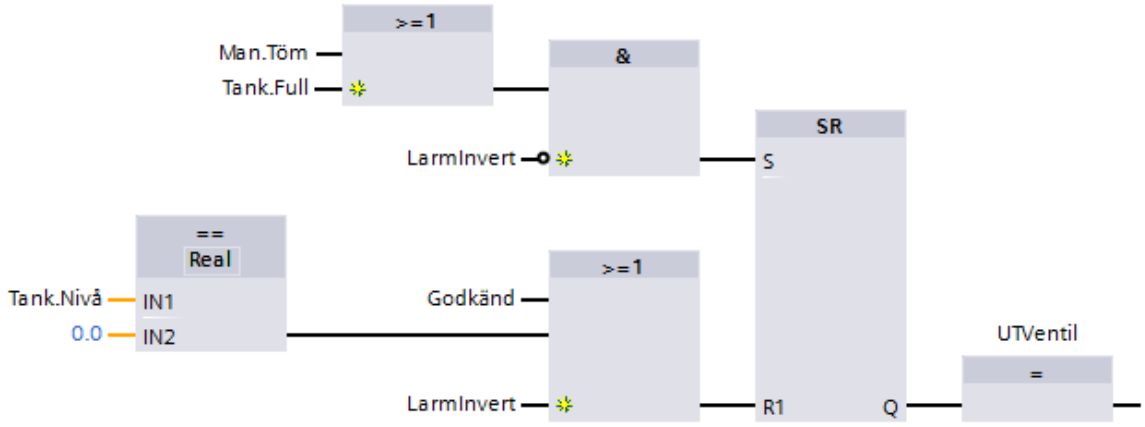
Figur 2.1: *Ladderlogik*

Dessa kontakter kan kopplas i serier eller parallellt beroende på vilket booleskt uttryck som ska erhållas. Närmast jorden ska en värdeadderande operator placeras, som exempelvis kan utgöras av en spole. En spole är en operator som tilldelar eller håller en boolesk variabel vid ett specifikt värde [13, 7].

2.1.1.2 FBD

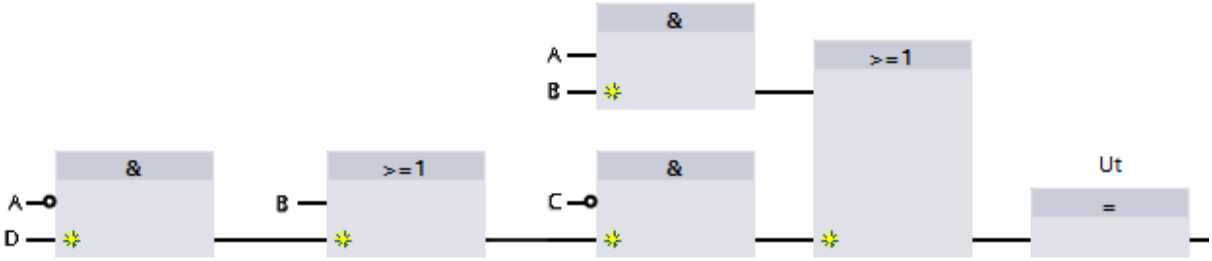
FBD, *Function Block Diagram*, illustrerar när signaler ska till och från olika kontrollblock. Ett *function block*, funktionsblock, kan vara både logiska och matematiska funktioner men även så kallade flip-flops som används för att aktivera och deaktivera signaler [7]. När

ett antal funktionsblock sätts samman bildas ett FBD. Ett funktionsblocks ingångar är på vänster sida och dess utgångar på höger. I FBD finns möjlighet till återkoppling av signaler, genom att koppla samman en utgång till en ingång på tidigare förekommande block. Som en följd av återkopplingen kan det vara svårt att se signalflödet. Ett exempel på FBD som kontrollerar en tanks volym ses i figur 2.2. Tanken töms när den är full eller en operatör väljer att tömma den. I systemet finns även ett larm, som skickar en falsk signal när den larmar, som ska öppna och hålla utventilen öppen om det är något som gått fel. När tanken är tom eller operatören väljer att stänga den ska utventilen stängas [7].



Figur 2.2: *Kontrollprogramm för en vätsketank*

I figur 2.3 illustreras hur ett FBD kan se ut för att erhålla det logiska uttrycket i ekvation (2.1).



Figur 2.3: *Ekvation* (2.1) *som funktionsblocksdiagram*

För både FBD och ladderlogik gäller följande fyra regler [7]:

- För att ett Network ska kunna tilldelas ett värde måste det logiska uttrycket för samtliga insignaler blivit värderat.
- Ett Networks utsignal ska inte tilldelas något värde innan samtliga interna vägar genom kretsen blivit värderade.
- Alla utsignaler ska blivit uppdaterade innan ett Network kan ses som färdigt.
- När data överförs från ett Network till ett annat, ska de värden som kommer från det tidigare Network framställas genom samma utvärdering. Det andra Network ska inte börja utvärderas, tills alla värden från den första kretsen är tillgänglig.

2.1.1.3 ST

Ytterligare ett sätt att skriva kod till PLC är att använda ST, *Structured text*. Detta programmeringsspråk är väl utformat för aritmetiska operationer och hantering av data. Till skillnad från tidigare nämnda språk, där signalen följde en given väg och exekverades i samma ordning, är ST uppbyggt med en prioriteringsordning som även används i matematiken [7]. Alltså har parenteser högst prioritet följt av funktioner och den lägsta prioriteten motsvarar booleska uttryck. Då flera matematiska operationer har samma prioritet att exekvera, uttrycks det då från vänster till höger. ST är lättläst och det finns möjlighet att kommentera mitt i koden. Det finns även möjlighet att iterera kod med hjälp av bland annat for-loopar och while-loopar. Ett exempel på ST kan ses i figur 2.4 där koden beskriver ett larm som aktiveras när effekten från ett system ligger utanför intervallet mellan 10 och 15 watt.

```
U,I,P : REAL;
Alarm : INT;

P := U*I

IF P > 15 THEN
        msg:='Alert';
        Alarm:=2;

ELSEIF P < 10 THEN
        msg:='Alert';
        Alarm=1;

ELSE
        Alarm=0;
```

Figur 2.4: *Illustration av ett larm skrivet med ST*

2.1.1.4 IL

Kod som skrivs med IL, *Instruction list*, har en struktur som är väldigt lik det maskinära språket Assembler, som är ett språk för lågnivåprogrammering [7]. Det är således lätt för en PLC att kompilera program som skrivs i IL. Fördelen med att vara nära maskinkoden är att möjligheten att optimera de kritiska programdelarna är bättre än vid högnivåprogrammering. En nackdel med att vara så nära maskinkod är att det kan vara svårt att följa flödet som koden exekveras i.

Programspråket IL består utav Labels, Operatorer och Operander [7]. Ett exempel på IL kan ses i figur 2.5, som utvärderar det logiska uttrycket i ekvation (2.1). I exemplet är *Start* en label, vilket motsvarar en etikett som programmeraren kan sätta på ett stycke kod. Detta gör att koden kan anropas från andra delar av programmet. Den mellersta kolumnen är operatorerna som innehåller instruktioner för det som ska utföras. Slutligen

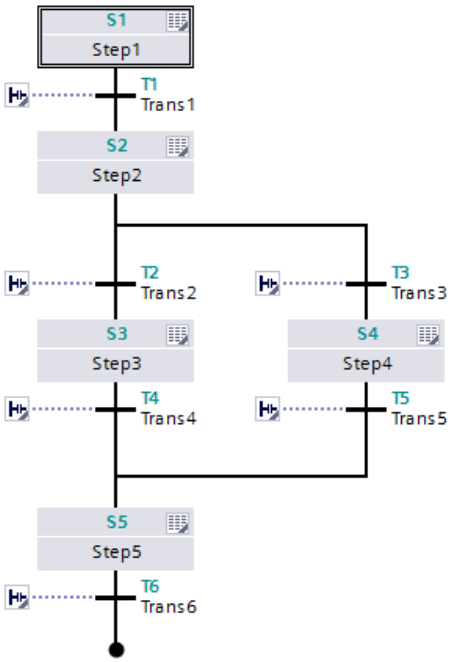
blir operanderna, variabler eller konstanter, som är placerade i den högra kolumnen utvärderade av operatorerna.

```
Start    LDN     A
         AND     D
         OR      B
         ANDN    C
         ST      Temp
         LD      A
         AND     B
         OR      Temp
         ST      Ut
```

Figur 2.5: *Illustration av ekvation* (2.1) *skrivet som IL*

2.1.1.5 SFC

Det sekvensiella tänkandet som används för att skriva PLC-kod har resulterat i att SFC, *Sequential Function Chart*, vuxit fram [7]. Några av de största anledningarna till att SFC har vuxit fram är att det ger en god överblick över programmets olika exekveringstillstånd. I figur 2.6 illustreras en SFC där varje tillstånd symboliseras av en ruta, även kallat steg. Ett exempel på ett steg kan vara att öppna en ventil. De kortare linjerna indikerar olika övergångar som kan ses som villkor som behöver uppfyllas för att byta tillstånd. Övergångarna och stegen är i sin tur programmerade med exempelvis Ladderlogik, FBD eller ST [7].



Figur 2.6: *Illustration av kod skrivet med SFC*

2.2 Operationer och sekvenser

Operationer är ett sätt att beskriva händelser och kan beskrivas i följd och dessa följder kan vara i beroende av varandra. Då en PLC programmeras kan viss kod vara beroende av annan kod och då finns möjlighet att beskriva programmet som ett antal operationer. Då ett större antal operationer används kan grafer användas för att beskriva samspelet mellan dessa, se avsnitt 2.3.

2.2.1 Operationer

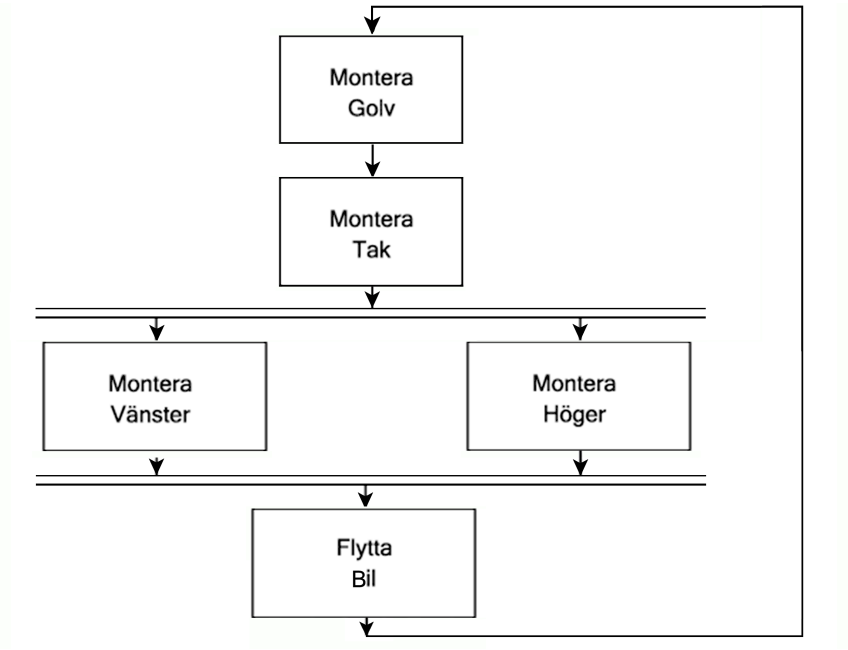
För att beskriva händelser, inte bara inom industrin, kan operationer användas vilket visas i figur 2.7 [16, 14]. En operation har ett start- och ett slutvillkor, så kallade *Precondition* respektive *Postcondition*. För att starta operationen *Fyll glas* måste ett glas först existera vilket betyder att Precondition, startvillkoret, måste uppfyllas. För att starta denna operation måste således *Finns glas?* vara sant. Operationen exekverar då tills dess Postcondition, slutvillkor, uppfylls vilket betyder att *Är glaset fullt?* måste vara sant för att operationen ska ses som genomförd.

En fördel med att införa operationer i systemkoden, är att de är baserade på Automata, som är en modell för att beskriva diskreta eventsystem [16, 17]. Automata kan bland annat användas för tidsoptimering och formell verifikation [14]. Det möjliggör även användning av formella metoder vid systemdesignen så som hantering av oförutsedda återstarter offline [17, 3].

Figur 2.7: *Operation för att fylla ett glas med vatten*

2.2.2 Operationssekvenser

Då ett antal operationer samverkar är det lätt att tro att operationerna sker parallellt, vilket inte alltid är fallet. Operationer består av Pre- och Postconditions och det är dessa villkor som avgör om operationerna kan utföras parallellt eller i sekvens. För att skapa överblick över operationernas samspel kan operationssekvenser, även kallat SOP, *Sequence of Operations*, användas [14]. SOP är ett grafiskt språk som kopplar samman operationer med linjer, pilar och booleska uttryck. I figur 2.8 illustreras monteringen av en modellbil med fyra komponenter i form av en SOP. Pilarna indikerar sekvensen som operationerna följer och de dubbla horisontella linjerna betyder att operationerna exekverar parallellt.



Figur 2.8: *SOP för att montera en bil*

2.3 Grafer

Vid stora och komplexa system bestående av en stor mängd operationer, kan grafer användas för att illustrera de relationer som råder mellan de olika operationerna. I en sådan graf utgörs noderna av systemets olika tillstånd och övergångarna av operationer.

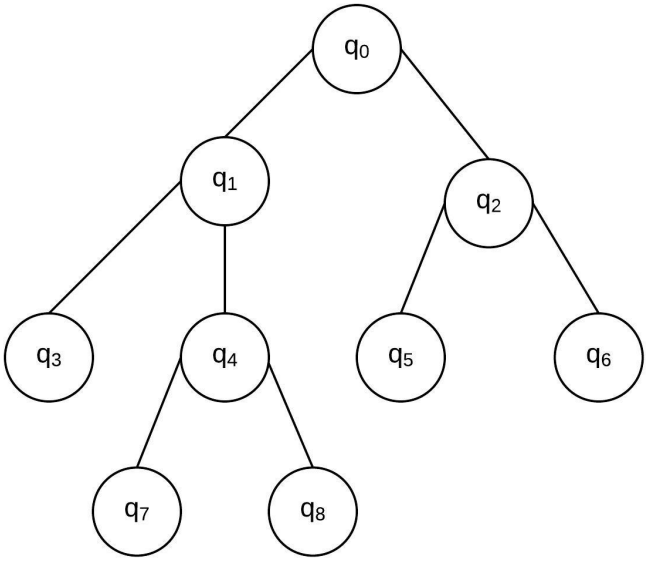
En graf, $\mathbb{G} = \{Q,T\}$, är en metod som används för att illustrera parvis relationer för en mängd av objekt. Grafen består av en mängd noder Q samt en mängd övergångar V som förgrenar de olika noderna. Grafen kan således representeras av en samling övergångar. En övergång representeras av $t \in T$ där $t = \{q_0, q_1\} \wedge t \subseteq Q$. Då grafen symboliserar ett operationsflöde utgör operationerna övergångarna mellan systemets olika tillstånd som således utgör noderna i grafen.

För en vanlig enkel graf är övergångarna symmetriska det vill säga att för varje övergång $t = \{q_0, q_1\}$ finns en motsvarande övergång $t = \{q_1, q_0\}$.

Ett enkelt exempel på en graf är ett spårvagnsnät där noderna utgörs av hållplatser samt övergångarna av rälsen som är dragen mellan hållplatserna.

2.3.1 Träd

Ett vanligt sätt att beskriva hierarkier i datastrukturer är med hjälp av så kallade träd. Ett träd kännetecknas av att alla noder i grafen är sammankopplade, grafen innehåller ej några cykler samt att en utav grafens noder ej har några ingående övergångar och utgör därav roten för trädet [18]. I figur 2.9 ges ett exempel över en graf av enklare trädstruktur.

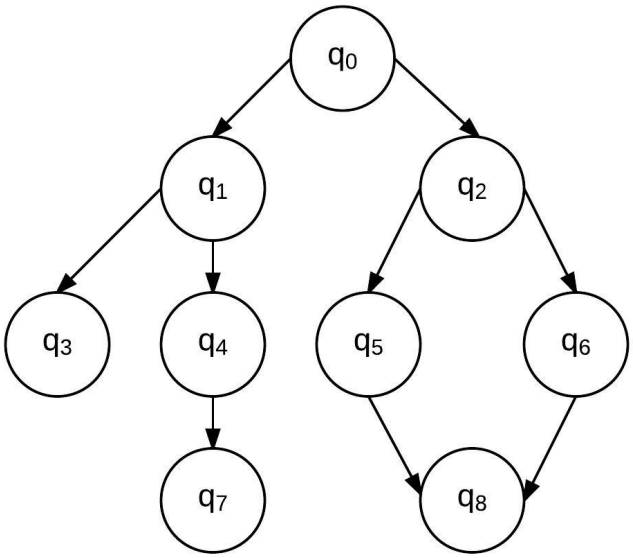


Figur 2.9: *Illustration av ett träd*

2.3.2 Directed Acyclic Graph

Till skillnad från övergångarna i vanliga grafer är övergångarna i riktade grafer asymmetriska, vilket således leder till att det bara går att färdas i en riktning inom grafen. Om en riktad graf inte har några cykler samt att dess övergångar sker i topologisk ordning är grafen en DAG, *Directed Acylic Graph* [18].

En topologisk ordning av en graf uppfylls om och endast om $Q = \{q_i,q_{i+1},...,q_n\}$ där $\forall t \in T$ där $t = (q_i,q_j) \wedge i < j$. Ett exempel av en DAG ges i figur 2.10.



Figur 2.10: *Illustraton av en Directed Acyclic graph*

3

Problemdefinition

Industrin har på många punkter inte uppdaterats och följt med i utvecklingen inom styrning av produktionssystem vilket gör att många företag går miste om den potential som forskning har möjliggjort [6]. Många företag lever därför kvar med äldre standarder och sättet styrenheterna programmeras på har inte uppdaterats och följt med i utvecklingen. Dock har forskning inom detta område bedrivits på Chalmers Tekniska Högskola och det behöver därför utvecklas en ny mjukvarustruktur som ska möjliggöra en implementering utav Chalmers operationskoncept ute i industrin, där Volvo kommer vara det företag som granskas [16].

Undersökningar av Volvos tillvägagångssätt vid programmering i PLC har utförts av Oscar Ljungkrantz och Knut Åkesson på institutionen för Signaler och System på Chalmers Tekniska Högskola år 2007 [9]. Efter åtta år finns det risk att ändringar har införts och att den utförda undersökningen inte längre till fullo stämmer överens med hur det ser ut i dagsläget.

Med hjälp utav en tydlig standard kan mycket av befintlig kod användas till nya projekt. Denna kod är dock ofta repetitiv vilket gör att en stor del av utvecklingstiden och således utvecklingskostnaderna läggs på att iterativt skriva samma kod, tid som istället skulle kunna läggas på mer värdeadderande aktiviteter. Därför finns det stor potential i att undersöka hur automatgenererad kod kan underlätta i utvecklingsprocessen. Med hjälp av automatiskt genererad kod kan fel orsakade av mänskliga faktorer reduceras samt säkerställa att önskade standarder upprätthålls [19]. Företagsstandarder skiljer sig ofta både vad gäller programstruktur samt programmeringsspråk. En tydligt strukturerad standard är dock av väsentlig betydelse för att kunna generera kod på ett önskvärt sätt [19].

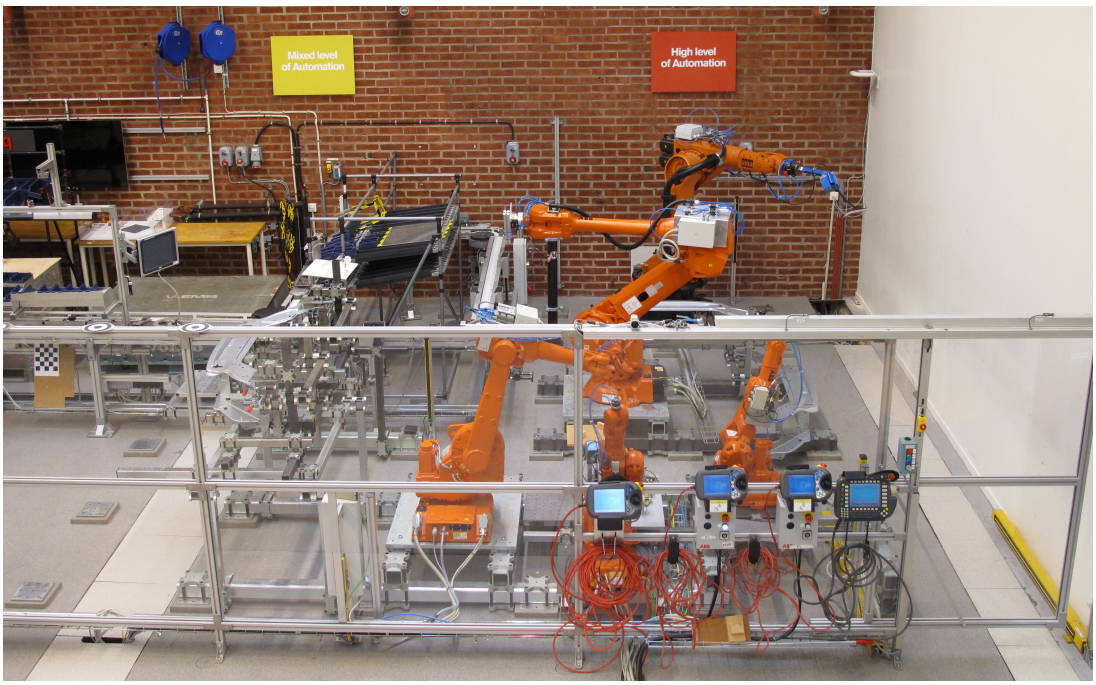
För att lösa detta problem ska det utvecklas ett nytt koncept som sedan ska implementeras i PSL för verifiering. Det nya konceptet består utav en ny mjukvarustruktur i form utav en ny standard för PLC-logik som är en kombination utav en nuvarande företagsstandard och forskning. Det ska även skapas förutsättningar för att underlätta utvecklingsprocessen av den nya mjukvarustrukturen genom automatisk kodgenerering för att undersöka och analysera potentialen för implementering i industrin. Detta koncept kommer sedan att fungera som en demonstrator för industrin. För att uppnå målet med en fungerande demonstrator behöver även ett antal uppdateringar av hårdvaran utföras och konstrueras.

4

Produktionssystemet

Produktionsystemet som detta koncept har utvecklats och utvärderats i är en del av PSL, *Production System Laboratory*, som är en resurs på Chalmers Tekniska Högskola. PSL fungerar i dess nuvarande utformning som ett redskap för verifiering och framtagning av forskningsresultat men även som ett inlärningsverktyg och för att förmedla kunskap i kurser inriktade mot automation och produktion. Produktionssystemet kommer fortsättningsvis hänvisas till som produktionscellen eller bara cellen och kan ses i figur 4.1. Produktionscellens huvuduppgift är att montera en modellbil med hjälp av transportband, fixturer och industrirobotar.

För att kunna visa och utvärdera det som skett inom projektet är det viktigt att det finns en fungerande cell att testa och implementera det framarbetade konceptet i. För att få produktionscellen att fungera för det nya konceptet måste ny hårdvara konstrueras och befintlig hårdvara förbättras. I följande kapitel kommer produktionscellen som helhet, dess förutsättningar samt de uppdateringar som krävts för att utföra projektet att beskrivas.

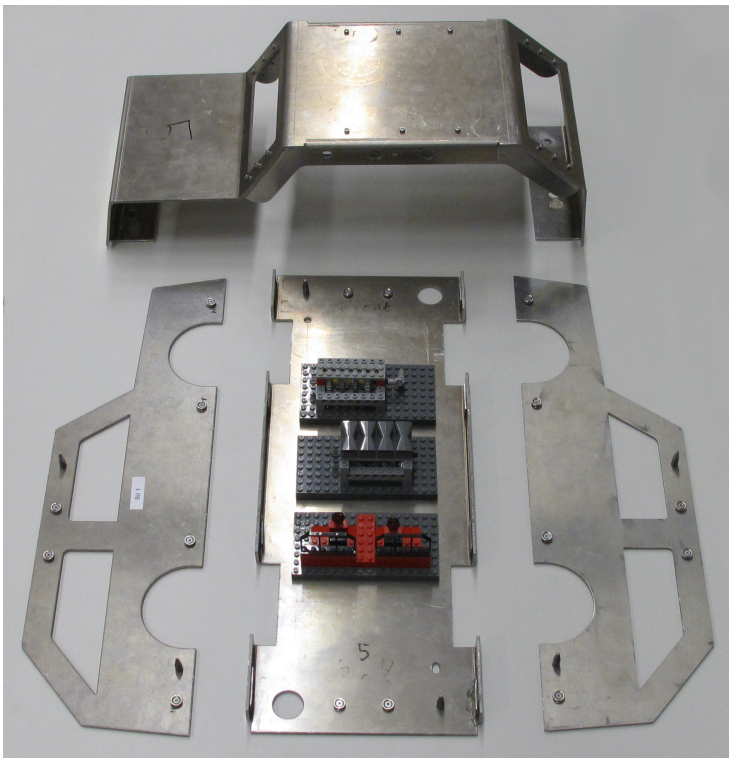


Figur 4.1: *Produktionssystemet i PSL*

4.1 Modellbil

Den produkt som i dagsläget är avsedd för produktionscellen är en modellbil. Denna modellbil består utav sju komponenter som kan ses i figur 4.2 och komponenterna är:

- **Karossdelar som är tillverkade i plåt**
  - Golv
  - Tak
  - Vänstersida
  - Högersida
- **Moduler som är tillverkade i lego**
  - Motor
  - Säte
  - Växellåda



Figur 4.2: *Modellbilens komponenter*

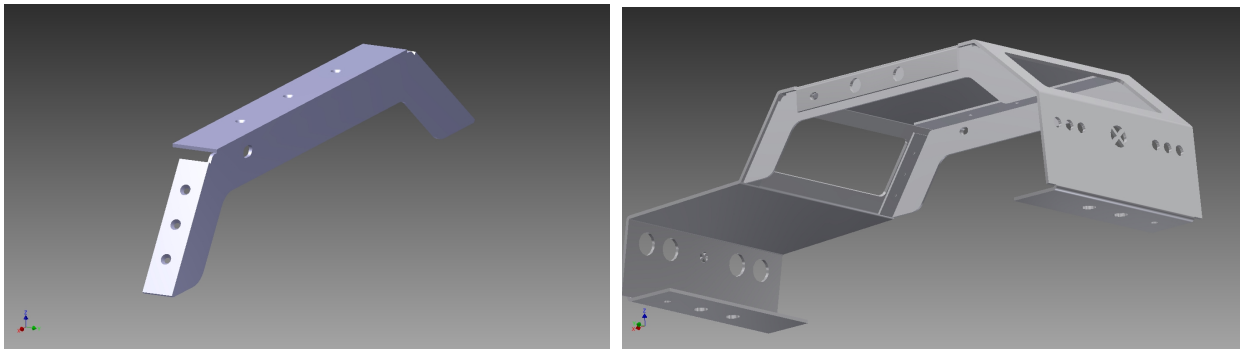
Plåtkomponenterna monteras med hjälp av magneter för att möjliggöra iterativa monteringscykler med minimalt slitage på fästningspunkterna. Styrpinnar är monterade på arbetsstyckena för att dirigera dessa i rätt position vid montering. Legomodulerna monteras genom att placeras i stansade hål i golvet.

Den befintliga modellbilens konstruktion behövdes förbättras med avseende på hållfasthet och utmattning vilket kräver en mer robust konstruktion. Modellbilens tak, vilket är en av karosskomponenterna i plåt, har deformerats på grund av åtskilliga monteringar under flertalet projekt och på grund av detta har ett problem med deformationer växt fram. Deformationerna på modellbilens komponenter har bidragit till att dimensionerna på modellbilen förändrats vilket gör det svårt att köra produktionscellen med ett och samma robotprogram. Robotprogrammen kräver en fin noggrannhet med mycket små toleranser vilket gör att monteringen ofta misslyckas på grund av modellbilens skiftande dimensioner.

Deformationerna har uppkommit då roboten lagt för stor kraft på takets ovansida vid plockning och modellbilen har inget som kan ta upp denna kraft vilket gjort att vinklarna ändrats.

4.1.1 Resultat av förbättrad modellbil

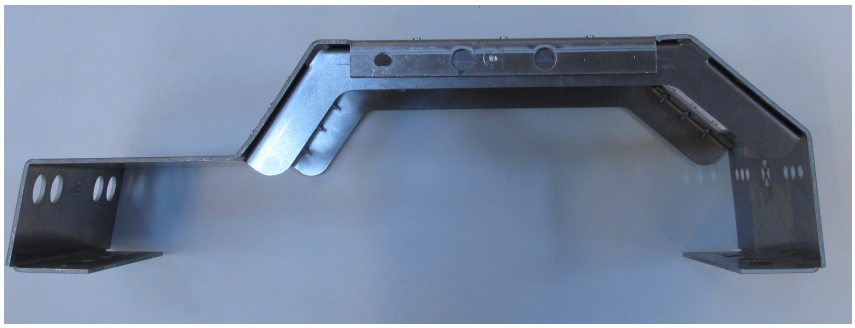
En profil designades i *Autodesk Inventor* som är en virtuell 3D-miljö för konstruktion och resultatet kan ses i figur 4.3a. Profilen fästs längs insidan av modellbilens tak på båda sidorna, som visas i figur 4.3b, och med hjälp av denna profil fördelas kraften från roboten ner i profilen och förstärker modellbilens tak vilket bidrar till mindre deformation och längre hållbarhet. Detta leder i sin tur till att modellbilens dimensioner behålls vilket underlättar vid exekvering i produktionscellen då noggrannheten stiger och toleranserna minskar.



(a) *Profil* **(b)** *Profil monterad i taket*

Figur 4.3: *Profilen designad i 3D-miljö*

Profilen konstruerades och tillverkades i stål i Chalmers Tekniska Högskolas prototypverkstad som är en resurs för olika forsknings- och utbildningsprojekt. Materialvalet låg till grund för att dels utnyttja materialets goda hållfasthetsegenskaper men även dess magnetism då komponenterna fästs med magneter. Komponenten skars ut i en vattenjet för hög noggrannhet med små toleranser. Komponenten bockades sedan till sin tänkta form i en bockningsmaskin vilket även ökade de redan goda hållfasthetsegenskaperna. Profilen monterades sedan i modellbilen och resultatet kan ses i figur 4.4 och ritningarna kan ses i Appendix E.



Figur 4.4: *Framtagen profil*

4.2 Transportfixturer

En fixtur är en anordning för att fixera ett arbetsstycke under en sekvens av operationer i en serietillverkning. Operationerna kan exempelvis vara robotoperationer, olika operationer för skärande bearbetning eller montering. En fixturs syfte är att fixera ett arbetsstycke på samma sätt under hela serietillverkningen, vilket bidrar till att de olika produkterna blir lika och är således av vikt i kvalitetssynpunkt [20].

För att fixera modellbilens komponenter används en transportfixtur. Den befintliga konstruktionen består utav fyrkantsrör som är sammansvetsade på en platta och tillverkade i aluminium. För att göra den anpassningsbar till modellbilens alla komponenter finns tillhörande moduler för varje komponent. Dessa fästs på fixturen med hjälp av magneter som är placerade på fixturen.

De befintliga transportfixturerna är tillverkade genom tillverkningsmetoder som inte är optimala vid hårda krav på noggrannhet vilket har bidragit till att problem i produktionscellen har uppkommit. Dimensionerna på de olika transportfixturerna avviker från varandra vilket gör, precis som i modellbilens fall, att robotarna inte kan utföra sina operationer med samma robotprogram då de kräver en hög noggrannhet. För att åtgärda detta problem behövde en helt ny design konstrueras och tillverkas.

4.2.1 Fixturkonstruktion

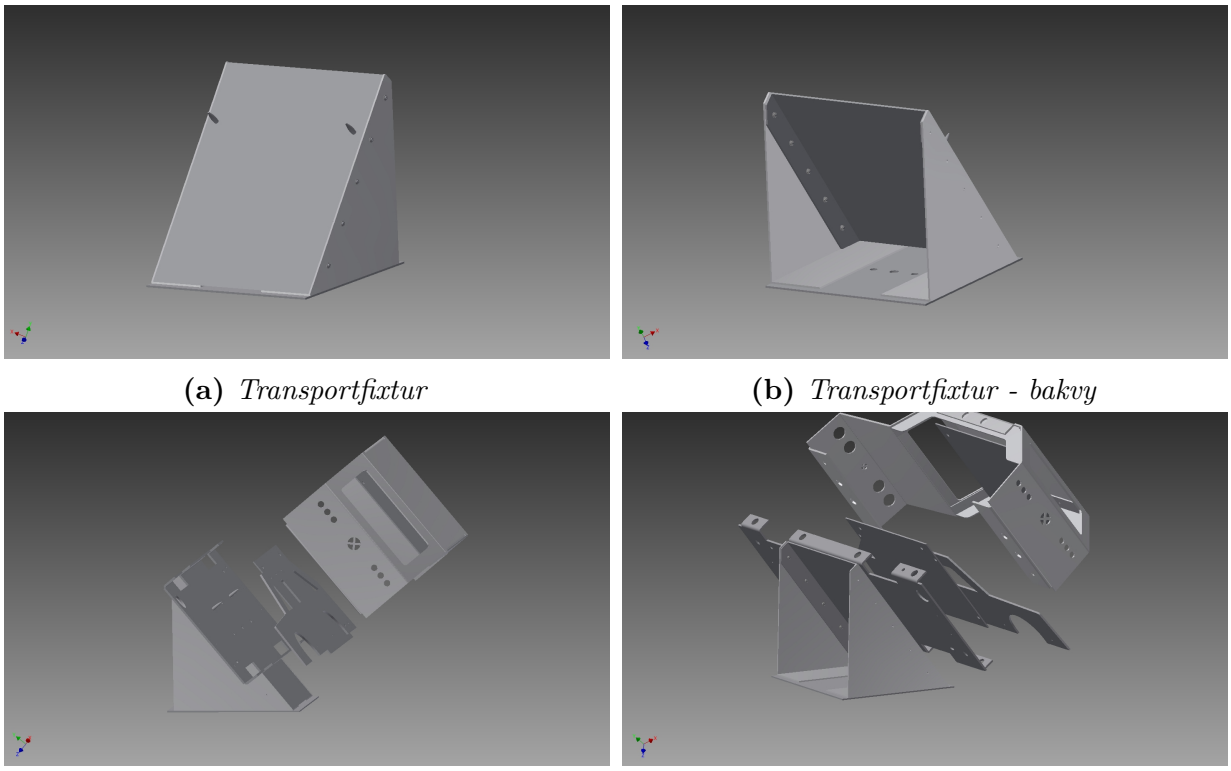
En fixtur består oftast utav ett antal komponenter som styrpinnar och ytor för att föra ner arbetsstycket och placerade det på rätt plats vid varje iteration. Det finns också stöd för att undvika deformation på arbetsstycket vid exekvering samt klämmor för att låsa fast arbetsstycket. Själva fixturkroppen håller alla dessa komponenter samman [20] och tillsammans låses objektet i alla dess frihetsgrader. Trots att flexibilitet ofta är en faktor som vill uppnås i industrin tillverkas fortfarande många fixturer som endast är kompatibla med en produkt. Forskning bedrivs angående att implementera modularisering i fixturdesign för att öka flexibiliteten och standardiseringen [21]. Istället för att konstruera en fixtur för varje arbetsstycke tillverkas istället en fixtur med olika moduler för ett bredare spektra av arbetsstycken.

Vid tillverkning av en fixtur finns det fem stycken designkriterier [20]:

- Total fixering av arbetsstycket, alltså helt låst i alla frihetsgrader. Ska kunna stå emot krafter och moment i alla riktningar som arbetsstycket utsätts för under tillverkningsprocessen.
- Noggrann lokalisering och positionsfel skall minimeras
- Jämn kraftfördelning på fixturen och arbetsstycket
- Ingen interferens, eller störning, mellan fixturkomponenter och verktyget som arbetar på fixturen
- Fixturen ska vara lätt att använda, exempelvis lätt fixering och borttagning av en komponent från fixturen

4.2.2 Resultat av konstruktion utav ny transportfixtur

Även transportfixturen designades och konstruerades i *Autodesk Inventor* och resultatet kan ses i figur 4.5a & 4.5b. Transportfixturen är en fixtur som består utav sex komponenter, en grundplatta, två sidor, en mellanplatta samt två styrpinnar. För att möjliggöra en flexibel fixtur konstruerades två olika moduler som tillsammans med fixturen kan transportera alla modellbilens plåtkomponenter, detta för att undvika att behöva en fixtur för varje modellbilkomponent. Modulerna kan ses i figur 4.5c & 4.5d. Med hjälp av styrpinnarna kan modulerna fästas på fixturen där sedan modellbilens komponenter monteras och fixeras.

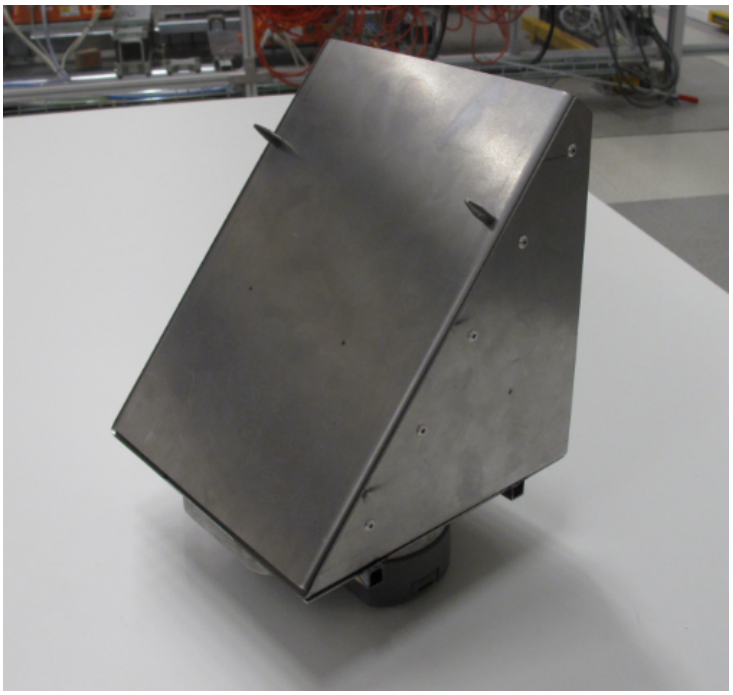


(a) *Transportfixtur* **(b)** *Transportfixtur - bakvy*

(c) *Transportfixtur med moduler* **(d)** *Transportfixtur med moduler - bakvy*

Figur 4.5: *Fixturen med och utan moduler designad i 3D-miljö*

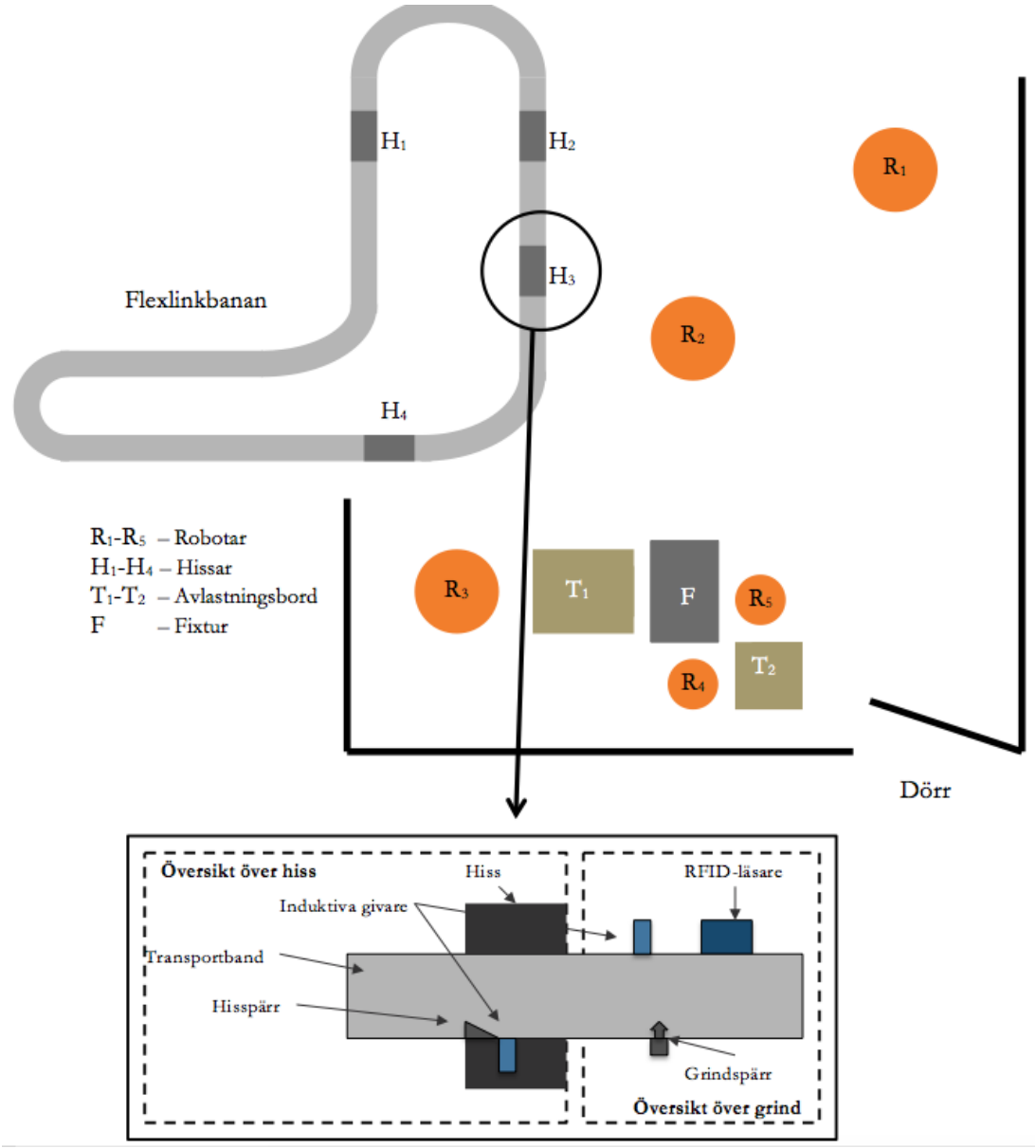
Precis som modellbilens profil tillverkades fixturens komponenter med en vattenjet i prototypverkstaden för att erhålla hög noggrannhet. Dessa bockades sedan till rätt form och monterades. Resultatet kan ses i figur 4.6 och ritningarna kan ses i Appendix E.



Figur 4.6: *Resultat av transportfixtur*

4.3 Produktionssystemets utformning

Produktionssystemet i PSL är ett komplett produktionssystem som fungerar som en enhet genom styrning av en PLC. Produktionsscellen kan ses i figur 4.7 och innehåller ett transportband av märket Flexlink, som i fortsättningen kommer benämnas som ”Flexlinkbanan”, fem industrirobotar, två avlastningsbord samt en fixtur.



Figur 4.7: *Översikt i produktionscellen, figuren är inte skalenlig*

4.3.1 Flexlinkbanan

Flexlinkbanan är produktionssystemets transportband och det står för all materialförsörjning till produktionscellen genom manuell påläggning. Flexlinkbanan är utrustat med flertalet olika komponenter så som sensorer för att exempelvis kunna detektera arbetsstycket samt hissar och grindar för att kontrollera de objekt som transporteras på flexlinkbanan. För att transportera objekt måste flexlinkbanans paletter, som beskrivs i avsnitt 4.3.1.3 användas.

4.3.1.1 Hissar

På flexlinkbanan sitter fyra hissar vars uppgift är att hissa upp och fixera komponenter för att underlätta för robotar att lyfta in dessa i produktionscellens arbetsområde och för

operatörer att förbereda komponenter. Varje hiss består utav tre komponenter vilket kan ses i figur 4.7 och de är:

- **Induktiv givare**
  - En induktiv givare är en elektrisk lägesgivare som kan detektera metalliska objekt utan att röra vid objektet [5]. Givaren kan således ge två olika utsignaler, sant & falskt som även kan benämnas som 1 & 0, där sant eller 1 motsvarar detektion och falsk eller 0 motsvarar ingen detektion. Denna signal kan sedan läsas av en PLC.
- **Hisspärr**
  - En spärr som med hjälp av en PLC kan sättas i två lägen, öppen eller stängd. Detta stopp gör att objektet som ska hissas upp stannar vid rätt position på hissen.
- **Hiss**
  - Två cylindrar som med hjälp av pneumatik, även kallat tryckluftsteknik, lyfter upp hissen. Pneumatik är när något utnyttjar gaser, oftast luft, för att överföra energi och åstadkomma rörelse [5].

I hissen fixeras och låses objekten fast och lyfts upp av de pneumatiska cylindrarna.

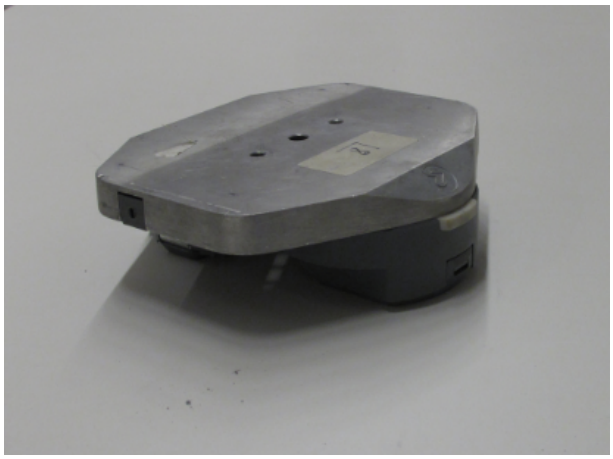
4.3.1.2 Grindar

Tolv stycken grindar är även placerade på flexlinkbanan varav fyra stycken förekommer tillsammans med en hiss. Grindarnas uppgift är att stänga eller öppna passager på transportbandet, vilket gör att antalet komponenter på olika positioner av transportbandet kan kontrolleras. Dessa grindar styrs av en PLC. Varje grind har tre komponenter vilket kan ses i figur 4.7 och de är:

- **Induktiv givare**
- **Grindspärr**
  - Fungerar på samma sätt som hisspärren men grindspärrens funktion är att hindra eller tillåta paletterna att passera grinden.
- **RFID-läsare**
  - RFID står för *Radio-frequency identification* vilket möjliggör att läsa och lagra information genom radiovågor via RFID-tags [5]. Detta kan till exempel utnyttjas för att identifiera vilken komponent som skickas in i ett systemet.

4.3.1.3 Paletter

Allt som ska transporteras i flexlinkbanan måste fästas på så kallade paletter och de visas i figur 4.8. Modellbilens plåtkomponenter fästs på transportfixturen som i sin tur är fästa i en palett. Modellbilens legomoduler fästs däremot direkt på en palett. Paletterna använder sig av RFID-teknik och är försedda med en RFID-tag som sedan kan läsas av en RFID-läsare. Med hjälp av detta kan en specifik komponent identifieras i produktionssystemet.

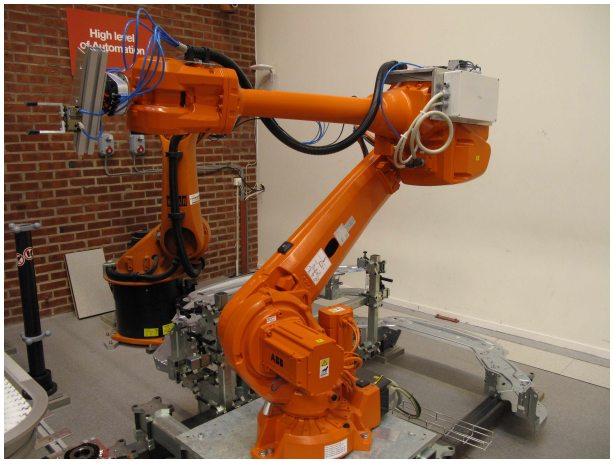


Figur 4.8: *Palett*

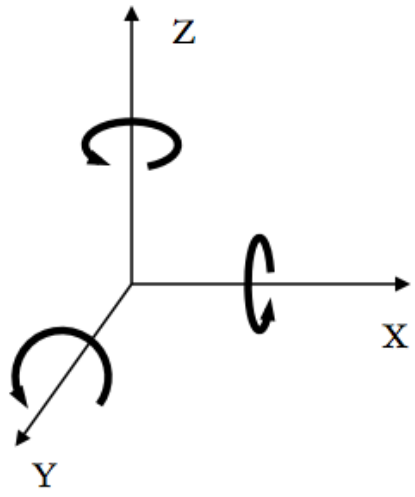
4.3.2 Robotar

De fem industrirobotarna som finns i cellen är i olika storlek varav fyra från tillverkaren ABB och en från KUKA och de är namngivna som robot R1-R5, där R1 är roboten från KUKA och R2-R5 är robotarna från ABB. Dess placering kan ses i figur 4.7. Roboten från KUKA och robot R3 kommer inte användas i detta projekt. Robotarna är alla fästa i flyttbara fundament vilket gör att cellen är flexibel vid förändringar.

Robotarna är uppbyggda med ett antal leder och länkarmar för att möjliggöra robotens rörelser. De tre robotar som används i projektet, är alla 6-axliga vilket betyder att de kan utnyttja alla sex frihetsgrader och en av robotarna visas i figur 4.9a. För att uppnå sex frihetsgrader ska roboten kunna röra sig linjärt i XYZ-planet men även kunna rotera kring respektive axel vilket illustreras i figur 4.9b [5]. Detta ger roboten ett stort arbetsområde.



(a) *Industrirobot*

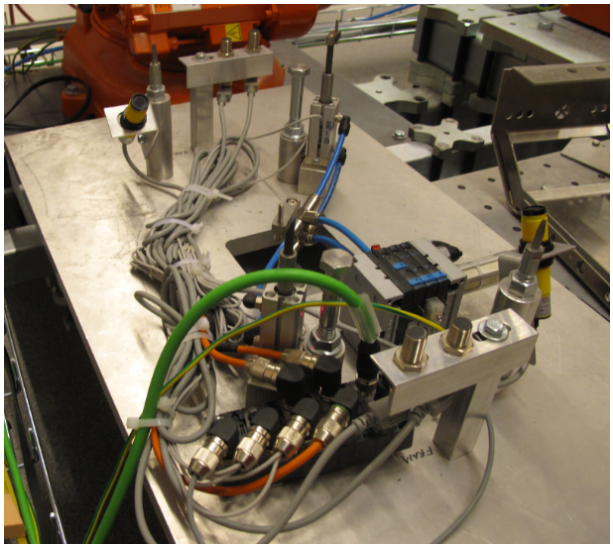


(b) *Illustrerar de sex frihetsgraderna*

Figur 4.9: *Figur a visar Robot R2 i produktionssystemet och figur b illustrerar de sex frihetsgraderna*

4.3.3 Fixtur

Modellbilen monteras på en fixtur som är centrerad i cellen, vilket gör att alla använda robotar som används i projektet kan nå fixturen. Fixturen kan ses i figur 4.10. Fixturen består utav ett antal komponenter som styrpinnar, låscylindrar samt sensorer. Styrpinnarna hjälper till att föra ner och fixera komponenterna på rätt ställe i fixturen för att sedan låsa den sista frihetsgraden med hjälp av två låscylindrar. Ett antal sensorer är installerade i fixturen för att ge signaler till en PLC om vilka komponenter som är monterade på fixturen.



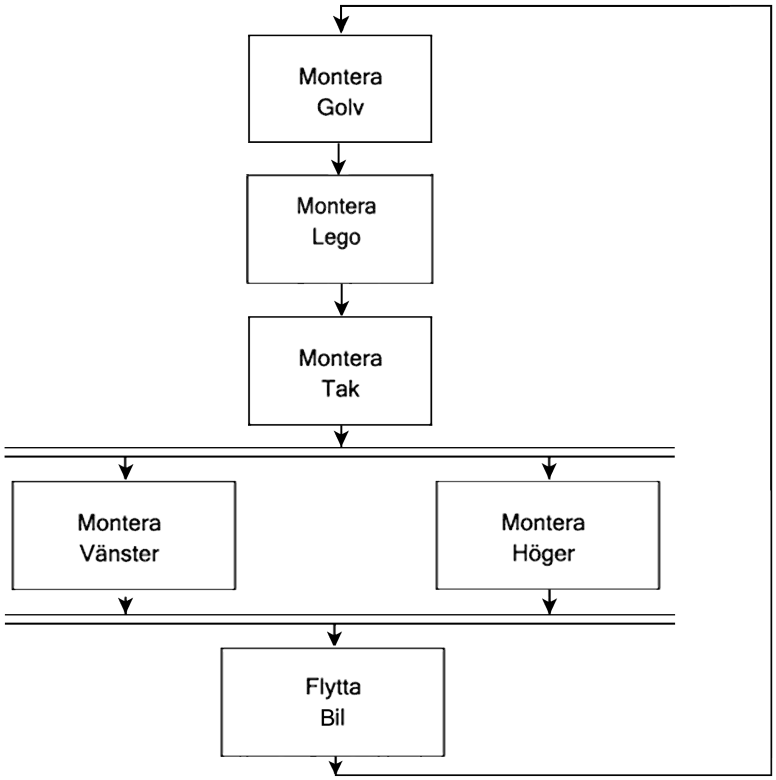
Figur 4.10: *Fixturen där modellbilen monteras*

4.3.3.1 Avlastningsbord

För att underlätta monteringen innehåller cellen två avlastningsbord som är placerade i anknytning till fixturen. Dessa är till för att underlätta monteringen av modellbilen genom att skapa buffertar och för avlämning av färdigmonterade produkter.

4.3.4 Montering av modellbil

Monteringsprocessen av modellbilen kan se olika ut beroende på hur dess program är utformat men den kan generellt beskrivas enligt en SOP som visas i figur 4.11.



Figur 4.11: *SOP över montering av modellbil*

De steg som utförs för att montera en av delarna är:

- **Steg 1 – Materialförsörjning**
  - I det första steget är det tillåtet med fri rörelse för operatören då denna är utom räckhåll för robotarna. I detta steg sker manuellt arbete så som beredning av material. Operatören använder tillhörande transportfixturer där modellbilens komponenter fixeras.
- **Steg 2 – Transport**
  - Efter att operatören startat processen skickas sedan paletten vidare in i flexlinkbanan. Paletten åker längs transportbandet till en av flexlinkbanans hissar som lyfter upp paletten med modellbilens komponent.
- **Steg 3 – Positionering i fixtur & montering**
  - När paletten hissats upp och fixerats i flexlinkbanans hiss kan robot R2 med hjälp av dess verktyg plocka upp komponenten och föra in komponenten till fixturen alternativt till ett avlastningsbord där monteringen sker automatiskt.
  - I detta steg är det inte tillåtet att vistas i cellen under körning då detta är inom robotarnas arbetsområde. För att kunna garantera operatörens säkerhet har säkerhetsbommar och säkerhetskameror installerats som automatiskt stoppar cellen då en operatör går inom angivet område.

4.4 Robotverktyg

En förutsättning för robotarnas arbete är att det finns robotverktyg för olika operationer. Dessa robotverktyg monteras i robotens munstycke med hjälp av mekaniska och/eller pneumatiska låsningar. De robotverktyg som finns tillgängliga för detta projekt är två olika typer av verktyg:

- **Sugverktyg**
  - Sugverktyget består utav två parallella munstycken som med hjälp av pneumatik kan aktivera ett vakuum. Detta möjliggör hantering av objekt med platta ytor.
- **Gripverktyg**
  - Gripverktyget liknar en gripklo med två armar som med hjälp av pneumatik kan öppnas och stängas. Detta möjliggör plockning av ett antal olika geometrier.

5

Standard

Det finns huvudsakligen två sorters utformningar av programmeringsstandarder. Det första består av ett bibliotek av komponenter och instruktioner för hur dessa ska användas. Det andra består istället av regler och rekommendationer för hur koden bör skrivas. Volvos standard är skriven på det senare sättet och när Chalmers operationskoncept implementeras i PLC-programmering kan det liknas vid en standard skriven på det första sättet. De beskrivs mer ingående i avsnitt 5.4 & 5.3. Därefter beskrivs den nya standarden som har skapats genom kombinationen av dessa två.

5.1 Tags

Tags används i såväl standarden som i den automatiska kodgenereringen och dess uppgift är att definiera in- och utgångar som PLC:n är kopplad till. Tags består av ett namn, som bör vara relevant till hårdvaran som är inkopplad, en datatyp och en adress, se figur 5.1. När hårdvaran ska användas behövs enbart taggens namn anropas istället för adressen [22].

PLC tags

	Name	Tag table	Data type	Address
424	AGVDocka_Magnetventil2	ASi Enheter	Bool	%Q160.1
425	AGVDocka_Magnetventil3	ASi Enheter	Bool	%Q160.2
426	AGVDocka_Magnetventil4	ASi Enheter	Bool	%Q160.3
427	Fixturkonsoll_Givare1	ASi Enheter	Bool	%I160.0
428	Fixturkonsoll_Givare2	ASi Enheter	Bool	%I160.1
429	AS-i_Bred	ASi Enheter	Bool	%I160.4

Figur 5.1: *Exempel på olika tags*

I figur 5.1 och under kolumnen *Name* syns taggens namn. Tag-lista visar i vilken lista tag:en finns. Under *Data type* definieras vilken typ av data som en tag hanterar. Längst till höger är *Address* som i sin tur visar vilken adress som den aktuella taggen har. Det finns tre olika sorters adresser, ingångar som betecknas med ”I”, utgångar betecknade med ”Q” och interna tags betecknade med ”M”.

5.2 Operationsblock

I Siemens TIA Portal V13, en utvecklingsmiljö för PLC-logik, skiljs det på Functions och Function Blocks så till vida att Function Blocks inte har något dedikerat minne. Function Blocks, även kallade operationsblock, funktionsblock eller endast block, används som kodblock eller subrutiner i kod som för övrigt skrivs i Functions, funktioner. Ett mycket basalt sätt att beskriva ett operationsblock är just som en subrutin med in- och utvariabler där något som utförs inuti blocket.

5.3 Chalmers operationskoncept implementerat i PLC

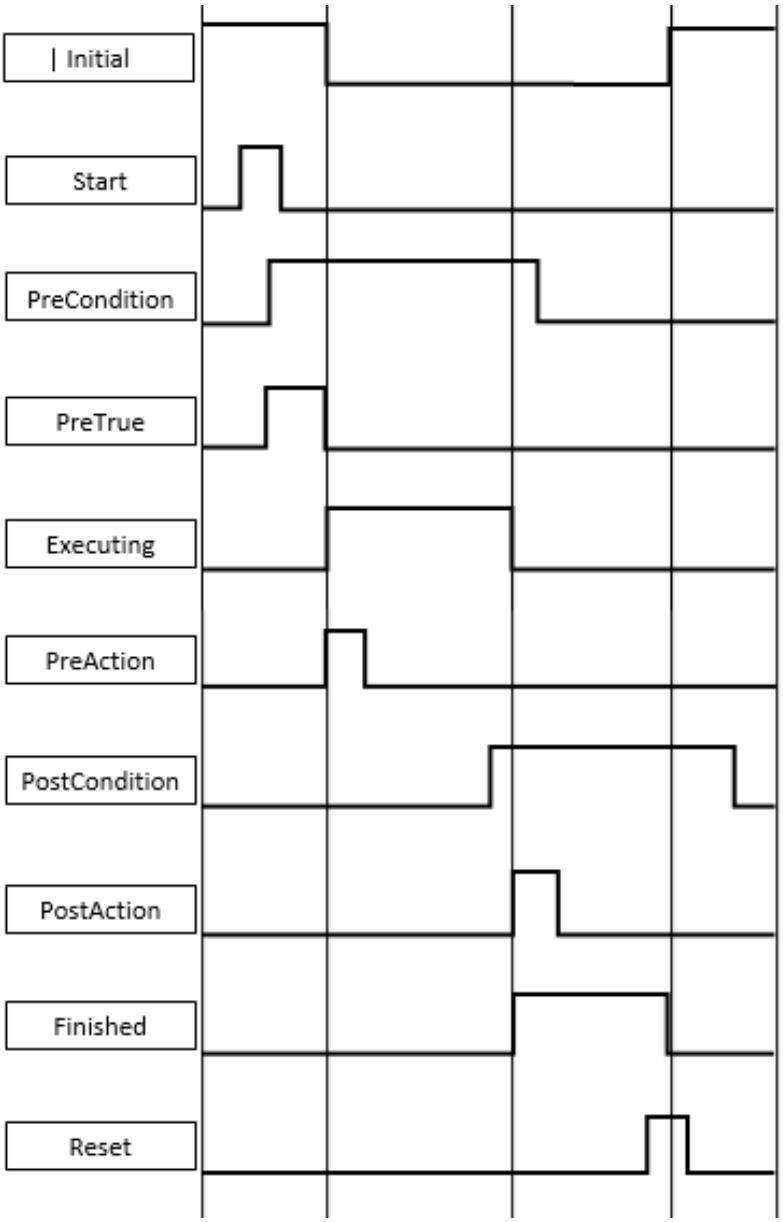
Skulle Chalmers operationskoncept implementeras vid programmering av en PLC kan ett standardutförande av operationsblock tas fram. Det vill säga att när varje uppgift beskrivs med villkor för när de får utföras och när de är färdiga kan de alla beskrivas enligt tabell 5.1 vid programmering. Där syns även vilka in- och utgångar som kan behövas för det standardiserade operationsblocket.

Tabell 5.1: *In- och utgångar för standardiserat operationsblock*

Ingångar	**Utgångar**
Start	*Initial*
PreCondition	*PreTrue*
PostCondition	*PreAction*
Reset	*Executing*
	PostAction
	Finished

En utformning av det standardiserade operationsblocket har sedan tidigare implementerats i programmeringen av PLC:n för produktionscellen på Chalmers. I figur 5.2 visas ett tidsschema över hur in- och utsignalerna i detta block förändras med tiden och hur de påverkas av varandra. In- och utgångarna beskrivs nu enligt den kronologiska ordning som de används när operationsblocket exekveras. Innan blocket börjar exekvera är utgången *Initial* sann. Utanför blocket skrivs vilka initiala villkor som måste vara uppfyllda för att operationsblocket ska få exekveras. När dessa villkor blivit uppfyllda blir insignalen *PreCondition* sann tillsammans med utsignalen *PreTrue* som indikerar att operationsblocket är redo att gå från *Initial* till *Executing*. Vid normal exekvering av operationsblocket är *PreTrue* endast sann en mycket kort stund innan läget *Executing* blir sann då *PreTrue* blir falsk. *Executing* indikerar att operationsblocket exekveras och blir sann samtidigt som utsignalen PreAction. Utsignalen *PreAction* används utanför blocket som villkor för de uppgifter som ska utföras samtidigt som operationsblocket exekverar. Så länge som *Executing* är sann väntar programmet på att insignalen *PostCondition* ska bli sann. På samma sätt som för *PreCondition* skrivs villkor utanför blocket men denna gång för när operationsblocket är slutfört. När *PostCondition* blir sann blir även utsignalen *PostAction* sann vilket gör det möjligt att använda den som villkor för uppgifter som ska genomföras efter att exekveringen av blocket är färdig. Samtidigt som *PostAction* blir sann blir *Executing* falsk och exekveringen av blocket är färdigt. Utgångarna är globala variabler

och kan läsas på andra ställen i koden utöver endast i samband med exekveringen av det specifika operationsblocket. Ingången *Reset* kan aktiveras för att återgå till initialläget där *Initial* är sann.



Figur 5.2: *Tidsshema över in- och utgångarna*

Fördelen med denna utformning av operationsblock är att den är generell nog för att kunna användas i alla sammanhang vilket tillåter återanvändning och eliminerar behovet av operationsblock med annan utformning. Koden blir mer lättförståelig då den alltid ser ut och fungerar på samma sätt oavsett vem som har skrivit koden och vilken funktion operationsblocket uppfyller. Operationskonceptet lägger fokus på aktiviteterna i programmet och ser till att logiken inte sprids ut på för många nivåer i koden utan är lättöverskådlig för de aktuella operationerna. En annan fördel är att konceptet ger mer kontroll över vilket tillstånd i operationen befinner sig i.

5.3.1 Säkerheten

I produktionscellen i PSL ligger säkerhetsstyrningen i ett program som är frikopplat från den övriga koden. Detta innebär att säkerheten fungerar så som tänkt även om resten av programmen inte gör det. Det minskar även risken för att säkerheten glöms i koden så att den inte fungerar. Att ha en fristående säkerhet är optimalt när det är många olika programmerare som arbetar på cellen och särskilt när programmerarna är studenter som inte alltid är till fullo insatta i befintligt program eller sättet att programmera PLC överlag.

5.4 Volvos PLC-programmeringsstandard

Volvo lägger idag väldigt stor vikt vid att deras program ska utgöras av en lättöverskådlig kod för att underlätta operatörernas arbete vid felsökning av driftstopp [9]. De är måna om att koden ska vara av en flexibel struktur för att kunna återanvändas för att effektivisera mjukvaruutvecklingen samt generera den överskådliga kod som uppnås genom den generella strukturen [9]. Volvo är även måna om att minimera programbiblioteket för deras PLC-enhet. Utöver de delar av koden som tillhandahåller de autonoma produktionsenheterna ska den även klara av att hantera HMI, *Human Machine Interface*, säkerhetssystem, larmhantering samt övriga delar som berörs för att upprätthålla en produktionsenhet av hög standard [9].

För att dessa mål ska eftersträvas vid programmering av alla Volvos PLC-enheter, när de som skriver koden utgörs av konsulter och stora utvecklingsteam, krävs skrivna regler. Dessa regler samlas i en skriven standard där det tydligt framgår hur programmeringen ska gå till. Generella regler som gäller för programstrukturen inkluderar exempelvis att koden ska vara simpelt skriven så att felsökning ska förenklas, att koden i varje Network inte får bli för lång så att det blir svårläst och att det genomgående språket vid programmering ska vara amerikansk engelska [8, 23].

5.4.1 Variabler

Variabler är globala i PLC oavsett om de är skapade för att vara globala, interna eller temporära. På grund av detta är det viktigt att variablerna hanteras på ett sätt som förhindrar misstag och svårhanterliga databaser. Variabler och adresser för fysiska I/O:s, in- och utgångar i en fysisk enhet, så kallade minnesceller, samlas i den så kallade tag-listan. Minnesceller får endast vara definierade av Volvo [8]. Tag-listan kan även användas till andra variabler men för att interna variabler inte ska förväxlas med minnesceller kräver Volvo att interna variabler endast ska läggas i databaser. Interna variabler får inte läsas eller skrivas utanför instansdatablock. Enligt standarden ska temporära variabler undvikas [8].

För att erhålla en god överblick är det viktigt att alla förändringar som införs dokumenteras kontinuerligt så att nästkommande programmerare förstår befintlig kod och vet vad som behöver ändras eller läggas till. En av de regler som finns skrivna i Volvos standard för PLC-programmering är att alla förändringar måste dokumenteras. Detta görs separat för varje Project i ett biblioteksobjekt kallat ”Changes” som finns i varje PLC-projekt.

5.4.2 Koden

Koden som används vid PLC-programmering på Volvo är FBD men med SFC i Main-programmet och inuti funktionsblocken [24, 8]. Som kommunikation in till och ut ur funktionsblock använder de sig ofta av en variabel av datatypen word för att endast ha en variabel in och en ut. Inuti blocken packas variabeln upp och delas upp i separata bits där varje position i word-variabeln är dedikerad till en boolsk funktion, det vill säga en funktion som har lägena sant eller falskt. På detta sätt kan 18 variabler packas ihop i en enda variabel. Namnen på variabeln väljs noggrant och efter Volvos standard för att försvåra förväxling av variablerna [23].

För att strukturera upp projektträdet över alla funktioner, funktionsblock och databaser sorteras de i olika mappar. Grupperingarna skapas efter den funktion som blocken utför och efter vilket sorts block det är. Exempelvis ligger alla databaser samlade i en mapp kallad ”DB:s” och alla funktioner som rör robotarna ligger i en mapp kallad ”Robots”[24]. De funktionsblock som finns är kallade ”Sequences” och ligger även dessa i en separat mapp [24].

5.4.3 Säkerhet och felhantering

Säkerhet och felhantering behandlas i nästan alla funktioner i Volvos PLC-kod. Detta innebär att en stor del av all kod behandlar dessa delar. Alternativet till att ha med kod som rör säkerheten överallt är att ha ett övergripande säkerhetssystem som är fristående från övrig kod.

5.5 Resultat

En kombination av Volvos PLC-standard och Chalmers implementerade operationskoncept har gjorts och i detta avsnitt redogörs för resultatet. Chalmers operationskoncept har använts som bas och sedan har anpassningar gjorts för Volvos standard. Programvarorna som använts vid tidigare kandidatarbeten i cellen är desamma som de som används på Volvo och som anges i deras standard. Dessa program är bland andra Siemens Tia Portal V13 och WinCC.

5.5.1 Ny standard

Main-program skrivs i ett operationsblock av hög komplexitet istället för i SFC enligt Volvos standard. Den högsta operationen anropar andra operationsblock men är den enda funktion som ska behöva startas manuellt av operatören när tillverkningsprocessen startar, programmet fortsätter sedan exekvera tills det avslutas.

De operationsblock som har implementerats har delats in i två underkategorier: Abilities och Operations. De är båda utformade enligt operationskonceptet men skiljer lite från varandra i hur koden är utformad inuti operationsblocken. Syftet med Abilities är att fungera som den minsta byggstenen i PLC-programmet vilket innebär att de ska utföra små aktiviteter och kunna repeteras önskat antal gånger utan återställning. De aktiveras

genom ingången *Start* och kan återställas med *Reset* även om detta inte krävs vid normal exekvering. Det är meningen att Abilities inte ska ha något minne eller medvetande om övrig kod och de ska inte vara programspecifika. De ska alltså enbart utföra en aktivitet och vara helt fristående från övriga aktiviteter. Det finns ingen återkoppling från Abilityn när den är färdigexekverad för att undvika onödig kommunikation och alltför många så kallade handskakningar i programmet.

Operations är istället mer sekvensstyrda. Så som de har implementerats exekveras de automatiskt då deras *PreConditions* har uppfyllts. Till skillnad från en Ability går Operations inte direkt tillbaka till *Initial*. *Reset* måste aktiveras för att operationsblocket ska återställas. Operations kan beskrivas som nivån över Abilities i grad av komplexitet. Detta eftersom Operations utför något mer omfattande uppgifter och har ett mer specifikt ändamål. Operations kan bestå av antingen Abilities, annan kod eller någon kombinationsgrad däremellan. Operations kan tillåtas ha fler villkor för start, exekvering och avslutande av operationen än vad Abilities kan. Fortfarande skall de inte i onödan låsas till specifika program men å andra sidan skall inte hänsyn behöva tas till sådana saker som alltid gäller utanför blocket. Exempelvis ska en Operation inte kunna utföras i ett läge som innebär att skador på produktionscellen kan uppstå. Operations kan användas på olika nivåer i koden. De Operations som är av lägsta nivå i komplexitet skiljer inte särskilt mycket från Abilities. De ska vara skrivna på ett så generellt sätt som möjligt för att förenkla återanvändning och aldrig ha onödiga *PreConditions* och *PostConditions*. För en mer detaljerad beskrivning av Abilities och Operations se stycket för styrning 6.1.2.

Utgången *Finished* kan användas som kommunikation mellan operationsblocken för att exempelvis se till att en operation inte påbörjas innan en annan har avslutats. En högre nivå på Operations innebär att de blir mer låsta till sekvensstyrning av specifika program. Uppgifterna de gör blir mer specialiserade och komplicerade och består ofta av sammansättningar av Operations på lägre nivåer. *PreConditions* och *PostConditions* för Operations på högre nivåer handlar mer om i vilken ordning och när de ska utföras i programmet snarare än om den detaljerade säkerheten eftersom säkerheten istället till stora delar styrs på de lägre nivåerna. Med detaljerad säkerhet menas här den säkerhet som är direkt sammankopplad till en specifik operation eller uppgift i motsats till den övergripande säkerheten som påverkar mycket större delar av programmet och cellen, exempelvis nödstopp.

Kombinationen med Volvos standard innebär bland annat att en del regler från Volvos standard implementeras. Exempelvis används amerikansk engelska genomgående i programmet så när som på några få namn på I/O:s som finns kvar från programmet i den gamla PLC:n. Scrollning i sidled ska undvikas och har vid implementering inte krävts. Tag-listan har endast använts till I/O:s och temporära variabler har undvikits men ej uteslutits helt. Volvos prioritering av kod där läsning, felsökning och införande av ändringar är simpel stämmer överlag väl överens med Chalmers operationskoncept som gör koden enhetlig, kompakt och flexibel.

Ett avseende där standarderna inte stämmer väl överens är vid läsning och skrivning av interna variabler vilka, liksom alla andra sorters variabler, är globala i PLC. Enligt Volvos standard är det inte tillåtet att läsa eller skriva interna variabler på andra ställen än i den aktuella funktionen. I den kombinerade standarden görs detta konstant på grund av utformningen av operationsblocken. Varje läsning eller skrivning av in- eller utgångar som *Start* eller *Finished* innebär en läsning av en intern variabel men detta är ett

medvetet undantag. Detta eftersom det är permanenta in- och utgångar som är likadana för alla operationsblock läggs inga extra variabler till som komplicerar koden. I den nya standarden är det tillåtet att använda sig av alla variabler som tillhör grundutförandet av operationsblocken Ability och Operation.

6

Styrning

Efter att den nya standarden utvecklats behövde denna testas och verifieras genom implementering i PSL. Därför har det utvecklats en styrning för produktionscellen där både PLC-styrning och robotstyrning ingår, för att få en helhetsbild av standarden och hur den kan implementeras i ett produktionssystem. Det finns tre robotar som används i produktionssystemet och det har utvecklats ett program för styrning av dessa genom ABB:s programvara *RobotStudio*. PLC-styrningen består utav ett program som bygger på den nya standarden som beskrivits och det utvecklades i TIA-portalen.

6.1 PLC-styrning

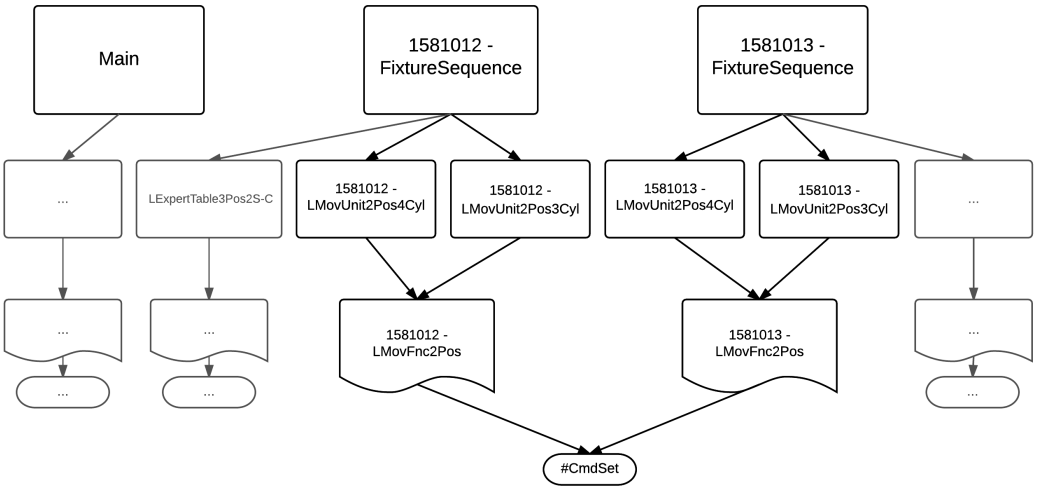
För att verifiera den nya standarden har den implementerats i PLC:n i PSL för styrning för montering av en modellbil. Vid implementeringen av den nya standarden i PSL har även hänsyn tagits till utformningen av styrningskoden för Volvos program och den kod där Chalmers operationskoncept sedan tidigare har implementerats.

6.1.1 PLC-styrning på Volvo

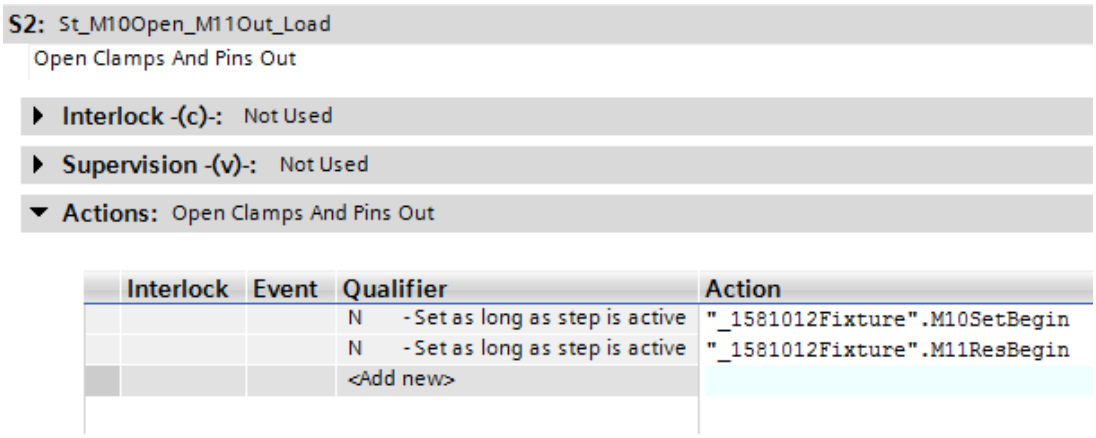
Volvos Main-program, i den exempelkod som tillhandahållits av Volvo, är skriven i SFC men har stora likheter med operationer i den nya standarden. Även om det finns ett program som heter Main exekveras den i själva verket parallellt med två andra program [24]. Dessa tre program eller operationer inbegriper alltså Main och två olika fixturstyrningar, alla skrivna i SFC där tydliga villkor och exekveringar finns som kan jämföras med *PreConditions* och *Actions* i den nya implementerade standarden. Även undernivåer av operationer som kan liknas vid den nya standardens Operations finns. Dock finns likheterna i dessa undre operationer till Operations endast i hur hierarkin av operationer är uppbyggd ty dessa undre nivåer i hierarkin är helt sekvensstyrda och saknar *PreConditions* och *PostConditions* som istället sköts av huvudprogrammet.

I figur 6.1 visas en schematisk bild över delar av hierarkin i den kod som erhållits av Volvo. Överst ses de tre nämnda operationerna men en detaljerad beskrivning ges endast för ett horisontellt led i hierarkin för en av de två operationerna kallade Fixture Sequence. I Appendix B.1 ses SFC-programmet för operationen. Det SFC-Step, steg, som kallas *St_M10Open_M11Out_Load* öppnar klämmorna i fixturen så att en komponent kan matas in i cellen för bearbetning eller montering av något slag. Klämmorna öppnas med koden som skrivits i det generella Networket *Actions* som finns i varje Step och som kan ses i figur

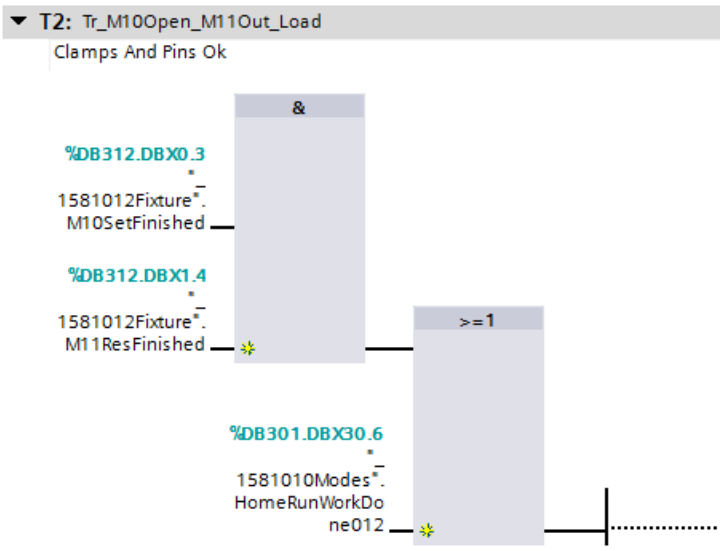
6.2. När signalen *M10SetBegin* blir sann börjar exekveringen av operationen och fortsätter sedan tills signalen blir falsk igen. Villkor för när exekveringen av ett Step kan avslutas anges i Transition. Transition T2 *Tr_M10Open_M11Out_Load* kan ses i figur 6.3 och ser till att *M10SetFinished* är sann. *M10SetFinished* är en utsignal från operationsblocket som insignalen *M10SetBegin* startar.



Figur 6.1: *Hierarkin i Volvos kod*



Figur 6.2: *Actions inuti ett Step i SFC-kod*



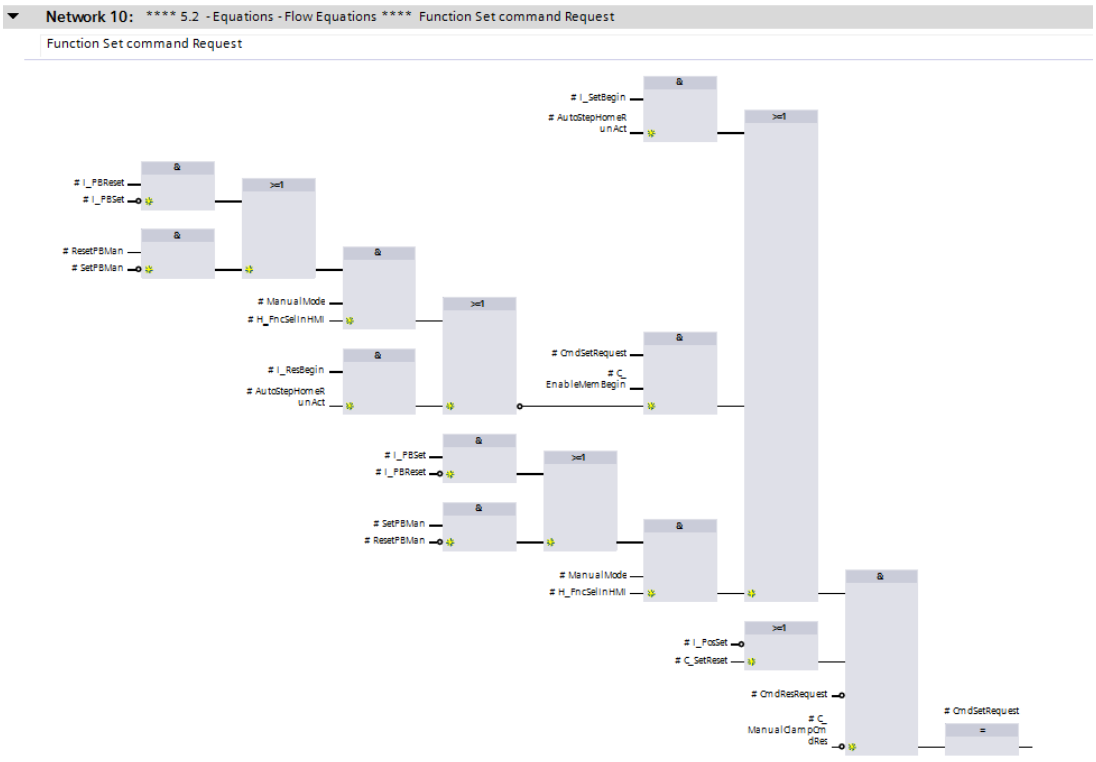
Figur 6.3: *Transition Tr_M10Open_M11Out_Load i SFC:n FixtureSequence*

Vid jämförelser med den nya standarden ses mycket tydliga likheter mellan Volvos *Actions* och Chalmers *PreActions* samt *PostActions* där skillnaden är att det på Volvo inte skiljs på vad som exekveras i början av operationen och vad som exekveras i slutet annat än med programkoden inuti aktuellt Step. Den kod som finns i en Transition är direkt jämförbar med de villkor som skrivs som villkor för insignalen *PostCondition* och *M10SetFinished* läses som utsignal för när blocket har exekverat klart på samma sätt som *Finished*. Den allra största likheten mellan de två styrningssätten är dock start-kommandot. När *M10SetBegin* skrivs som sann kan det jämföras med att insignalen *Start* till en Ability skrivs som sann, med den enda skillnaden att signalen är aktiv under hela exekveringen av operationen till skillnad från vid en Ability där *Start* sätts som falsk så fort som operationen påbörjat exekvering.

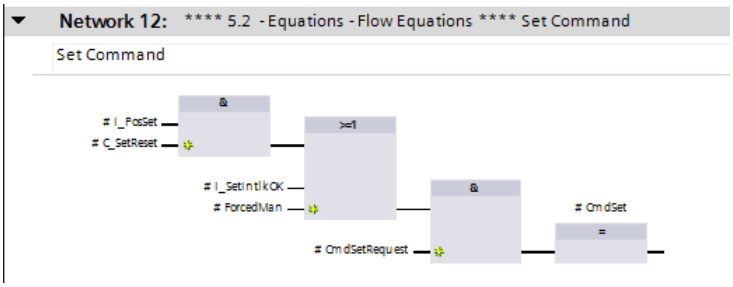
Operationsblocket *LMovUnit2Pos4Cyl* är det block som har ovan nämnda in- respektive utgång *M10SetBegin* och *M10SetFinished*. Detta operationsblock återfinns på nivån under de tre översta operationerna i hierarkin. Tack vare sättet som *LMovUnit2Pos4Cyl* styrs och startas på kan stora likheter med en Ability från den nya standarden ses. Dock utför blocket komplexa uppgifter, har en *Finished*-variabel och innehåller andra operationsblock vilket gör att *LMovUnit2Pos4Cyl* snarare bör liknas vid en Operation. När ingången *M10SetBegin* är sann är ingången *I_SetBegin*, start-kommandot till *LMovFnc2Pos* också sann.

Operationsblocket *LMovFnc2Pos* kan alltså påstås ligga på den tredje nivån i hierarkin över kodstrukturen. Även i detta operationsblock finns kod som behandlar larm trots att det i det övergripande operationsblocket *LMovUnit2Pos4Cyl* finns många Network som behandlar säkerheten. Detta beror på att Network:en kallade Alarm faller inom kategorin Interlock och istället behandlar villkor för exekvering. Koden i dessa Network kan jämföras med *PreCondition* och *PostCondition*. Utöver villkoren i Interlock finns många villkor invävda i logiken för exekveringen av varje block, med booelsk logik, vilket kan ses i figur 6.4 där insignalen *I_SetBegin* tillsammans med övrig logik sätter variabeln *CmdSetRequest*. *CmdSetRequest* sätter i sin tur utgången *CmdSet* i Network:et som kan ses i figur 6.5 samt en utgång kallad *FuncRunning* som är likvärdig med *Executing*. Utgången *CmdSet* är dels

en utgång från operationsblocket *LMovFnc2Pos* men den är även en fysisk I/O-utgång för styrning i cellen. Operationsblocket *LMovFnc2Pos* kan alltså sägas vara en sorts Ability och ligger i det här fallet längst ned i hierarkin för styrningen. Längst ned i den schematiska bilden över hierkin som visas i figur 6.1 illustreras även utgången *CmdSet* som är en ren output till cellen.



Figur 6.4: *Logik för variabeln CmdSetRequest*

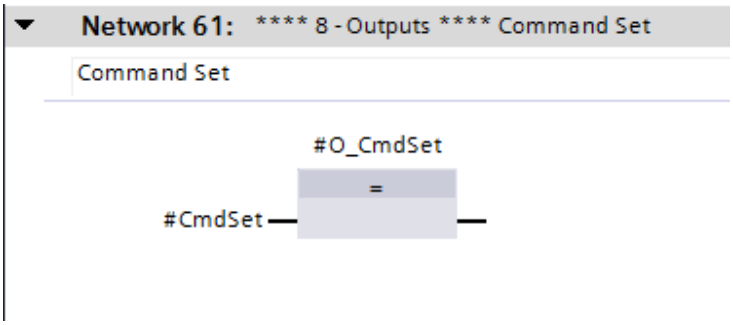


Figur 6.5: *Logik för variabeln CmdSet*

Det finns många versioner av operationsblocket *LMovUnit2Pos4Cyl*, dels för olika program men desto fler versioner finns för att skilja på olika antal klämmor som finns i cellen och som skall styras. I programmet som undersökts finns det 16 versioner av detta operationsblock och till stora delar ser koden identisk ut i dem alla men i vissa avseenden finns det smärre skillnader. Under *LMovUnit2Pos4Cyl* i hierarkin ligger operationsblocket *LMovFnc2Pos* vilket är samma oavsett hur många klämmor som skall styras och därför finns det enbart flera för att skilja på vilket program de hör till. Av detta operationsblock finns det istället fem stycken men detsamma med repetitiv kod gäller även här. Trots alla

de olika versionerna av dessa två operationsblock styr de alla en och samma fysiska utgång som sätts med variabeln *CmdSet*.

När utgångar ska läsas från ett operationsblock som ligger inuti ett annat operationsblock utanför dem båda undviks det att läsa denna variabel direkt trots att den alltid är global, enhetligt med Volvos standard. Lösningen på detta kan ses i figur 6.6 där utgången för det yttersta blocket sätts till samma värde som utgången för det inre blocket. Varje sådan översättning av utgångar till övergripande utgångar kräver sitt eget Network i kategorin Outputs vilket bidrar med i genomsnitt nio extra Networks i varje funktionsblock. Även dessa Networks är mycket lika varandra i koden och kan sägas vara repetitiva.



Figur 6.6: *Hantering av interna utgångar i Volvos kod*

Nästintill alla Networks i varje funktion samt funktionsblock är uppdelade i olika kategorier definierade av Volvo själva. Det finns sex kategorier och de finns sammanställda i tabell 6.1. Inputs är den kategori som hanterar ingångarna och däribland den variabel av typen word som används för enklare kommunikation av många booelska variabler mellan operationsblocken. I kategorin Inputs fås variabeln av typen word som ingång och sedan packas denna upp i operationsblock i Network:et och varje bit sparas som respektive variabel av typen boolean vilka fås som utgångar från blocket. Networks i kategorin Machine Status gör istället det motsatta, det vill säga att många boolska variabler fås som ingångar till operationsblocket och som utgång finns endast en variabel av typen word.

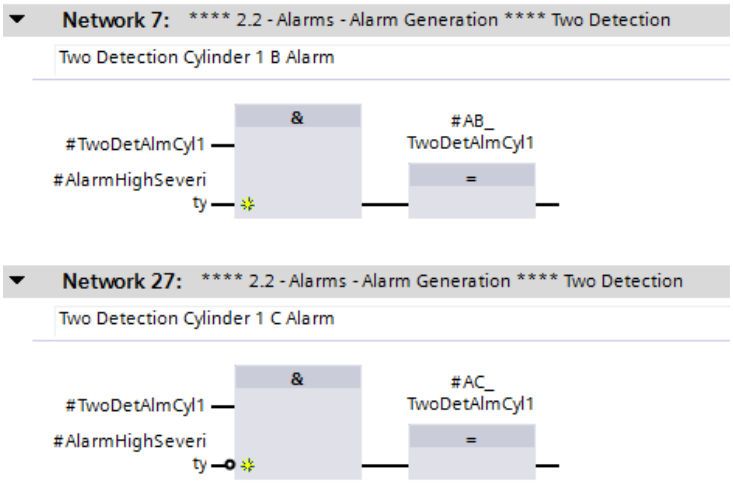
Tabell 6.1: *Kategorier för Networks i Volvos kod*

Kategorier
Inputs
Machine Status
Interlock
Submodule
Outputs
Alarms

Interlock, även kallad Equations, är Volvos version av *PreCondition* och *PostCondition* för i dessa Networks finns logiken som anger de villkor som måste vara uppfyllda för att funktionens olika uppgifter ska få utföras. Exempelvis finns det ett Network inom kategorin Interlock som ser till att de specifika klämmorna inte får stängas förrän vridbordet är i rätt läge. Interlock förekommer nästan enbart på de högre nivåerna i hierarkin på samma sätt som att Operations har fler villkor än vad Abilities har i den nya standarden. Kategorin Submodule är de Networks som faktiskt utför uppgifter och alltså exekverar delar av

sekvensstyrningen. Dessa Networks är få i förhållande till de andra kategorierna som sköter allt runt om kring exempelvis styrning av HMI och larm, så kallade korsrelaterade funktioner. Network-kategorin Outputs förekommer mest i de undre delarna av kodhierarkin eftersom dessa existerar enbart för att utgångar från operationsblock inuti operationsblock ska kunna läsas även längst upp i hierarkin.

Slutligen finns kategorin Alarms och dem utgör majoriteten av alla Networks i Volvos kod. Alarms delas upp i två underkategorier: Alarm Generation och Alarm Summary. Networks av typen Alarm Generation innehåller alltid samma logik, det som skiljer dem åt är variablerna. Koden i dessa block kan ses i figur 6.7. I figuren jämförs det Network som behandlar *Two Detection Cylinder 1 B Alarm* med det som behandlar *Two Detection Cylinder 1 C Alarm*. Som synes är det enda som skiljer dem båda åt variabeln *AlarmHighSeverity* som för den ena ska vara aktiverad och för den andra inte vilket avgör vilken sorts larm det ska aktivera. Sortens larm anges med bokstaven i namnet, här B och C. Det finns alltså flera versioner av larm för en och samma cylinder och sedan finns det dessutom samma uppsättning för varje cylinder som finns med, exempelvis finns det även ett Network som behandlar *Two Detection Cylinder 4 B Alarm*. Den andra kategorin Alarm Summary bidrar med två Networks för varje typ av larm som de har. I koden som har undersökts är det fyra stycken, A till D. Det första av de två Network:en kontrollerar ifall ett eller flera av larmen av den typen är sann och om det är något som är sant sätts en variabel *Alarm B* exempelvis. När *Alarm B* blir sann hanterar då det andra av dessa två Networks det och ser till att utvariabeln *O_Alarm B* också blir sann så att denna kan läsas utanför operationsblocket eller funktionen. Som det går att se i operationsblocket *LMovUnit2Pos4Cyl* i Appendix B.2, ett bra exempel på ett typiskt operationsblock, är det en mycket stor del av alla Networks som faller inom kategorin Alarm. Att dessa dessutom är mycket lika varandra inuti kodmässigt talar för att detta är delar av koden som skulle lämpa sig väl för automatisk kodgenerering.

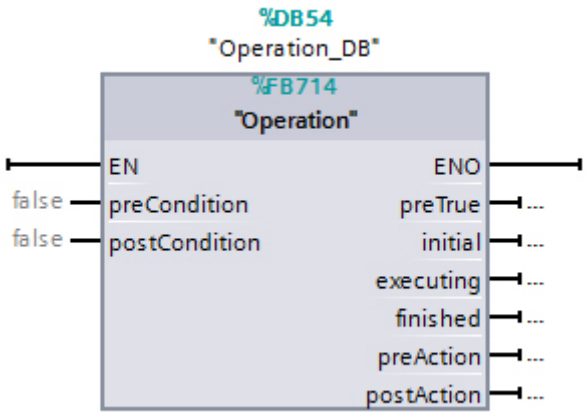


Figur 6.7: *Två exempel på Networks inom kategorin Alarm i Volvos kod*

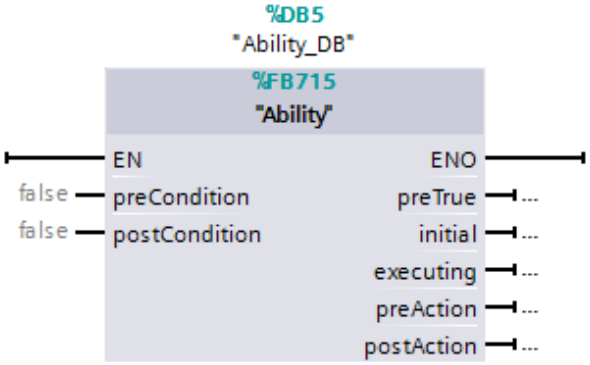
6.1.2 Resultat

Operationsblocken Operation och Ability har skapats och implementerats enligt den nya standarden. I figurerna 6.8 och 6.9 ses hur operationsblocken Ability respektive Operation

ser ut i programmet då de används i funktioner i koden. I figurerna syns in- och utgångarna som definierar de specifika operationsblocken. För koden inuti operationsblocken se Appendix G.

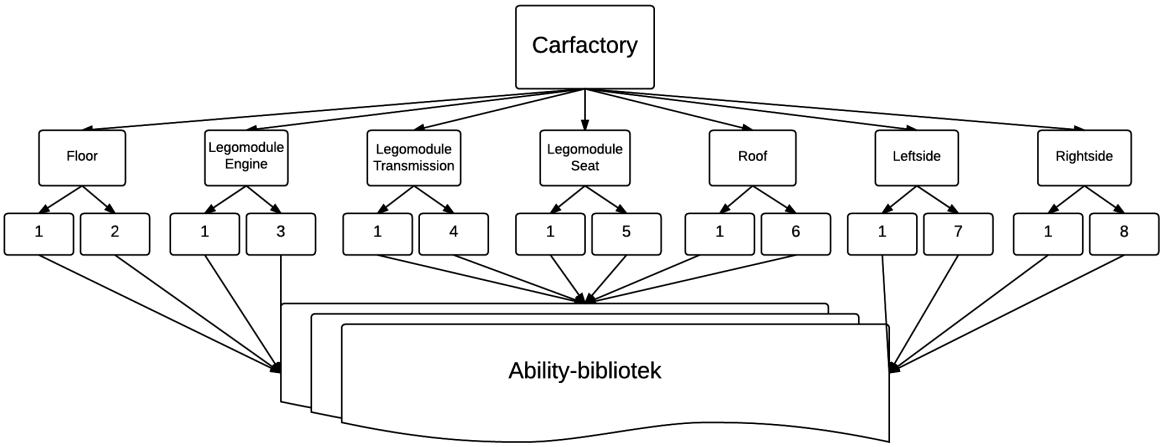


Figur 6.8: *En Operation*



Figur 6.9: *En Ability*

Hierarkin för programmet som styr monteringen av modellbilen i produktionscellen representeras i figur 6.10. Längst upp ses den Operation som är Main-blocket och som initierar hela programmet och som enligt standarden är det enda som kräver aktiv initiering av operatören. Denna Operation innehåller sekvensstyrningen av andra Operations. Den näst översta nivån i hierarkin består av Operations som även de är mycket låsta till det specifika programmet.



Figur 6.10: *Illustration av hur hierarkin av Operations och Abilities ser ut i den utvecklade styrningen*

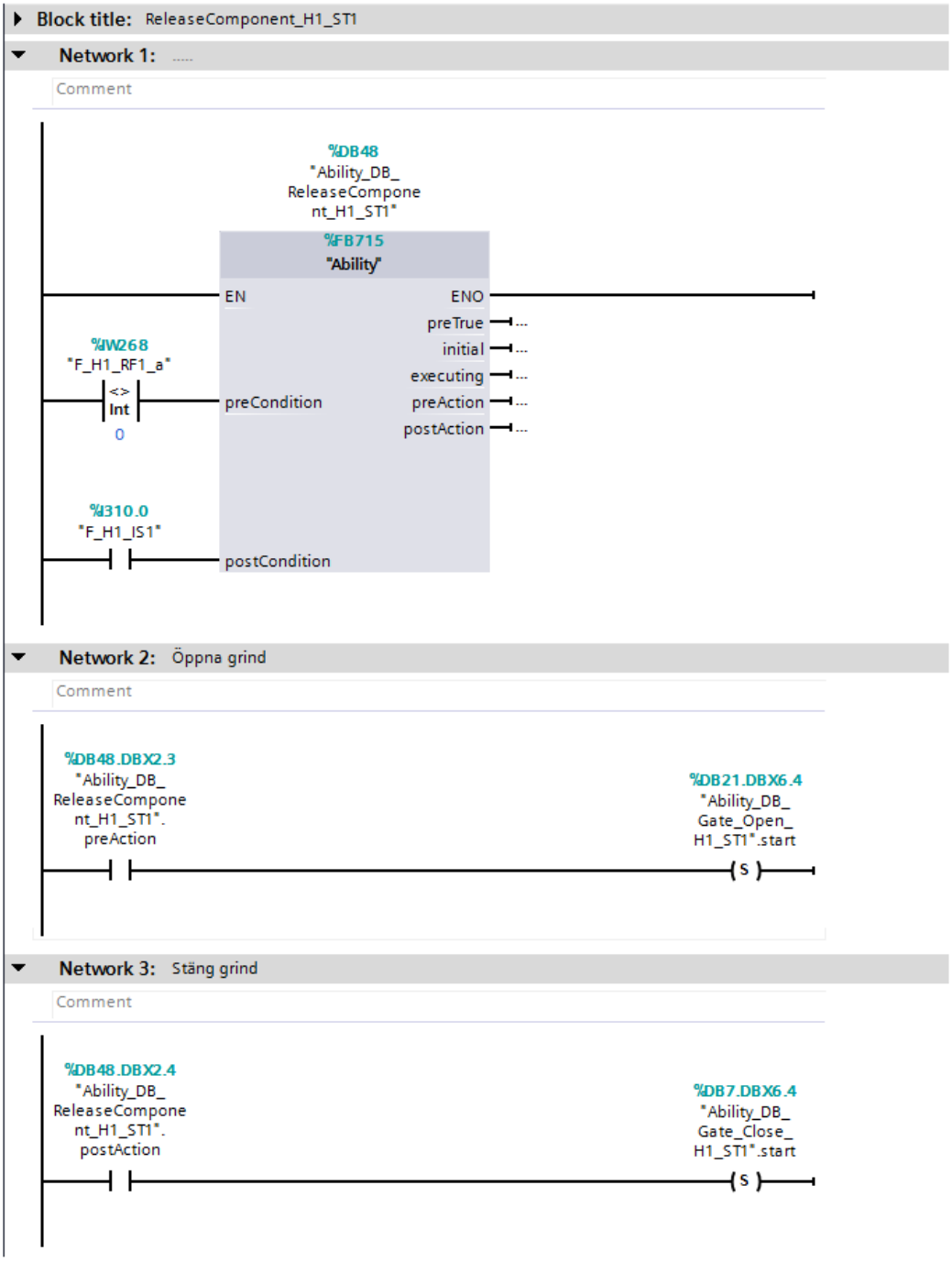
Ju längre ned i hierarkin desto mindre styrda är operationsblocken, särskilt så långt ned som vid Ability-nivåerna. Abilities är, som specificerat i standarden, inte låsta till särskilda program över huvud taget och på så sätt mycket återanvändbara. Värt att anmärka på är dock att den lägsta nivån av Abilities, de allra minsta modulerna, är skriven på en något lägre nivå än vad som kan anses optimalt. Detta beror på att styrningen via ett HMI förenklas om även de enklaste av operationer kan utföras med hjälp av operationsblock. Den lägsta nivån av Abilities utför endast en uppgift på en så grundläggande nivå att endast en I/O-utgång används. För att undvika de bekymmer som kommer av att ha alltför många Abilities, och som beskrivs i avsnittet om Abilities ovan, har en uppdelning gjorts av Abilities på den lägsta nivån och de som utför fler operationer än bara en. De är placerade i grupper kallade Micro Abilities och Macro Abilities vilket gör att biblioteket inte blir mindre överskådligt. Bortsett från antalet Abilities kan det ses som att programmering med så kallade Micro Abilities är helt i enlighet med Chalmers operationskoncept eftersom de sannerligen är återanvändbara och inte alls kod- eller funktionsspecifika.

Implementeringen av säkerhetsfunktioner för cellen har valts som en avgränsning för kandidatarbetet. Den säkerhet som har använts är samma som den befintliga säkerheten som har använts av den föregående PLC:n och har bara förts över så som den är.

6.1.2.1 Implementerade Abilities

I utformningen av en Ability finns det ingen variabel som indikerar när operationen är slutförd men vid implementeringen i cellen krävdes det ändå en sådan i särskilda fall. För att inte bryta mot operationskonceptet infördes inte Finished, vilket skulle ha skapat ett mellanting mellan en Operation och en Ability. Istället infördes vid de tillfällena unika variabler fristående från utformningen av Ability:n. Även om extra variabler bör undvikas görs i detta fall ett undantag eftersom det annars hade lett till en mycket mer komplex lösning vilket kan påstås strida mot Volvos standard som föredrar simpla lösningar framför komplexitet.

Ett exempel på en Macro Ability från den implementerade koden är *ReleaseComponent_H1_ST1* som kan ses i figur 6.11 där koden utanför funktionsblocket visas. *H1_ST1* är namnet på den grind som finns vid hiss nummer 1, se figur 4.7 över cellen. Den här Ability:n används för att släppa igenom endast en komponent genom grinden vilket innebär att den öppnar grinden och därefter stänger den igen när en komponent har passerat. Längst upp syns namnet på operationsblocket, vilket är *ReleaseComponent_H1_ST1* och direkt under det Network 1 som innehåller själva operationsblocket som här är av utformningen Ability. Bilden visar programkoden utanför blocket som använder sig av dess in och utgångar. Det villkor som krävs för att ingången *PreCondition* ska bli sann är i det här fallet att värdet som fås från RFID-läsaren *F_H1_RF1_a* ska vara skilt från 0, det vill säga att det befinner sig en komponent framför den. När *PreCondition* har blivit sann inuti blocket blir även utsignalen *PreAction* sann vilken i Network 2 har satts som villkor för att ingången *Start* till Micro Ability:n *Gate_Open_H1_ST1* ska få bli sann. Då öppnas grind *H1_ST1*. Villkoret för att *PostCondition* ska bli sann är att signalen från sensorn *F_H1_IS1* blir sann för när den blir sann innebär det att en komponent passerar sensorn placerad efter grinden som just öppnades. Direkt när *PostCondition* har blivit sann blir utsignalen *PostAction* sann och den har i Network 3 satts som villkor för att insignalen *Start* till Micro Ability:n *Gate_Close_H1_ST1* ska få bli sann. När den sistnämnda blir sann stängs alltså grind *H1_ST1*.



Figur 6.11: *Exempel på en Ability*

6.1.2.2 Implementerade Operations

De Operations som har skapats är alla en del i att producera modellbilen i cellen och därför är få av dem helt generella för till exempel flera sorters produkter eller andra sorters produktionsflöden. Optimalt hade varit att endast ha generella operationsblock eftersom de då är optimala moduler men det är inte realistiskt att skriva ett sådant program, en del låsningar och specificeringar måste göras. Särskilt när all sekvensstyrning också görs med operationsblock.

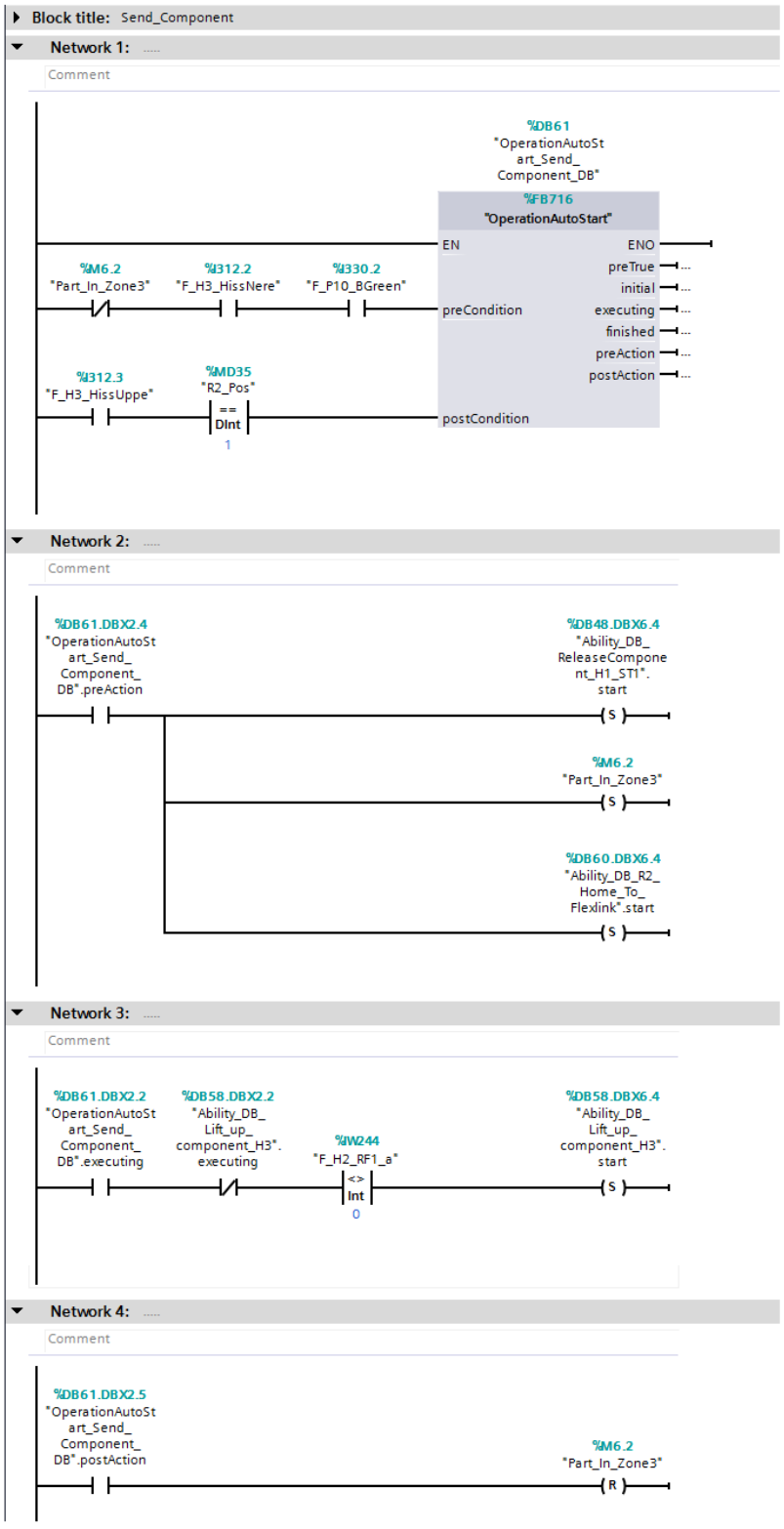
De Operations som har implementerats är skrivna så att de ska vara självstartande då villkoren för deras *PreConditions* är uppfyllda. Detta hjälper vid sekvensstyrningen av cellen så att programmet kan exekveras utan stopp men samtidigt vara något mindre låst till ett program än om en Operation alltid startar nästa Operation i enlighet med ett specifikt program.

Ett exempel på en Operation är *Send_Component*, se figur 6.12 för koden utanför operationsblocket. Denna Operation inväntar operatören signal på att en komponent är redo att med flexlinkbanan åka in till hissen där den hissas upp för att möjliggöra att roboten kan hämta den. Längst upp syns namnet på operationsblocket, vilket är *Send_Component* och direkt under det Network 1 som innehåller själva operationsblocket som här är av utformningen Operation. Bilden visar programkoden utanför blocket som använder sig av dess in och utgångar. Det första villkoret för att denna Operation ska få börja exekveras är att sensorn som känner att hissen är nere, *F_H3_HissNere*, är sann vilket innebär att hissen är i sitt nedre läge. Nästa villkor som måste vara uppfyllt är att variabeln *Part_In_Zone3* inte får vara sann. Denna variabel har skapats på grund av att de olika modellbilskomponenterna inte får ligga precis bredvid varandra på banan eftersom att de då slår i varandra och därför kontrolleras det att det endast är en komponent i taget i varje zon. När dessa båda villkor är uppfyllda inväntas montörens signal vilket är en tryckning på knappen *F_P10_BGreen*.

Montören trycker alltså på knappen då den aktuella komponenten är fixerad på fixturen. Så snart alla dessa villkor har uppfyllts så utförs operationerna med utgången *PreAction* som villkor vilket här är tre olika operationer, Network 2. Den första skickar signalen *Start* till den Ability som beskrivits i avsnittet Implementerade Abilities 6.1.2.1, *ReleaseComponent_H1_ST1* som släpper igenom komponenten genom den första grinden. Den andra sätter variabeln *Part_In_Zone3* eftersom det befinner sig en komponent i zonen så fort som grinden har öppnats. Den tredje skickar *Start* till robot R2, *R2_Home_To_Flexlink*, vilken kör roboten från hemmaläget till det läge som kallas *Flexlink* vilket är ett vänteläge innan roboten kan plocka komponenten. Eftersom roboten inte utför plockningen av komponenten i denna Operation innebär det att den kan återanvändas till alla olika sorters komponenter. I Network 3 används operationsblockets utgång *Executing* tillsammans med att en sann signal från sensorn *F_H2_RF1_a* inväntas innan *Start* kan skickas till Ability:n *Lift_Up_Component_H3*. Dessutom får inte utgången *Executing* för *Lift_Up_Component_H3* vara sann vilket ser till att *Start* inte fortsätter skickas efter det att Ability:n har påbörjat exekvering.

Villkoren för att *PostCondition* ska bli sann och därigenom avsluta exekveringen av blocket är dels att sensorn *F_H3_HissUppe* blir sann vilket innebär att hiss *H3* har lyfts upp till sitt översta läge och dels att robot R2 är framme och står still i positionen *Flexlink*. När *PostCondition* har blivit sann gör även *PostAction* det och *Part_In_Zone3* kan återställas.

Eftersom detta är en Operation innebär det även att utsignalen *Finished* blir sann och kan användas för att fortsätta programmet, exempelvis med en till Operation som ser till att roboten plockar komponenten.



Figur 6.12: *Exempel på en Operation*

6.2 Robotstyrning

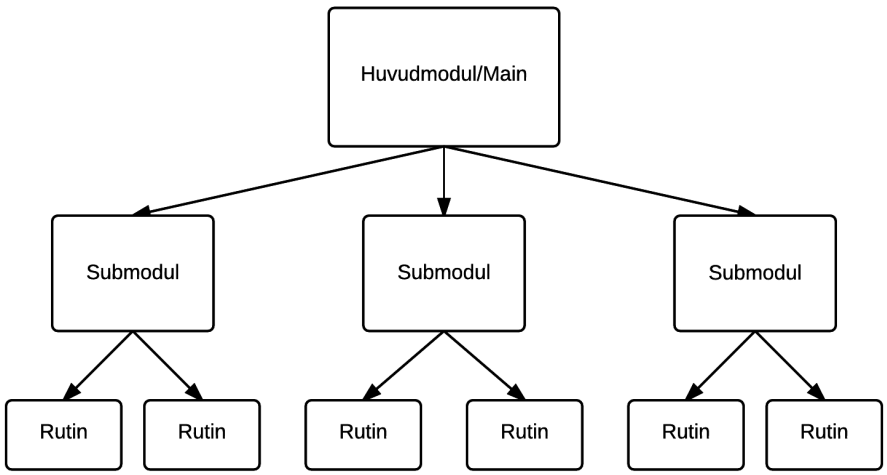
För att kunna exekvera i produktionscellen måste robotarna styras. Det har utvecklats ett program där ett antal robotoperationer är skapade och där målet är att montera modellbilen. Samtliga robotar som används i projektet är levererade av ABB och därför används deras programmeringsspråk som heter RAPID [12].

6.2.1 RAPID

Programspråket RAPID är uppbyggt utav moduler, system- och programmoduler, som i sin tur är uppdelade i rutiner [12]. Den programdata som används är numeriska värden i form utav heltal och används exempelvis till att definiera en punkt i ett koordinatsystem men även booleska uttryck används.

Systemmoduler är de övergripande modulerna som innehåller information som är grundläggande för robotens installation som exempelvis servicerutiner eller verktygsdefinitioner. Systemmodulerna finns alltid lagrade i minnet.

Programmoduler innehåller kod som beskriver robotens rörelser. Ett robotprogram brukar oftast bestå utav en huvudmodul som kallar på flera submoduler för att dela upp programmet i mindre beståndsdelar. Varje programmodul är sedan ytterligare uppdelad i något som kallas rutiner som exempelvis kan vara en specifik rörelse. En typisk uppdelning kan ses i figur 6.13.



Figur 6.13: *Illustrerar en typisk programuppdelning i RAPID*

Den kod som beskriver robotens rörelser kallas för instruktioner och anropas från de olika rutinerna [12]. Exempel på instruktioner är:

- **Move J**
  - Rörelseinstruktion där roboten rör sig på snabbaste och bästa, för robotens leder, sätt till önskad position, där J står för Joint.

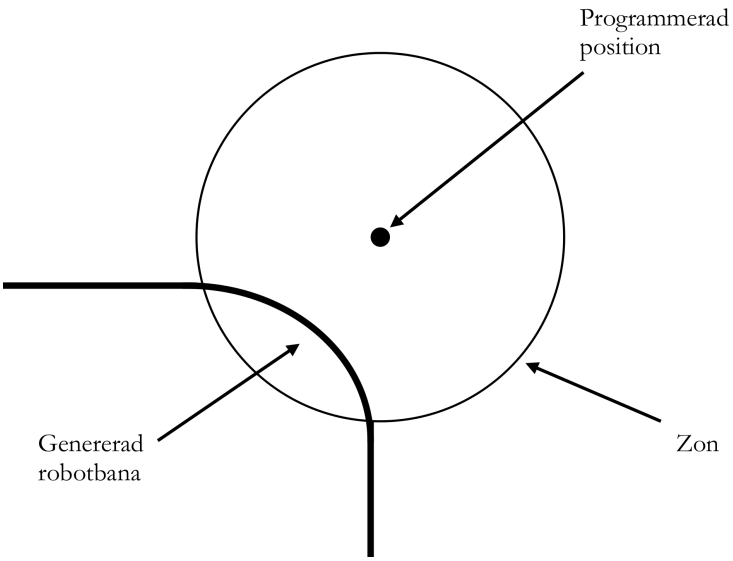
- **Move L**
  - Rörelseinstruktion där roboten rör sig linjärt till önskad position, där L står för Linear
- **SetDO**
  - Instruktion för att ett- eller nollställa en digital utgång. Exempelvis starta en digital signal som startar sugverktygets vakuum.
- **WaitTime/WaitUntil**
  - Instruktion för att få roboten att vänta en viss tid respektive vänta tills ett uttryck blir sant.
- **IF/While/For**
  - Olika typer av instruktioner för att exempelvis exekvera en sekvens beroende på om en variabel är sann eller falsk, repetera en sekvens till uttrycket blir falskt respektive repetera en sekvens ett bestämt antal gånger.
- **TPWrite/TPRead**
  - Instruktion för att skriva till robotens styrenhet, även kallad *FlexPendant*, respektive läsa av ett val som utförs av en operatör på styrenheten.

För att utföra en instruktion behövs ett antal argument och ett exempel på en rörelseinstruktion kan ses i ett exempel 6.1 nedan.

Exempel 6.1

MoveL, position1, v100, z10, tool1\WObj:=workobject1;

- MoveL – Själva instruktionen som ska utföras
- position1 – Positionen som roboten ska röra sig mot
- v100 – Hastigheten som roboten ska röra sig
- z10 – Hur exakt robotens bana måste vara när roboten övergår mellan instruktioner. Alltså inom vilket område som roboten får ”ta en genväg” och skapa en genererad bana. Detta används för att roboten tar mindre skada vid genererade banor än exakta banor, då roboten undviker start & stopp–rörelser och utför en mjukare rörelse. Även kallat ”fly-by point”. Se figur 6.14.
- tool1 – Robotens aktuella verktyg och alltså det aktuella koordinatsystem som roboten rör sig inom. Robotens koordinatsystem förklas mer under nästkommande avsnitt.
- WObj:=workobject1 – Arbetsobjektets koordinatsystem. Detta är ett valbart argument och vid användning uttrycks den aktuella positionen i arbetsobjektets koordinatsystem.



Figur 6.14: *Illustration över zoner vid robotprogrammering*

6.2.2 Online/Offline

Det finns två huvudsaklig tillvägagångssätt för att programmera en industrirobot. Programmeringen kan utföras både online samt offline [25].

Onlineprogrammering innebär att all programmering utförs manuellt i styrenheten [5]. I denna styrenhet finns alla instruktioner lagrade och måste programmeras för hand i rätt sekvens. Positionerna som roboten ska röra sig emellan sparas genom att manuellt, med handkontrollen, manövrera roboten till varje enskild punkt och där lagra den i programmet. Av detta skäl måste därför produktionen stoppa vid exempelvis beredning av nya produkter i produktionen, vilket är kostsamt [25]. Denna typ av programmering kan vara fördelaktig vid enklare processer och fördelen är att det inte kräver hög kunskapsnivå inom programmering dock lämpar det sig inte vid stora och komplexa processer [25].

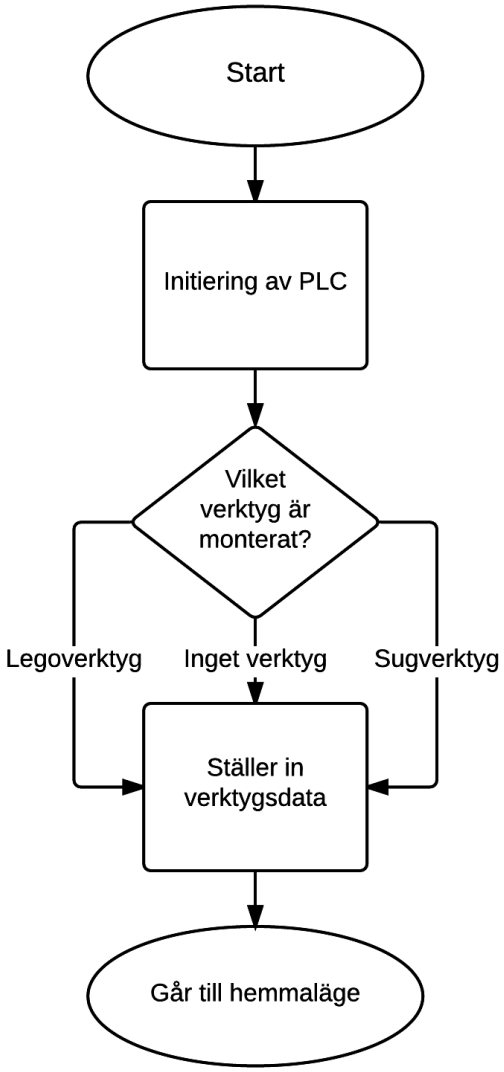
Offlineprogrammering däremot tillåter produktionen att fortgå medan programmering sker. Offlineprogrammering kan exempelvis ske i ABB:s programvara, *RobotStudio*, som läser in RAPID-koden. Detta ger en bättre översikt än fallet med onlineprogrammering som endast utförs genom styrenheten. Inom offlineprogrammering finns det även möjligheter att skapa en virtuell modell i form av en arbetsstation och på så sätt skapa programmen genom simulering, vilket ger en hög flexibilitet vid förändringar [25]. Genom detta kan tiden då produktionen står still reduceras med upp till 85 % jämfört med onlineprogrammering [5]. I den virtuella miljön kan även program simuleras utan risk för kollision eller personskador, dock kräver detta att den virtuella miljön stämmer väl överens med den fysiska.

6.2.3 Resultat

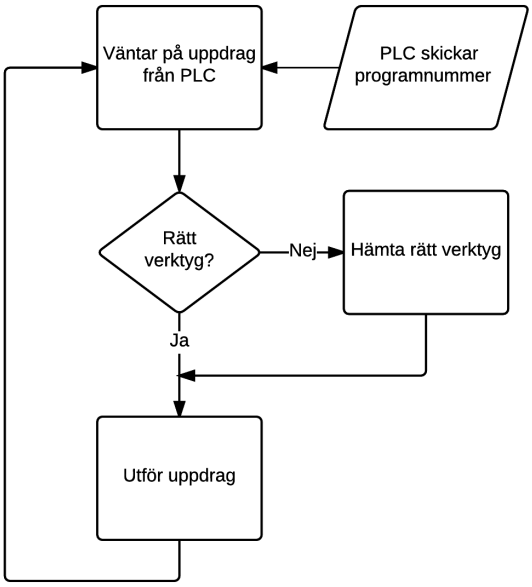
I detta avsnitt kommer det redogöras för resultatet och implementeringen för styrning av robotarna. Föregående års kandidatarbete har legat som grund för utvecklingen och implementeringen av de vidareutvecklade robotprogrammen [26].

6.2.3.1 Robotprogrammens uppbyggnad

Robotprogrammen som skapats är uppdelat i ett antal olika moduler som var för sig är ansvariga för olika delar i programmet. En huvudmodul initierar processen genom en manuell start via styrenheten. För en mer utförlig förklaring av den manuella initieringen se Appendix A. Huvudmodulen anropar sedan ett antal submoduler som initierar PLC-kommunikationen samt definierar vilket verktyg som roboten är utrustad med samt sätter roboten i hemmaläge. När detta är utfört går programmet in i en loop där programmet väntar på att PLC:n ska skicka ett uppdrag till roboten. När ett uppdrag skickas till roboten, i form av ett programnummer, utförs uppgiften och programmet fortsätter sedan i samma loop och väntar på nytt uppdrag ska ges från PLC-enheten. Arbetsgången för den manuella initieringen kan ses i figur 6.15 och programloopen med order från PLC kan ses i figur 6.16.



Figur 6.15: *Illustration av den manuella initieringen*



Figur 6.16: *Illustration av programloopen*

I loopen som programmet befinner sig i när den inväntar uppdrag från PLC:n är uppbyggt enligt figur 6.17. Denna loop anropar i sin tur de definierade uppdragsrutinerna som ligger i uppdragsmodulen samt ger varje uppdrag ett specifikt programnummer. Programnumren motsvarar de *Case* som syns i programmet.

Alla robotens rörelser är definierade i rutiner och de beskriver en robotoperation och dess rörelser. Rutinerna är placerade i den så kallade uppdragsmodulen i det utvecklade robotprogrammet. För att få ett flexiblare program utgör varje rutin en kort rörelse för roboten för att sedan möjliggöra kombination av dessa till fullskaliga program. Robotarna rör sig alltid mellan sina respektive väntelägen, vilket betyder att de alltid startar och slutar sitt uppdrag i någon av dessa positioner. Väntelägena kan ses i tabell 6.2.

Tabell 6.2: *Robotarnas väntelägen*

Robot	**Väntelägen**
R2	Hemmaläge
	Flexlinkposition
	Fixturposition
	Avlastningsbord 1/Table 1-position
R4	Hemmaläge
	Avlastningsbord 1/Table 1-position
	Fixturposition
R5	Hemmaläge
	Avlastningsbord 2/Table 2-position
	Fixturposition

```
22  PROC GorUppdrag(num Uppdrag)
23    !*********************************************************
24    !Funktion: Väljer uppdrag efter uppdragsnummer från PLC
25    !*********************************************************
26    !
27    ! Sätter att roboten lämnar hemmaläge
28    !
29    !
30    ! Väljer uppdrag efter uppdragsnummer från PLC
31    TEST Uppdrag
32
33  CASE 20:
34        IF DOutput(R2UT_HomePos)=1 THEN
35            home_to_flexlink;
36        ENDIF
37  CASE 21:
38        IF DOutput(R2UT_HomePos)=1 THEN
39            home_to_fixture;
40        ENDIF
41  CASE 22:
42        IF DOutput(R2UT_FlexlinkPos)=1 THEN
43            pick_carfloor;
44        ENDIF
45  CASE 23:
46        IF DOutput(R2UT_FlexlinkPos)=1 THEN
47            pick_roof;
48        ENDIF
49  CASE 24:
50        IF DOutput(R2UT_FixturePos)=1 THEN
51            mount_floor;
52        ENDIF
53  CASE 25:
54        IF DOutput(R2UT_FixturePos)=1 THEN
55            mount_roof;
56        ENDIF
```

Figur 6.17: *En del av programloopen som påvisar dess uppbyggnad*

Rutinerna är uppbyggda av ett antal instruktioner beroende på uppdrag och är uppdelade i ett antal olika kategorier:

- **Pick-rutiner**
  - Rutinerna används för att plocka upp en specifik komponent från flexlinkbanan alternativt Avlastningsbord 1 och är namngivna i koden enligt pick_komponentnamn exempelvis *pick_roof*.
- **Mount/Place-rutiner**
  - Rutiner används för att montera en karosskomponent i fixturen, placera en legomodul på Avlastningsbord 1 eller placera den färdigmonterade modellbilen på Avlastningsbord 2. De är namngivna i koden enligt mount/place_komponentnamn, exempelvis *mount_roof*
- **Rörelserutiner**
  - Rutinerna används för att skapa en rörelse från ett vänteläge, X, till ett annat vänteläge, Y. De är namngivna i koden enligt X_to_Y, exempelvis *home_to_flexlink*.

Ett exempel på en rutin visas i figur 6.18. I den specifika rutinen kontrolleras det att rätt verktyg är monteras för att sedan utföra ett antal instruktioner. Instruktionerna motsvarar ett antal rörelser mellan olika fördefinierade positioner för att sedan sätta en digital signal till 1, vilket i detta fall motsvarar att sugverktyget aktiveras. Roboten utför sedan ytterligare några rörelseinstruktioner. De inledande och avslutande SetDO-instruktionerna är till för kommunikation med PLC:n och förklaras i avsnittet 6.2.3.3.

```
303 PROC pick_rightcarside()
304 SetDO R2UT_HomePos,0;
305 SetDO R2UT_FlexlinkPos,0;
306 SetDO R2UT_FixturePos,0;
307 SetDO R2UT_Kand13_BordHemmalage,0;
308
309         IF DOutput(R2UT_Kand13_Sugverktyg)=0 AND DOutput(R2UT_Kand13_Gripper)=1 THEN
310             ToolChange_to_Sugverktyg;
311         ENDIF
312     MoveL flexlink_wait_pos, v600, z15, R2sug\WObj:=wobj0;
313     MoveJ flexlink_wait_pos20, v600, z15, R2sug;
314     MoveL pre_pick_rightside_pos, v600, z15, R2sug;
315     MoveL pick_rightside_pos, v100, fine, R2sug;
316         SetDO DO10_6,1;
317         WaitTime 0.5;
318     MoveL pre_pick_rightside_pos, v600, z30, R2sug;
319     MoveL flexlink_wait_pos20, v600, z15, R2sug;
320     MoveJ flexlink_wait_pos, v600, z15, R2sug\WObj:=wobj0;
321
322 SetDO R2UT_FlexlinkPos,1;
323 ENDPROC
```

Figur 6.18: *Exempel på en uppdragsrutin*

6.2.3.2 Verktygsbyte

På grund av att modellbilens karosskomponenter och legomoduler har olika dimensioner behöver roboten olika verktyg för att kunna hantera dessa, vilket är beskrivet i avsnitt 4.4. För att möjliggöra en flexibel produktion där produktionscellen kan montera hela modellbilen utan stopp för verktygsbyte har det utvecklats ett automatiskt verktygsbyte för robot R2.

För att möjliggöra det automatiska verktygsbytet har dels ett befintligt stativ förbättrats i produktionscellen, som är i form av två dockningsstationer som placerats i närheten av roboten. Det har även utvecklats två uppdrag som har skapats som rutiner i uppdragsmodulen för att styra roboten:

- Verktygsväxling till Sugverktyg, namngivet i koden som *ToolChange_to_Sugverktyg*
- Verktygsväxling till Gripverktyg, även kallat Legoverktyg. Namngivet i koden som *ToolChange_to_Legoverktyg*

För att roboten ska veta vilket verktyg som är monterat sätts digitala variabler vid bytet. Detta innebär att när exempelvis gripverktyget är placerat i dockningsstationen sätts

dess variabel till falsk och när sugverktyget är monterat, sätts dess variabel till sann, och motsvarande vid motsatt byte. Dessa variabler används sen vid detektering av vilket verktyg som är monteras på roboten. Inför varje robotoperation som innebär att hämta en komponent testas detta i koden så att rätt verktyg är monteras och således eliminerar risken för fel i monteringsprocessen samt eventuell skada, vilket riskeras genom att utföra en operation med fel verktyg. Dock kan verktygsbytet även initieras av en PLC genom operationer som utvecklats för att kunna optimera produktionssystemet.

6.2.3.3 Robot-PLC–kommunikation

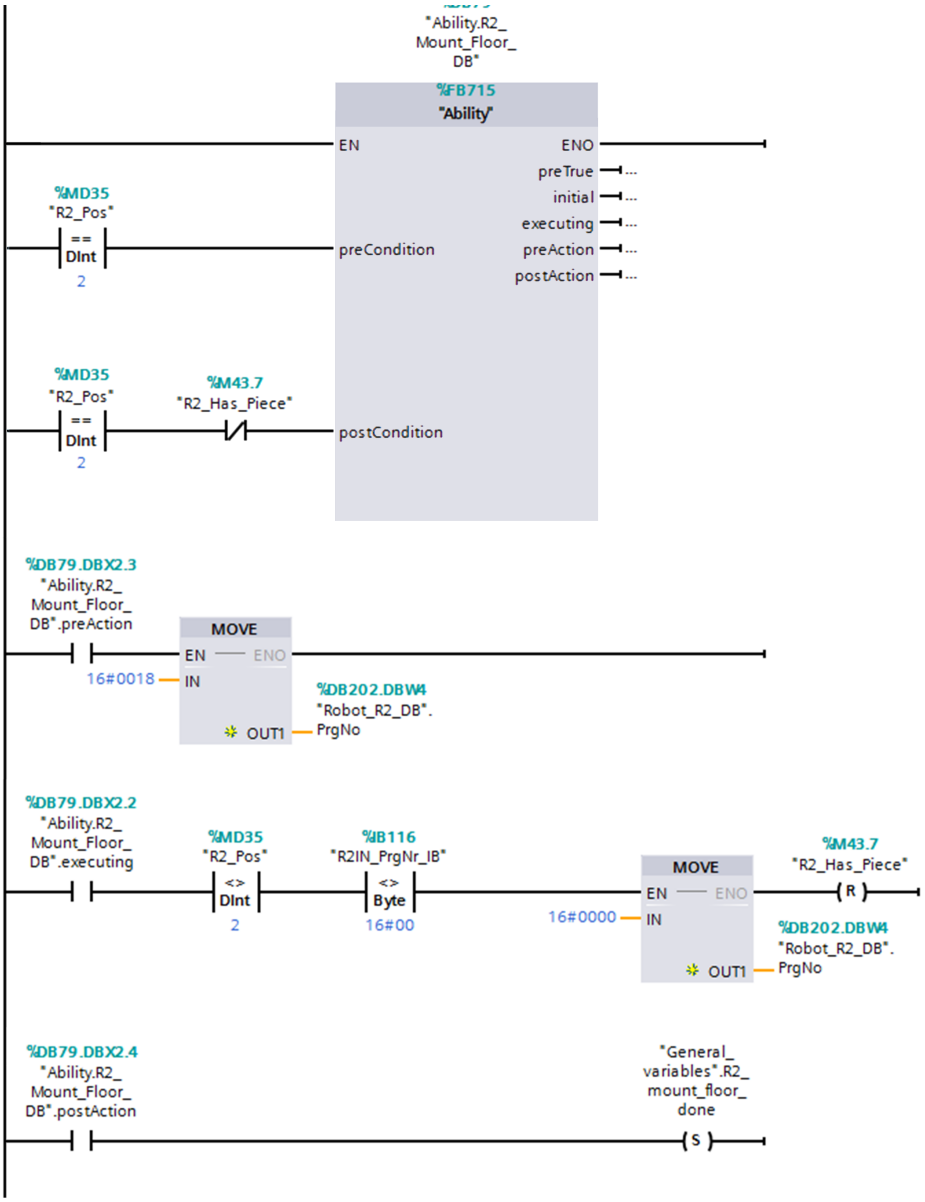
För att kunna kommunicera mellan den styrande PLC:n och robotarna måste signaler mellan dessa enheter skickas. Programmen är uppbyggda med ett antal väntelägen som beskrivits i avsnitt 6.2.3.1. För alla dessa väntelägen har en digital utgång definierats som är uppkallad efter det aktuella vänteläget. Om en utgång sätts till 1, alltså sann, genom en SetDO-instruktion motsvarar det att roboten har anlänt i det angivna vänteläget. På samma sätt motsvarar 0, alltså falsk, att roboten har förflyttat sig från det angivna vänteläget.

Rutinerna är uppbyggda så att de startar med att nollställa alla väntelägen, vilket symboliserar att roboten är i exekveringsläge. Anledningen till att alla väntelägen nollställs är att de olika rutinerna ska kunna exekveras utan inbördes ordning. Varje rutin avslutas sedan med att sätta en digital signal som symboliserar vilket vänteläge roboten befinner sig i efter utfört uppdrag, vilket visas i figur 6.18. Fördelen med att arbeta med väntelägen är att dessa signaler kan användas som olika typer av villkor för den styrande PLC:n, alltså som Pre- och Postconditions. Exempelvis för att kunna kontrollera att roboten står i en viss position innan PLC:n skickar ett programnummer för att göra ett specifikt uppdrag som kräver att roboten är i den positionen. Med andra ord skulle ett sådant villkor kunna vara att roboten måste stå i fixturposition för att kunna få uppdraget att montera golvet i fixturen.

För varje rutin i robotprogrammet som beskriver en robotoperation har en *Robot Ability* skapats i PLC-styrningen för att åstadkomma kommunikation mellan dessa. På grund av den standard som utvecklats är alla skapade på ett liknande sätt och de visas i figur 6.19.

Alla Robot Abilities består utav fyra Network. Det första består utav själva Ability-blocket där Pre- och Postcondition definieras och de består alltid utav villkor för i vilket vänteläge roboten ska befinna sig i sitt start- respektive slutläge i form utav variabeln *R2_Pos*. Är det en Mount- eller Pickrutin finns även en variabel som kontrollerar att roboten har respektive inte har en komponent. I nästa Network utnyttjas en av Ability:ns utsignaler, i detta fallet *preAction*, för att skicka ett programnummer till roboten. Programnumret skrivs alltid hexadecimalt och de anropar då de olika ”Case” som är definierade i robotprogrammet. När roboten får ett programnummer startar exekveringen och roboten speglar då tillbaka det programnummer som angivits. Robot Ability:n har då gått in i sitt tillstånd *executing* och aktiverar det tredje Network:et. Detta kontrollerar att det speglade programnumret är skiljt från noll, vilket visar att roboten börjat exekvera. När detta är uppfyllt skickas programnummer noll för att säkerställa att roboten bara utför dess uppdrag en gång. När roboten utfört sitt uppdrag kommer den stå i det vänteläge som är definierat som slutvillkor i *postCondition*. Då kommer Robot Ability:n gå in i tillståndet *postAction* och in i sitt fjärde Network. Där sätts en kontrollvariabel till sann som senare kan utnyttjas i

den övergripande styrningen för att veta att roboten utfört sitt uppdrag.



Figur 6.19: *Illustration av en Robot Ability, Mount_Floor, som möjliggör kommunikation mellan robotarna och PLC:n för att montera modellbilens golv*

Det förekommer många robotoperationer i ett automatiserat produktionssystem såväl i industrin som i detta projekt. I den utvecklade styrningen för produktionscellen finns ett antal robotoperationer, *Robot Abilities*, varav 26 stycken för robot R2, tolv för R4 respektive sex för R5 som alla tillsammans möjliggör kommunikation mellan PLC:n och robotarna för att kunna montera modellbilen. På grund av att styrningen bygger på den utvecklade standarden ser alla ut på ett liknande sätt. Detta är då ett kodsegment som är repetitivt förekommande och således tidskrävande att programmera. Det är därför intressant att undersöka möjligheterna att minska tiden det tar att skapa kodsegmentet för att spara tid i utvecklingsprojekt. Därför är det de ovan beskrivna kodsegmentet som kapitel 7, Automatisk kodgenerering, kommer att behandla.

7

Automatisk kodgenerering

En av de vanligaste metoderna som används vid framtagning av styrlogik för en PLC, är att programmeraren själv designar programmet mot en teknisk specifikation som beskriver programmets önskade och oönskade funktionalitet [27]. Den här metodiken fungerar givetvis bra mot mindre projekt då en platt kodstruktur är möjlig, men blir problematisk vid utveckling av större projekt. Utvecklingen av koden sker ofta i lag, vilket gör det svårt att få en övergripande struktur över projektet. Det här leder i sin tur till att felsökning av programmet blir av en onödig komplexitet och upptar således stora delar av utvecklingstiden. Ett sätt att bli av med delar av de problem som orsakas av mänskliga faktorer samt att få en övergripande struktur över projektet är att programmerarna arbetar mot tydligt angiven standard, vilket utvecklats i kapitel 5.

För att stödja samt utveckla nuvarande programmeringsmetodiker har automatisk kodgenerering vuxit fram som ett allt större forskningsområde [28]. Mycket av den forskning som bedrivits inom området använder sig utav högnivåprogrammering med diskreta eventsystem som ett centralt verktyg för att generera kod enligt IEC 61131-3-standarden [29, 27, 30, 28].

Även om forskningsområdet inom automatisk kodgenerering har vuxit, har det ej ännu resulterat i något större industriellt genomslag. En anledning till det är att export- och importfilerna ter sig olika i de programvaror som tillåter denna funktionalitet. Några som försöker åtgärda detta problem är PLCopen som möjliggör export- och import av projekt mellan programvaror som ursprungligen inte är kompatibla med varandra [31].

Det finns en stor potential inom automatgenerad kod, framförallt på grund av de minskade utvecklingstider som kan uppnås. Hos företag med väldefinierade standarder ges programmerarna möjlighet att återanvända stora delar av koden. Vilket i sin tur leder till att utvecklingstiden kan minskas avsevärt. För större projekt så går det inte att bortse från att tiden som får ägnas åt den här typen av ”copy & paste programmering” är av betydande art. Här finns stora möjligheter att automatgenerera den här typen av kodsegment och således minska utvecklingskostnaderna. Dock kan dessa problem ofta återkomma vid högnivåprogrammeringen, även vid användning av objektorienterade språk och dess verktyg. En lösning är att använda sig utav aspektorienterad programmering som presenterades av Bengtsson, et.al. [32].

Med användning av aspektorienterad programmering under utvecklingen kan den återkommande koden istället utgöras av aspekter och således skapa en effektivare kod som både är mer överskådlig samt mer flexibel för ändringar [32]. En annan fördel med aspektorienterad programmering som ett utvecklingsverktyg är att olika delar i koden inte behöver separeras som exempelvis styrning och säkerhet, istället kan säkerheten implementeras som aspekter i koden som berör styrningen [32].

7.1 Kommunikation med TIA-Portalen

För att möjliggöra kodgenerering från en extern utvecklingsmiljö krävs det att en kommunikation mellan TIA-Portalen och den externa utvecklingsmiljön upprätthålls.

7.1.1 Openness

För att uppnå kommunikationen mellan kodgeneratorn och TIA-Portalen har ett API, *Aplication Programming Interface*, från Siemens vid namn Openness använts. API:t tillhandahåller funktioner för kommunikation med TIA-Portalen via en C#-applikation.

Med hjälp utav Openness, kan således ett program för kommunikation som möjliggör extern körning av projekt samt import och export av olika datatyper till TIA-Portalen konstrueras. Kommunikationen för export och import sker via dokument av typen XML, *Extensible Markup Language*, som håller all nödvändig information för de olika filtyperna.

7.1.2 XML

XML utgör en generell struktur för att organisera data så att dess struktur både är överskådlig för människor och likväl för datorer. XML växte fram som en förenkling av språket SGML, *Structured General Markup Language*, som utvecklades av Goldfarb, Mosher och Lorie från IBM under 1970-talet [33].

7.1.2.1 Grundläggande struktur

De grundläggande beståndsdelarna i ett XML-dokument utgörs av så kallade element som även kan betecknas som noder vid beskrivning av XML-dokumentens hierarkiska struktur [33]. Ett element består av en start- och en slut-tag där information kan kapslas in mellan de två elementen enligt följande:

`<Tagnamn>Information</Tagnamn>.`

XML-språket tillåter även inkapsling av andra element vilket i sin tur möjliggör en hierarkisk strukturering av de olika elementen. De olika elementen får dock inte överlappa varandra, utan de inkapslade elementens sluttagar måste deklareras innan sluttagar från element högre i hierarkin[34]. Då ett element ej har någon underliggande hierarkisk struktur kan start- och sluttagen kombineras enligt:

`<Tagnamn/>.`

Informationen kan även lagras inom så kallade attribut där informationen tilldelas en identifierare och dess värde syns inom citationstecken. Användningen utav attribut i synnerhet i botten av den hierarkiska strukturen kan ofta bidra till att den intuitiva förståelsen för strukturen ökar och kan exempelvis deklareras inom dokumentet enligt:

`<Tagnamn ID="Information"/>.`

För att undvika semantiska konflikter vid inläsning av data från en mottagare som hanterar flera namnrymder kan de berörda elementen kapslas in inom en namnrymd för att undvika problemet [33]. Element som härstammar från en annan namnrymd kan då specificeras med ett prefix enligt exempel 7.1.

Exempel 7.1: Ett XML-dokument som berör mer än en namnrymd

```
<Figurer xmlns="URI"
xmlns:utseende="URI">
  <Rektangel>
    <Höjd>2cm</Höjd>
    <Bredd>4cm</Bredd>
    <utseende:Färg>Röd</utseende:Färg>
  </Rektangel>
</Figurer>
```

I exemplet ovan är elementen inkapslade i två stycken namnrymder där de element med ej tillhörande prefix tillhör den första deklarerade namnrymden. URI, *Uniform Resource Identifier*, inom namnrymdsdefinitionen används för att identifiera den givna namnrymden och är oftast bestående av en URL, *Uniform Resource Locator*, där ofta information om den givna namnrymnden tillhandahålls [33].

7.1.3 C#

För att använda den funktionalitet som Openness tillhandahåller har C# använts som programmeringsmiljö för utvecklingen av de program som rör kodgenereringen.

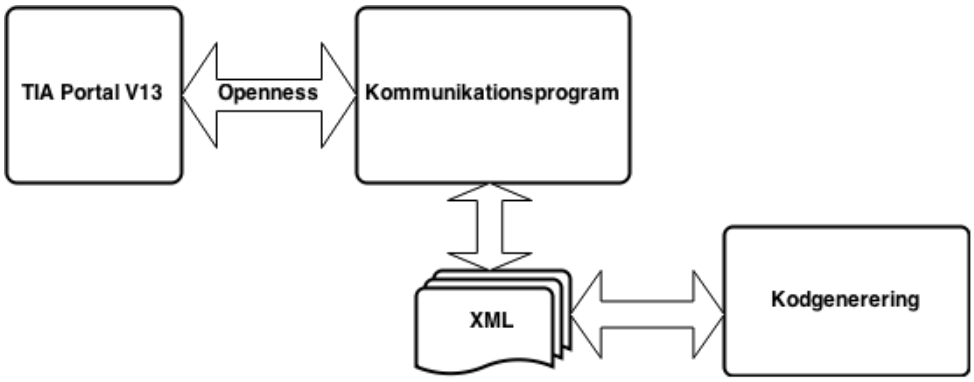
C# är ett objektorienterat programmeringsspråk som utgör en del av Microsofts stora ramverk .NET [35]. C# har sina rötter i C-familjen av programmeringsspråk vilket underlättar utvecklingen för personer med tidigare erfarenheter från C, C++ eller Java [36]

En vanlig uppfattning av C# är att språket är plattformsberoende. Vilket det till en början även var, men det har numera inom fristående Mono-projekt utvecklats versioner som är kompatibla med andra operativsystem än Windows [35].

7.1.4 Resultat

Med hjälp av Openness har ett program för kommunikation med TIA-Portalen utvecklats. Programmet fungerar som ett komplement till den nya utvecklingsmiljön som konstruerats för extern utveckling av funktionsblock. I detta kapitel kommer Funktioner att benämnas som Funktionsblock. För en illustration över hur kommunikationen mellan de olika programvarorna se figur 7.1.

Det utvecklade programmet ansluter till Siemens TIA Portal vid programstart. Från startfönstret kan det önskade projektet sedan öppnas. Då ett projekt valts listas de PLC-enheter som finns tillgängliga inom projektet. Därifrån väljs sedan den PLC-enhet



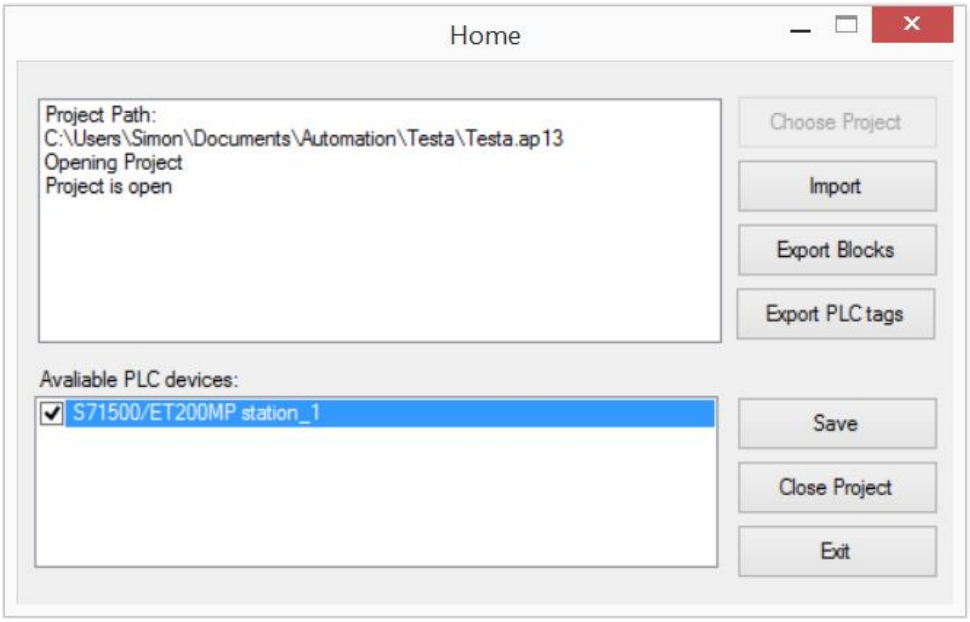
Figur 7.1: *Överblick av kommunikationen mellan programvarorna*

som kommunikationerna ska rikta sig mot. I tabell 7.1 listas de funktioner som implementerats i programmet för olika datatyper. Programmet läser och sparar data i form av XML-dokument på en önskad plats på hårdisken och möjliggör således extern manipulation av de olika datatyperna. Alla funktioner som berör import och export har tilldelats egna programfönster, för att göra programmet mer intuitivt samt för att öka användarvänligheten.

Tabell 7.1: *Tillgängliga funktioner för olika datatyper i kommunikationsprogrammet*

Data Type	**Import**	**Export**
Database	*Yes*	*Yes*
Function Block	*Yes*	*Yes*
PLC Tags	*No*	*Yes*

Programmets startfönster kan ses i figur 7.2. Figurer över andra programfönster samt en programbeskrivning kan ses i Appendix D.



Figur 7.2: *Startfönster för kommunikationsprogrammet*

7.2 Kodgenerering

För att utvärdera autogenerering av PLC-logik, har det utvecklats ett program för automatisk kodgenerering baserat på grafteorier med influenser från aspektorienterad programmering.

7.2.1 Avbildningstabeller & klassen Dictionary

Avbildningstabeller används ofta för konstruktion av olika grafer och datastrukturer. Klassen Dictionary används inom programmeringsspråket C# för att konstruera olika avbildningstabeller. En avbildningstabell utgörs av nycklar och värden. Varje nyckel pekar mot ett värde och utgör med tillhörande värde en så kallad avbildning, *key-value pair* [35]. För att konflikter ej ska uppstå måste alltså varje nyckel vara unik inom avbildningstabellen. En illustration över en avbildningstabell kan ses i tabell 7.2, där personnummer har angivits som nycklar för att peka på en individ.

Tabell 7.2: *En illustration av en avbildningstabell*

Nyckel	Värde
19440404-4444	Anna Andersson
19770707-7777	Bertil Bertilsson
19550505-5555	Carl Carlsson

Dictionary är en generisk klass vilket innebär att den inte är bunden till en begränsad mängd datatyper. Den tar två olika typparametrar vid deklarationen som specificerar datatypen för nyckeln respektive värdet. I det enklare exemplet 7.2 illustreras hur Dictionary kan användas för en avbildning.

Exempel 7.2

```
Dictionary<string, int> map = new Dictionary<string, int>();
int age = 71;
string name = "Anna Andersson";
map.Add(name, age);
Console.Out.Write(map[name].ToString());

Output: 71
```

Ovanstående kodexempel tar alltså namnet Anna Andersson som nyckel till avbildningstabellen och åldern 71 som dess värde. Då nyckeln *name* sedan anges till avbildninstabellen skrivs Annas ålder ut. Hädanefter kommer den engelska terminologin maps att användas för att benämna avbildningstabeller.

7.2.2 Aspektorienterad programmering

Objektorienterade programmeringsspråk är i dagsläget de språk som förekommer mest frekvent. Även om objektorienterade språk har funktionaliteter såsom *inkapsling*, *inheritance*

och *polymorphism* för att effektivisera koden samt att göra de olika programmodulerna lättare att återanvända, har de sina brister [37].

Inom större objektorienterade projekt så är vissa funktioner benägna att spridas över flertalet olika klasser. Dessa funktioner brukar benämnas som korsrelaterade. En följd av en hög andel korsrelaterade funktioner inom ett projekt är att det blir svårt att separera de olika modulerna inom programmet samt att projektet innefattar en hög andel repetitiv kod [37].

Aspektorienterad programmering löser de ovan nämnda problem som kan uppstå med den objektorienterade programmeringen, genom att dela in de funktioner som uppkommer frekvent inom olika delar av programmet i aspekter [32]. Fördelen med den aspektorienterade programmeringen är att den tar dessa korsrelaterade funktioner och delar in dem i olika aspekter där en korsrelaterad händelse utgörs av en aspekt. Det här medför att endast en instans av kod behöver skrivas för att uppfylla en händelse som annars hade kunnat vara spridd över flertalet olika klasser [32, 38, 37]. Aspekten vävs sedan in i de punkter i under kompilering vid så kallade Join Points som i de flesta fallen utgör de punkter i koden där händelser av korsrelaterande betydelse behöver utföras [38]. En kortfattad beskrivning av aspektorienterad programmerings grundläggande terminologi och struktur är:

- **Join Points**
  - Är en väldefinierad punkt i programstrukturen där ytterligare funktionalitet kan tilläggas [39]. En Join Point kan exempelvis utgöras av en metod eller ett funktionsanrop.
- **Weaving**
  - Är processen där programmets kärnfunktionalitet vävs samman med de olika aspekterna för att således generera ett fungerande system [39].
- **Point Cuts**
  - Kvantifierar eller anger en uppsättning av Join Points [32].
- **Advice**
  - Beteendet som ska exekveras vid eller omkring en Join Point [39].

7.2.3 Resultat

Genom att den nya standarden som utvecklats ger en tydlig struktur över de olika programmen, skapas det även vissa aspekter inom koden som förekommer med högre frekvens än övriga. Det har kunnat urskiljas att de kodsegmenten som förekommer vid kommunikationen med ABB-robotarna har förekommit med väldigt hög frekvens. Därav har programmet inriktats mot generering av de aspekter som rör robotkommunikationen. I den nya standarden sker robotkommunikationen via Robot Abilities som kan ses i figur 6.11.

7.2.3.1 Nytt programmeringsspråk för PLC-logik

För att göra den grafiska delen av programmet mindre omfattande har det utvecklats ett nytt språk för att illustrera ladderlogik. Ladderlogiken exekveras i topologisk ordning det vill säga att koden exekverar ut från nätet *Powerrail* i ett Network. I figur 7.3a exekveras alltså ett Network från nätet som utgörs av den vertikala linjen i vänstra delen av figuren till operatorerna i den högra delen av figuren. Eftersom ladderlogiken exekverar i topologisk ordning kan ett Network likställas med en DAG, *Directed Acyclic Graph*. Språket som använts i utvecklingsmiljön har därför designats för att efterlikna de övergångar som sker till och från en grafs olika noder.

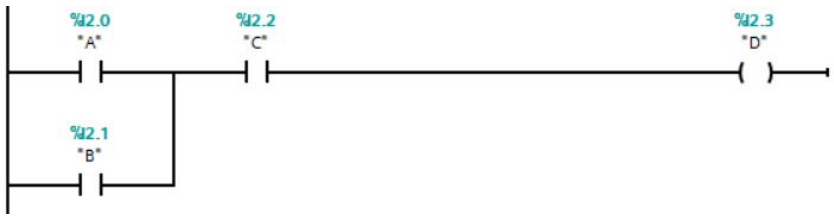
Exempel 7.3: En koppling mellan en kontakt med operanden A och en kontakt med operanden C uttrycks på följande sätt i den nya miljön

```
Contact(A) => Contact(C)
```

Kopplingarna sker alltid parvis och ger, som det ses i exempel 7.3, alltid en entydig riktning för övergången. För ytterligare illustrationer av kodstrukturen och jämförelser med ladderlogik se figur 7.3a och figur 7.3b.

```
Powerrail  => Contact(A)
Powerrail  => Contact(B)
Contact(A) => Contact(C)
Contact(B) => Contact(C)
Contact(C) => Coil(D)
```

(a) *Språket som används i den nya utvecklingsmiljön*



(b) *Ladderdiagram i TIA Portalen*

Figur 7.3: *Jämförelse av ett Network skrivet i språket som används i den nya utvecklingsmiljön och ladderdiagram*

7.2.3.2 Struktur på XML-filer

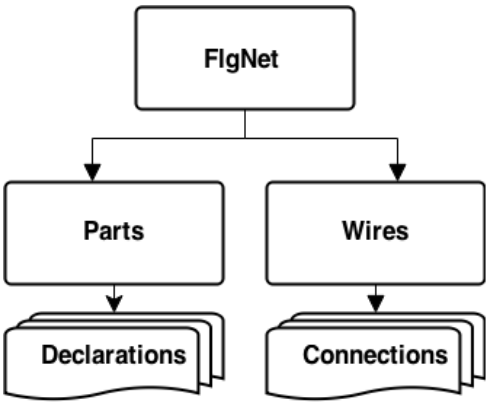
Funktionsblocken som skrivs i den nya utvecklingsmiljön behöver även struktureras och lagras i XML-dokument för att möjliggöra import till TIA-Portalen, se avsnitt 7.1.

I XML-filen finns det två segment av hierarkier under XML-dokumentets rot. I det första segmentet deklareras de installerade produkterna och produktversionerna som behöver vara installerade på måldatorn för import. De produkter som behövs för att importera filer från den nya utvecklingsmiljön finns listade i tabell 7.3.

Tabell 7.3: *Installerade produkter som behövs för import av filer*

Produkt	Version
Totally Integrated Automation Portal	V13 SP1 Update 1
TIA Portal Openness	V13 SP1
STEP 7 Professional	V13 SP1 Update 1
WinCC Basic	V13 SP1 Update 1

Det andra segmentet är det som utgör själva funktionsblocket. Inom detta ryms det olika element vid namn *SW.CompileUnit*. I ett sådant element deklareras alla element som rör ett Network. Inom *SW.CompileUnit* ryms elementet *FlgNet* där de två hierarkierna *Parts* och *Wires* finns som underliggande element. För en grafisk illustration över dessa hierarkier se figur 7.4.



Figur 7.4: *Hierarkierna för sammankoppling av noderna inom XML-dokumentet*

I *parts* deklareras alla variabler och operatorer samt tilldelas unika identifieringsnummer som senare används inom *Wires*-hierarkin. Variabler deklareras som element av typen *Access* med dess synlighet, datatyp och identifieringsnummer som attribut till elementet. Operatorer deklareras som en *Part* där dess attribut och eventuellt underliggande element kan variera beroende vilken typ operatorn utgörs utav. Värt att notera är att varje variabel och operator deklareras för varje instans som de förekommer i det aktuella Network som elementet *Parts* berör.

Exempel 7.4: Deklaration av de variabler som behövs för en kontakt med variabeln preAction från databasen Ability.R2_Home_to_Flexlink_DB som operand

```
<Parts>
    <Access Scope="GlobalVariable" Type="Bool" UId="40">
        <Symbol>
            <Component Name="Ability.R2_Home_to_Flexlink_DB" />
            <Component Name="preAction" />
        </Symbol>
    </Access>
    <Part Gate="kontakt" UId="41" />
</Parts>
```

I hierarkin inom elementet *Wires* kopplas de olika noderna samman med hjälp av de identifieringsnummer de blivit tilldelade i *Parts*. Varje övergång deklareras inom ett element vid namn *Wire*. Det är även viktigt att de olika elementen av typen *Wire* struktureras i filen efter den topologiska ordning som programmet sedan exekverar. Kopplingarna sker inom elementet *Wire* där variabler beskrivs som element av typen *IdentCon* och operatorer som typen *NameCon*. De basoperatorer som utgörs av en operand, uppkommer minst inom två *Wire*-element en för ingången och en för operanden. Alla operatorer som använts under projektet har endast en ingång från nätet. Om operatorn har mer än en ingång som i figur 7.3b sker kopplingen via en merger, som även den deklareras i *Parts*.

Exempel 7.5: Enklare sammankoppling av noder för ett Network med en kontakt samt en spole. I exemplet har kontakten tilldelats UID=41, spolen UID=43 samt två booleska variabler som tilldelats UID:s = 40 & 42.

```
<Wires>
   <Wire>
       <Powerrail />
       <NameCon UId="41" Name="in" />
   </Wire>
   <Wire>
       <IdentCon UId="40" />
       <NameCon UId="41" Name="operand" />
   </Wire>
   <Wire>
       <NameCon UId="41" Name="out" />
       <NameCon UId="43" Name="in" />
   </Wire>
   <Wire>
       <IdentCon UId="42" />
       <NameCon UId="43" Name="operand" />
   </Wire>
</Wires>
```

För ett komplett XML-dokument se Appendix H.

7.2.3.3 Programmering och algoritmer

Utvecklingen av den nya utvecklingsmiljön har skett i C# för att möjliggöra integration med programmet som berör kommunikationen med TIA-Portalen, se avsnitt 7.1.4.

För att skapa Siemens XML-struktur har klassen *XmlSerializer* använts under utvecklingen av programmet. *XmlSerializer* genererar en XML-dokument baserat på de underliggande klasshierarkier för en given klass inom projektet. Då de XML-dokument som genereras kräver omfattande och djupgående hierarkiska strukturer har *XmlSerializer* använts inom C#-projektet för att spegla dessa hierarkier.

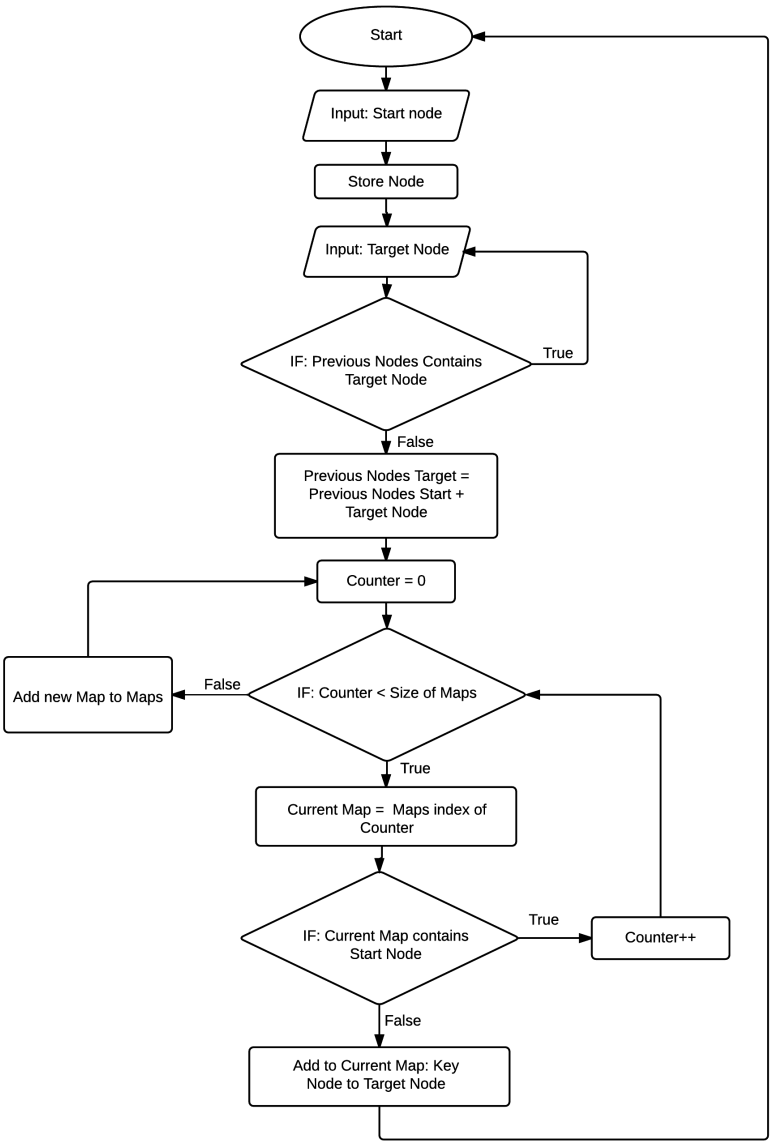
Beräkningar för att skapa grafer av typen DAG sker i klassen *Network*. Klassen *Network* består av lista med maps, avbildningstabeller, som utgör själva grafen. För att upprätthålla den topologiska ordningen innehåller *Network* en map som tar en nod som nyckel och pekar på en lista där föregående noder lagrats. Det skapas även en map där varje nod tilldelas ett identifieringsnummer, *UID*, som används senare vid sammankoppling av noderna i klasserna *Parts* och *Wires*.

Algoritmen för att sammankoppla noderna fungerar på så vis att användaren först får ange en startnod som *input* till programmet. Startnoden lagras sedan i klassen *Network*. Efter att startnoden har valts väntar programmet på att en målnod ska anges. Då målnoden valts undersöks först om den finns i listan av föregående noder för startnoden. Om så är fallet meddelas användaren att sammankopplingen är ogiltig och ombedes välja en ny målnod. Om valet av målnod godkänns skapas en ny avbildning för att säkerställa att

kommande sammankopplingar är av topologisk ordning. I den map som skapas utgör målnoden nyckeln som pekar på en lista med alla föregående noder samt målnoden själv.

Efter att målnoden har valts söker programmet igenom listan med maps för att se om det finns en map inte använder den aktuella startnoden som nyckel. Om så är fallet lagras avbildningen i den första map som inte innehåller startnoden som nyckel. Används startnoden som nyckel i alla maps lagras avbildningen i en ny map som sedan läggs till i listan med de övriga. Antalet listor blir således proportionellt med det högsta antalet utgångar en nod erhåller inom grafen. För en grafisk illustration över algoritmen se figur 7.5.

Då funktionsblocket ska genereras tar klasserna *Parts* och *Wires* ett objekt av *Network* som inparametrar till konstruktorn för att utföra de nödvändiga deklarationer av element som krävs för sammankopplingen av noderna.



Figur 7.5: *Algoritm för sammankoppling av noder*

7.2.3.4 Utvecklingsmiljön

För att underlätta utvecklingen kan de PLC-tags som exporterats via kommunikationsprogrammet, importeras och användas i databaser inom utvecklingsmiljön. Databaserna sorteras sedan efter olika datatyper och i utvecklingsmiljön väljs sedan den databasen som håller godkända datatyper för den valda operatorn. I tabell 7.4 listas alla operatorer med tillhörande datatyper.

Tabell 7.4: *Operatorer med tillhörande datatyper*

Operators	**Bool**	**Int**	**Dint**	**Word**
Contact	*Yes*	*No*	*No*	*No*
Negated Contact	*Yes*	*No*	*No*	*No*
Set Coil	*Yes*	*No*	*No*	*No*
Reset Coil	*Yes*	*No*	*No*	*No*
Equal	*No*	*Yes*	*Yes*	*Yes*
Not Equal	*No*	*Yes*	*Yes*	*Yes*
Move	*No*	*Yes*	*Yes*	*Yes*
Ability Block	–	–	–	–

I utvecklingsmiljön finns det två stycken templates att välja mellan, ett för en Robot Ability samt ett för en vanlig Ability. I det template som är avsett för en Robot Ability autogenereras de aspekter som rör robotkommunikationen. De autogenererade aspekterna kan ses i de tre nedersta Network:en i figur 7.6.

Utvecklingsmiljöns layout består primärt utav tre stycken fält, se figur 7.7. I det vänstra fältet deklareras alla övergångar mellan de olika noderna. I fältet i nedre högre hörnet aktiveras de Network som ska vara aktuellt för redigering och i det övre av de högra fälten visas de noder som är tillgängliga som startpunkter för nästkommande övergång.

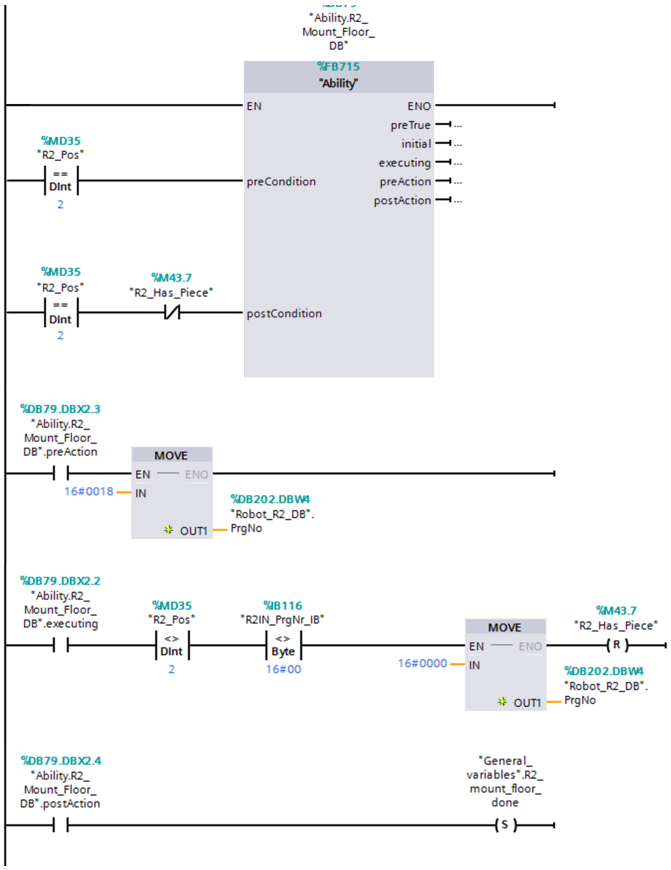
Programmeringen sker enligt det språket som nämnts ovan, där noderna skapas med hjälp av de operatorer som implementerats samt listan som tillhandahåller alla befintliga noder i grafen. Koden för respektive Network kapslas sedan in mellan två textrader som beskriver när Networket börjar samt slutar, vilket illustreras i exempel 7.6.

Exempel 7.6: Ett komplett Network i utvecklingsmiljön

```
Network 1 Start
Powerrail=> Contact(A)
Contact(A) => Coil(B)
Network 1 End
```

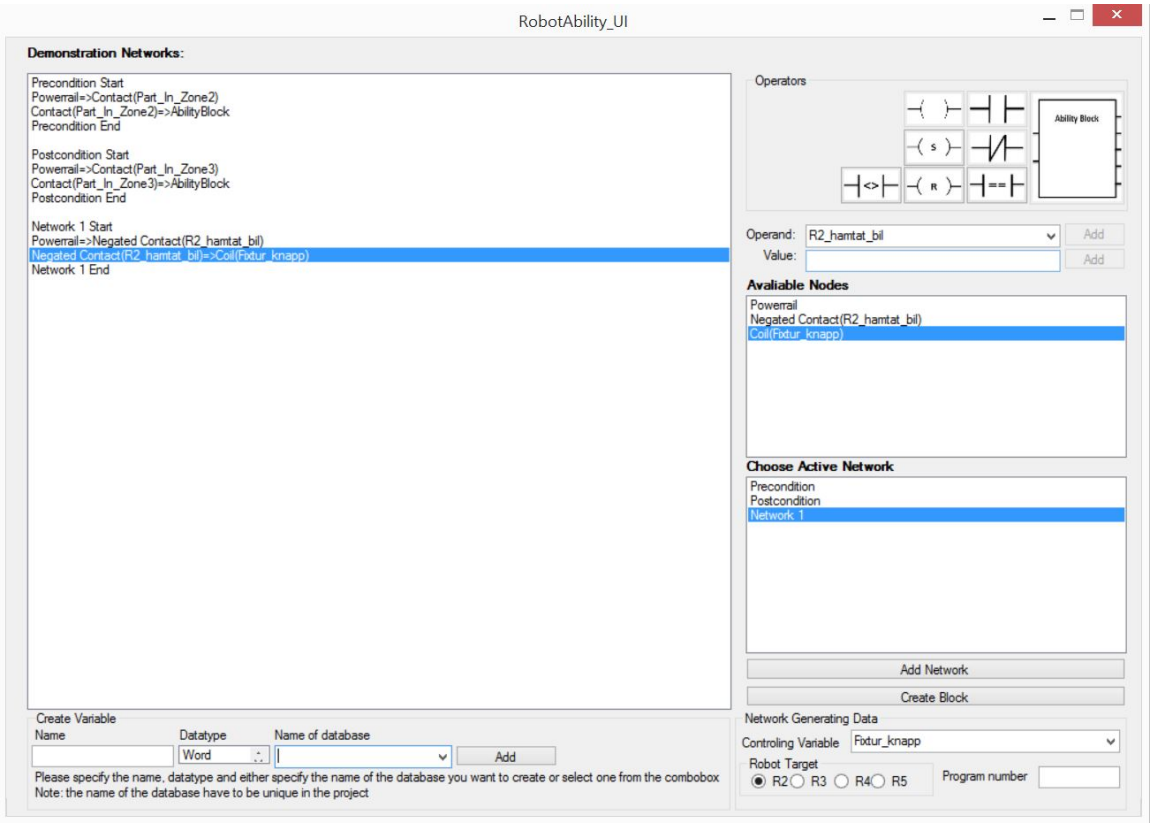
Det går även att konstruera och använda nya variabler och databaser i utvecklingsmiljön. Detta sker i den nedre delen av programfönstret och den nya databasen kommer generera ett separat XML-dokument som även den behöver importeras till TIA-Portalen för den ska finnas tillgänglig i det befintliga projektet. Varje databas som konstrueras utgörs av ett unikt XML-dokument som sedan kan importeras via det utvecklade kommunikationsprogrammet.

Vid utveckling inom det template som berör Robot Abilities behöver information anges för att de delar som berör robotkommunikationen ska autogenereras. Den utökade informationen som behöver anges är vilken robot koden är riktad till, robotens programnummer vilket



Figur 7.6: *Function som genererats från utvecklingsmiljön*

beskrivs i avsnitt 6.2.3.1 samt en kontrollvariabel som möjliggör övergripande styrning från Operations. *Weavingen* sker vid genereringen av funktionsblocket genom att de *Network* som berör aspekten för robotkommunikationen utgör inparametrar till konstruktorn för XML-dokumentets rot. Valet av *Join Point* sker automatiskt då de *Advices* som ska utföras placeras sist bland de Networks som konstruerats.



Figur 7.7: *Nya utvecklingsmiljön*

8

Diskussion

Syftet med detta kandidatarbete har varit att utveckla och verifiera ett koncept för hur industrin kan dra nytta utav den forsknings som bedrivits inom automation och i synnerhet att arbeta med operationer. Därför har en ny standard, med utgångspunkt från Volvo Car Corporation utvecklats. Med en tydlig standard är det sannolikt att en del kod blir repetitiv, därför har även en utvecklingsmiljö skapats för att bevisa potentialen i att automatisk generera kod för att reducera utvecklingstiderna. För att verifiera detta resultat har konceptet implementerats i PSL, där såväl hårdvaruuppdateringar samt robotstyrning varit nödvändigt för att ge en helhetsbild över hur konceptet fungerar. I detta kapitel kommer resultat diskuteras, eventuella problem beskrivas och hur de lösts samt redogöra för de utvecklingsmöjligheter som finns vid fortsatt arbete med konceptet.

8.1 Produktionssystemet

Den nya förstärkta modellbilen samt den konstruerade transportfixturen utvecklades för att öka noggrannheten i produktionssystemet samt för att öka hållfastheten, så att hårdvaran kan användas under en längre period. Vid utvärdering framkom ett antal problem som var tvungna att åtgärdas, bland annat problem med transportfixturen. Denna behövde justeras på grund utav att den fastnade i flexlinkbanan. Detta löstes genom att höja upp transportfixturen från paletten med hjälp utav två fyrkantsrör. Även takmodulen behövde justeras genom att höja modulen med nya hål för upphängning på fixturen. Den utvecklade profilen till modellbilen visade sig fungera väl. Genom dessa justeringar uppfyllde den nya hårdvaran sitt syfte, att öka noggrannheten i produktionssystemet samt att öka hållfastheten för modellbilen.

Ett annat problem var att på grund av tidsbrist tillverkades det bara en profil och en transportfixtur. För att göra en bättre utvärdering av dessa hade fler behövts tillverkas för att bättre kunna bevisa de nya konstruktionernas noggrannhet. Om fler profiler tillverkas kan de monteras på fler modellbilar vilket hade gjort att fler modellbilar hade kunnat användas vid exekvering av produktionscellen. Till följd av detta hade modellbilarna kunnat användas längre utan deformationerna som annars uppstår. Även fler transportfixturer kan tillverkas för att kunna bevisa potentialen i den nya konstruktionen som utvecklats. Detta hade exempelvis gjort att flertalet komponenter kan skickas in direkt för att optimera produktionsprocessen med avseende på bland annat tid.

8.2 Standard

Vid skapandet av den nya standarden lades grunden med implementeringen av Chalmers operationskoncept och efter mer än hälften av arbetets gång började Volvos standard att kombineras med det nya arbetssättet. Detta berodde delvis på brist av material från Volvo i början och delvis på att programmeringen av den nya PLC:n var tidskrävande och måste göras tidigt, innan litteraturstudien avslutats. Tack vare att Volvos standard och Chalmers implementerade operationskoncept styr olika delar av programmeringsprocessen var det möjligt att införa ändringar utifrån Volvos standard senare i processen. I efterhand kan det diskuteras huruvida den senare implementeringen av Volvos standard påverkade resultatet. Troligtvis är skillnaden inte stor men om den nya standarden är mindre applicerbar och tilltalande för Volvo så har detta arbetssätt motverkat huvudsyftet med projektet. Det som talar för att den nya standarden skulle kunna vara implementerbar i Volvos programmeringssätt är alla de likheter som upptäckts mellan Volvos program och programmen där den nya standarden införts. Volvos styrning har måhända tagits i beaktning sent i projektet men koden kan ändå tyckas vara anpassad till deras programeringssätt vilket talar för implementerbarheten.

Den allra största fördelen med den nya standarden är att koden är lättläst och enhetlig oavsett vem som har programmerat vad. En lättläst kod är ett huvudmål i alla programmeringsstandarder vilket den nya standarden har lyckats med. Ett önskemål i Volvos standard är att scrollande i sidled skall undvikas för läslighetens skull och det nya sättet att programmera med operationsblock gör att detta efterlevs utan extra steg att ta för programmeraren. På Volvo där de hyr in och har många olika programmerare är det en stor fördel att koden är lätt att sätta sig in i. När nya delar, ny hårdvara eller programändringar ska införas finns många av modulerna sedan innan och kan kombineras till önskad sekvens. De moduler som inte finns förs lätt in i form av nya Abilities och Operations utan att påverka befintligt utförande.

Säkerheten i produktionscellen i PSL är orörd i detta projekt och har införts på precis samma sätt som det var utformat i den gamla PLC:n. Kommande projekt kan inbegripa att införa den nya standarden även i de delarna och att lägga till eventuella nya delar i standarden som även behandlar säkerheten. I PSL är säkerheten övergripande och ingen extra kod behövs inom sekvensstyrningen för att den ska fungera som den ska. Detta ökar säkerheten i och med att risken för att kod som rör säkerheten glöms bort, tas bort eller ändras minskar och den är oberoende av övrig kod. Även om ändringar inte har gjorts i säkerheten ses fördelarna med det övergripande systemet som något som definitivt ska användas och implementeras i den nya standarden. I koden från Volvo sågs nackdelarna med mycket extra kod rörande säkerhet i alla nivåer av sekvensstyrningen som gör koden mindre lättläst och därmed mer svårförståelig.

Tack vare de hierarkimässiga likheterna mellan Volvos kod och det implementerade operationskonceptet vad gäller vissa delar av styrning och exekveringsvillkor förenklas ett eventuellt implementerande av den nya standarden i Volvos PLC-program. Den nya standarden är flexibel vad gäller hur hög grad av sekvensstyrning som ska implementeras även om ett program med förutbestämd sekvens i många avseenden motverkar fördelarna med operationskonceptet. Standarden är flexibel nog för att den ska kunna införas i Volvo utan att de behöver ändra sitt programmeringssätt i många andra avseenden än användandet av de utformade operationsblocken. Sedan kan Volvo själva välja vilka delar

de vill behålla från sitt gamla tillvägagångssätt och vilka de vill anamma från den nya standarden och metoden. Detta kan ses som en fördel eftersom det förenklar införandet av standarden i Volvos kod.

8.3 Styrning

Den programuppdelning och hierarki som skapats med Operations och Abilities utgör en stor fördel när en lättöverskådlig och tydlig standard skapas. Uppdelningen gör koden flexibel för förändringar då varje aktivitet beskrivs med en Operation och Ability som är fristående men lätt att kombinera med alla andra Operations och Abilities. Det ända som behöver förändras eller läggas till vid ändringar i sekvensen är de övergripande sekvensstyrande operationsblocken. Detta kan spara mycket tid i utvecklingsprocesser då programmet inte behöver skrivas om från grunden, istället används de befintliga modulerna, fast i en annan sekvens. Denna uppdelning gör det även lätt att utöka med nya moduler exempelvis när ny hårdvara införs.

Under projektets gång diskuterades införandet av parallell ”produktion” i form av byggande med någon form av klossar. I framtida arbeten förenklas införandet av denna process av det nya tillvägagångsättet vid programmeringen och kombinationen av de båda sekvensstyrningarna kan förenklas av den aktivitetsinriktade styrningen.

Ingen optimering av produktionssystemet har utförts vilket dock heller inte varit målet med projektet. Det har istället utvecklats en bra grund att bygga vidare på och som med enkelhet ska gå att utveckla. Ett problem som kan uppstå vid optimering är att storleken och omfattningen på de utvecklade operationerna gör att det blir svårt att utföra aktiviteter parallellt och på så sätt spara tid vid produktionsprocesserna. Detta beror på implementeringen av standarden och inte själva standarden i sig. Desto mindre operationer desto lättare är det att optimera för att effektivisera styrningen, dock önskas det att ändå erhålla en tillräckligt stor omfattning på grund av att antalet Operations då minskar och styrningen blir mer lättöverskådlig och flexibel vid förändring. Då detta projekt fokuserat på att skapa en tydlig standard och inte optimering har därför Operations tenderat att omfatta mer och på så sätt skapa en lättöverskådlig kod.

Ett exempel på något som kan optimeras är att skriva programmet så att roboten byter verktyg redan innan nästa komponent är redo att plockas. Detta är inte fallet med det program som skapats då roboten byter verktyg när den får uppdraget att plocka den aktuella komponenten. Detta problem har delvis framkommit genom det ovan beskrivna problemet, att den Operation som styr detta omfattar en större sekvens. Hade denna Operation delats upp och gjorts mindre hade det parallellt kunnat utföras ett verktygsbyte samtidigt som komponenten är på väg in i produktionssystemet.

Då detta projekt inte haft som syfte att optimera produktionssystemet finns det mycket att utveckla i det avseendet. Dels kan robotrörelserna optimeras för att utföra färre och kortare rörelser och på så sätt effektivisera energiåtgången eller tiden för att utföra en operation. Något annat som kan optimeras är PLC-styrningen. Som styrningen är uppbyggd nu krävs det att modellbilen monteras i en specifik ordning. En möjlighet till att utveckla produktionsprocessen hade varit att göra styrningen mer flexibel genom att utnyttja RFID-läsarna på flexlinkbanan för att läsa av vilken komponent som är på väg in i systemet. På så sätt låta roboten antingen montera komponenten i fixturen eller lägga

den på ett avlastningsbord till det är den komponentens tur att bli monterad. Ytterligare utvecklingsmöjlighet hade varit att försöka styra allt mer parallellt. Att optimera så att mer saker händer samtidigt och på så sätt få ner monteringstiden.

En fördel med det nya programmeringssättet är att sekvensstyrningen i Operations gör det möjligt att välja hur låst styrningen ska vara. Det är möjligt att göra en helt fast styrning som enbart utför aktiviteter i exakt den ordning som specificerats alternativt kan en helt aktivitetsbaserad styrning som utför varje aktivitet så snart som den kan utföras enligt de villkor som har satts som *PreConditions* erhållas.

Tack vare kommunikationen via in- och utgångar till operationsblocken Ability och Operation behövs inte den mindre genomskinliga kommunikationsmetoden med variabler av typen word som Volvo använder sig av i dagsläget. Ingenting hindrar dock från att fortsätta använda sig av samma metod efter införande av det nya programmeringssättet och den nya standarden.

En möjlig utvecklingspunkt hade varit att utvecklat och implementerat effektiva återstarter. Som programmet är uppbyggt kräver det en manuell återstart av alla variabler om styrningen skulle fastna i ett läge under exekvering för att sedan behöva börja om från början. Vid en utveckling av återstarter hade det kunnat vara möjligt för operatören och välja vilken återstartspunkt som anses mest lämplig och på så sätt gå tillbaka till den punkten genom att återställa rätt variabler och sedan fortsätta processen från den punkten. Detta för att minska den tid som upptas utav stopp i produktionen. Detta har verifierats av Patrik Bergagård genom hans forskning om effektiva återstarter i produktionssystem [3].

Robotstyrningen i produktionscellen visade sig fungera väl under verifieringen dock implementerades inte styrningen för robot R4 på grund av tidsbrist. Utvärderingen av robotstyrningen avsåg därför två robotar istället för de tre som var tänkt initialt. De legomoduler som var tänkt att monteras i modellbilen blir inte monterade under verifieringen av konceptet i PSL då robot R4 skulle plocka legomodulerna vid avlastningsbordet för att sedan montera de i modellbilen. Dock implementerades robot R2:s operationer med att lyfta in legomodulerna i cellen, på så sätt blev det utvecklade verktygsbytet implementerat. Denna avvikelse har dock ingen större påverkan på det slutgiltiga resultatet, då programmet för styrning i PSL endast var till för att verifiera det utvecklade konceptet. Projektet avsåg även att utveckla och implementera ett zonbokningssystem för robotarna men på grund utav tidsbrist och det sänkta antalet verksamma robotar blev inte detta utfört.

En möjlighet till vidareutveckling inom robotstyrningen hade varit att skapa en virtuell modell över produktionssystemet för att sedan importera den i *RobotStudio* för att utföra offline-programmering och utnyttja simulation. I detta projekt har programstrukturen skrivits offline men alla rörelser har programmerats online, vilket är en tidskrävande process. Med offline-programmering och simulering hade eventuellt utvecklingstiden kunnat minskat och mer tid kunde lagts på andra saker, vilket gör att projektgruppen rekommenderar detta till framtida projekt.

För att förbättra det utvecklade verktygsbytet kan en sensor installeras. Denna sensor skulle kunna bekräfta att verktyget verkligen är släppt i sin dockningsstation innan den försöker montera det andra verktyget. Detta hade ökat säkerheten och tillförlitligheten i produktionssystemet då det finns en risk att verktyget fastnar i robotens mekaniska låsning och på så sätt riskerar att förstöra både roboten i sig men även kringutrustning.

8.4 Automatisk kodgenerering

Den automatiska kodgenereringen användes och utvärderades under utvecklingen där empiriska tester visade sig att utvecklingstiden för en Robot Ability minskade med 65 % då den gick från 180,75 sekunder till 63 sekunder, se Appendix F. Tidsdifferensen mellan de båda utvecklingsmiljöerna utgörs inte av statisk karaktär, utan har snarare ett dynamiskt beroende av antalet Networks som genereras. Eftersom utvecklingsmiljön och programmet som kommunicerar med TIA-Portalen har separerats är utvecklingstiden beroende utav antalet funktionsblock som genereras, då tiden för importen ej är starkt beroende av antalet funktionsblock. En lösning för att uppnå jämnare utvecklingstider är att integrera utvecklingsmiljön med kommunikationsprogrammet. Ett annat positivt utfall, som en integrering mellan de två mjukvarorna skulle haft, är att mellanlagringarna av XML-dokument ej hade varit nödvändiga då detta hade kunnat skötas internt i programmet.

Eftersom Siemens ej tillhandahåller någon information över deras struktur i XML-dokumenten har en stor mängd tid fått läggas på att förstå sig på strukturen genom att exportera färdiga funktionsblock. Det har varit en relativt bra arbetsmetod eftersom svar snabbt har kunnat erhållas på hur strukturen skall se ut genom att konstruera och exportera ett block av typen som skall återskapas. Dock så har den här metodiken ibland orsakat kostsamma misstolkningar som lett till att stora delar av koden har fått skrivas om. Ett exempel på ett tillfälle när en sådan misstolkning har uppstått är då en variabel som förekommer mer än en gång i ett Network ska deklareras i *Parts*. Här hade först programmet utformats efter att en variabel endast skulle behöva deklareras en gång inom ett Network vilket inte är i linje med Siemens struktur där den istället ska deklareras en gång per instans.

Den automatiska kodgenereringen har under projektets gång visat sig ha god potential, inte minst på grund av den minskade utvecklingstiden programmet hade för av Robot Abilities. Det behövdes inte spenderas någon tid på felsökning av de 20 Robot Abilities som genererades under utvärderingen. Dock så krävs det en större samplingsmängd för att kunna dra några slutsatser av utvecklingsmiljöns utfall i felsökningen av projekten. Momentet med automatisk kodgenerering växte fram först ett par veckor in i projektet. Som en följd av det är det en del funktioner som ej kunnat implementerats på grund av tidsbrist. Ett exempel på en funktion som inte har blivit implementerad är import av databaser till den externa utvecklingsmiljön då den i nuläget endast stödjer import utav tag-listor. En annan del som behöver vidareutvecklas är programmets aspektorienterade funktionaliteter. Användaren behöver tilldelas mer kontroll vid valet av *Join Points* samt att implementera mer *Advices* i programmet. Ett område där det finns stor potential att införa automatisk kodgenerering av är för de kodsegment som berör HMI och säkerheten. Dessa två områden har inte berörts inom projektet då säkerheten i *PSL* hanteras externt samt att det inte har funnits tillräckligt med tid för att implementera HMI:er. I den kod som tillhandahållits av Volvo Cars utgörs dessa två områden av högst omfattande och repetitiva segment, se figur B.2 & B.3 i Appendix B.

Då det repetitiva kodsegementet som identifierats och utvärderats för automatisk kodgenerering endast utgör en liten del av den totala koden i projektet, så har den totala utvecklingstiden inte påverkats likt den för en Robot Ability. Dock så kan införande av *Advices* för fler repetitiva som exempelvis HMI-hantering och säkerheten minska den totala utvecklingstiden mer påtagbart. För att en programvara för automatisk kodgenerering ska kunna få ett industriellt genomslag krävs det förmodligen att språket i utvecklingsmiljön

påminner mer om ett vanligt konventionellt PLC-programmeringsspråk. Det krävs förmodligen också funktioner för att skapa *Advices* och sedan tillsätta dessa i olika templates, istället för att behöva skapa dessa i källkoden.

9

Slutsats

Produktionssystemen blir alltmer automatiserade och kräver därför mer fokus på hur de ska styras. PLC, *Programmable Logic Controller*, har i många år varit det ledande styrsystemet inom industrin men få ändringar i hur en PLC styrs har genomförts. Därför har detta blivit en allt mer prioriterad forskningsfråga. Många företag har utvecklade standarder som ska underlätta för att skapa PLC-logik och det förekommer mycket repetitiva segment på grund utav dess krav om tydlighet och överskådlighet. Att skriva kodsegment med repetitiv kod är en mycket tidskrävande process och därför finns där stor potential i att automatiskt generera de delarna. Därför har detta projekt haft som syfte att utveckla ett koncept för hur ett företag från tillverkningsindustrin kan implementera och dra nytta av den forskning som bedrivits inom automation. Projektet syftar också på att utveckla en miljö som kan autogenerera repetitiva kodsegment för att reducera utvecklingstiderna i ett projekt.

Konceptet som utvecklats under projektets gång innefattar primärt framtagning av en PLC-standard som kopplar samman Volvo Car Corporations nuvarande PLC-standard med forskning och utveckling som bedrivs på Chalmers Tekniska Högskola inom automation. Standarden ska möjliggöra en modulär programarkitektur genom att programmets olika Abilities och Operations delas upp i olika segment av programmet. Konceptet innefattar även automatisk kodgenerering utav frekvent förekommande aspekter inom de olika programmodulerna.

Konceptet har implementerats för utvärdering i PSL, *Production System Laboratory*, för montering utav en modellbil. Programmets olika Abilities har delats in i bibliotek för att öka flexibiliteten genom att de används från den övergripande styrningen. Standarden ger även en kontroll över exekveringen utav programmet på grund av det operationskoncept som har implementerats. Det har även skapats en utvecklingsmiljö för extern programmering av PLC-logik som möjliggör automatisk kodgenerering av repetitiva kodsegment. Programmeringsmiljön har utvärderats genom att generera alla förmågor för en produktionsrobot i projektet. Utvärderingen visade att utvecklingstiden minskade med 65 % för dessa Abilities då de genererades från den nya programmeringsmiljön. Det framtagna resultatet påvisar möjligheten för implementering utav forskningen som bedrivits på Chalmers Tekniska Högskola i industriella verksamheter samt möjligheten att minska utvecklingstiderna genom användning av automatisk kodgenerering.

För att det framtagna konceptet ska kunna utnyttjas i industriella verksamheter behöver det vidareutvecklas för att implementeras i mer storskaliga projekt. Operationskonceptet möjliggör en beskrivning utav programmets tillstånd genom diskreta eventsystem, vilket skapar möjligheter till att inkludera ett system för effektiva återstarter i konceptet för att minska tiden som upptas av stopp i produktionen. Vid en implementering av återstarter

kan istället operatören välja vilken återstartspunkt som anses lämplig och på så sätt gå tillbaka till det tillståndet istället för att börja om processen.

Programmeringsmiljön för automatisk kodgenerering behöver vidareutvecklas för att möjliggöra användning i industrin. Dock påvisar verifieringen potentialen den automatiska kodgenereringen har på utvecklingstiderna av PLC-logik i industrin. Vid vidareutveckling av det framtagna konceptet kan det med fördel implementeras i industriella verksamheter för att minska utvecklingstider. Då fås dessutom en standard som är mer mottaglig för kommande forskning så att nytta kan dras av den forskning som bedrivs kontinuerligt inom automation.

Litteraturförteckning

[1] A. Zoitl and R. Lewis, "Modelling control systems using iec 61499 (chapter 1)," vol. 2, 2014.

[2] A. U. M.G. Mehrabi and Y. Koren, "Reconfigurable manufacturing systems: Key to future manufacturing," *Journal of intelligent manufacturing*, vol. 11, pp. 403–419, 2000.

[3] P. Bergagård, *On restart of automated manufacturing systems*. PhD thesis, Chalmers University of Technology, 2015.

[4] K. Hitomi, "Automation - its concept and a short history," *Technovation*, vol. 14, no. 2, pp. 121–128, 1994.

[5] L. Hågeryd, S. Björklund, and M. Lenner, *Modern Produktionsteknik Del 2*. Liber AB, 2005.

[6] "Privat konversation med Kristofer Bengtsson."

[7] R. Lewis and I. of Electrical Engineers, *Programming Industrial Control Systems Using IEC 1131-3*. IEE control engineering series, Institution of Electrical Engineers, 1998.

[8] Volvo Car Corporation, *STANDARD VCS8015,39*, 2012.

[9] O. Ljungkrantz and K. Åkesson, "A study of industrial logic control programming using library components," in *Automation Science and Engineering, 2007. CASE 2007. IEEE International Conference on*, pp. 117–122, IEEE, 2007.

[10] M. Lucas and D. Tilbury, "A study of current logic design practices in the automotive manufacturing industry," *International Journal of Human-Computer Studies*, vol. 59, no. 5, pp. 725 – 753, 2003.

[11] T. Börjesson, C. Eliasson Lilja, M. Grönbäck, S. Kjerstadius, A. Larsson, and O. Noresson, "Tidseffektiv tillverkningscell med operationer," 2013. Signaler och System.

[12] ABB Robotics Products AB, *RAPID Reference Manual*.

[13] M. Fabian, "Industrial automation," 2006.

[14] K. Bengtsson, *Operation Specification for Sequence Planning and Automation Design*. PhD thesis, Chalmers University of Technology, 2009.

[15] Mitsubishi Electric, *Programmable Logic Controllers Beginners Manual*, 2006.

[16] K. Bengtsson, *Flexible design of operation behavior using modeling and visualization.* PhD thesis, Chalmers University of Technology, 2012.

[17] P. Bergagård, S.Parsaean, K. Bengtsson, and M. Fabian, "Implementing restart in a manufacturing system using restart states," in *Calculating restart states for systems modeled by operations using supervisory control theory. P. Bergagård*, 2014.

[18] J. Kleinberg and É. Tardos, *Algorithm design.* Pearson Education India, 2006.

[19] P. Falkman, E. Helander, and M. Andersson, "Automatic generation: A way of ensuring plc and hmi standards," in *IEEE International Conference on Emerging Technologies and Factory Automation, ETFA*, 2011.

[20] L. Yun-Hui, "Optimal fixture layout design for 3-d workpieces," 2004.

[21] S. Hiroshi, "Automatic setup planning and fixture design for machining," 1990.

[22] Siemens, *SCE Training Curriculum for Integrated Automation Solutions Totally Integrated Automation (TIA)*, 2013.

[23] "Två studiebesök på Volvo Car Corporation, 2015-03-18 och 2015-02-25."

[24] "Utdrag av PLC-kod från Volvo Car Corporation."

[25] Z. Pan, J. Polden, N. Larkin, S. V. Duin, and J. Norrish, "Recent progress on programming methods for industrial robots," 2011.

[26] D. Nord, N. Borg, N. Skog Lidander, and W. Herrera Dowland, "Utveckling och implementering av metoder för enkel styrning och återstart av automatiserade produktionssystem," 2014. Signaler och System.

[27] J. Thieme and H. . Hanisch, "Model-based generation of modular plc code using iec61131 function blocks," in *IEEE International Symposium on Industrial Electronics*, vol. 1, pp. 199–204, 2002.

[28] M. Steinegger and A. Zoitl, "Automated code generation for programmable logic controllers based on knowledge acquisition from engineering artifacts: Concept and case study," in *Emerging Technologies & Factory Automation (ETFA), 2012 IEEE 17th Conference on*, pp. 1–8, IEEE, 2012.

[29] K. Sacha, *Automatic code generation for PLC controllers*, vol. 3688 LNCS of *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics).* 2005.

[30] C. . Cheng, C. . Huang, H. Ruess, and S. Stattelmann, *G4LTL-ST: Automatic generation of PLC programs*, vol. 8559 LNCS of *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics).* 2014.

[31] PLCopen. http://www.plcopen.org/pages/tc2_motion_control/, 2008. accessed 23-March-2015.

[32] K. Bengtsson, B. Lennartson, O. Ljungkrantz, and C. Yuan, "Developing control logic using aspect-oriented programming and sequence planning," *Control Engineering Practice*, vol. 21, no. 1, pp. 12–22, 2013.

[33] D. Nolan and D. T. Lang, *XML and Web Technologies for Data Sciences with R.* Springer, 2014.

[34] C. Pozrikidis, *XML in scientific computing.* CRC Press, 2012.

[35] J. Skansholm, *Skarp programmering med C#.* Studentlitteratur, 2008.

[36] P. G. Anders Hejlsberg, Scott Wiltamuth, *The C# programming language.* Pearson Education, second ed., 2006.

[37] Z. Y. Dong, *The study on the implementation of AOP in.NET platform*, vol. 336-338 of *Applied Mechanics and Materials.* 2013.

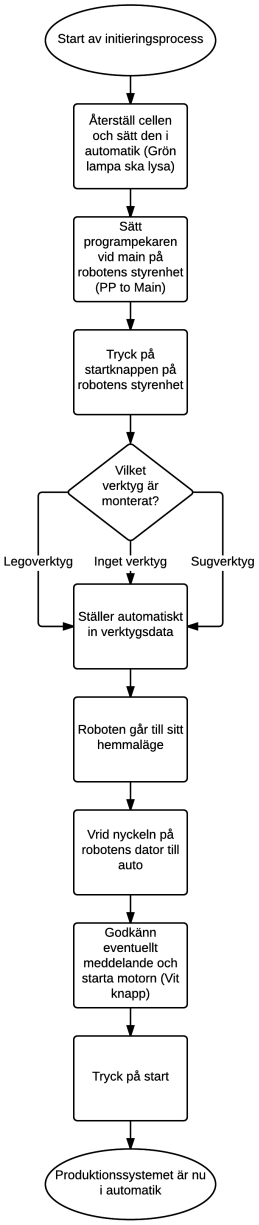
[38] M. D. Groves, *AOP in .NET; practical aspect-oriented programming.* Manning Publications, 2013.

[39] R. Filman, T. Elrad, S. Clarke, and M. Akşit, *Aspect-oriented Software Development.* Addison-Wesley Professional, first ed., 2004.

A

Start av produktionscellen

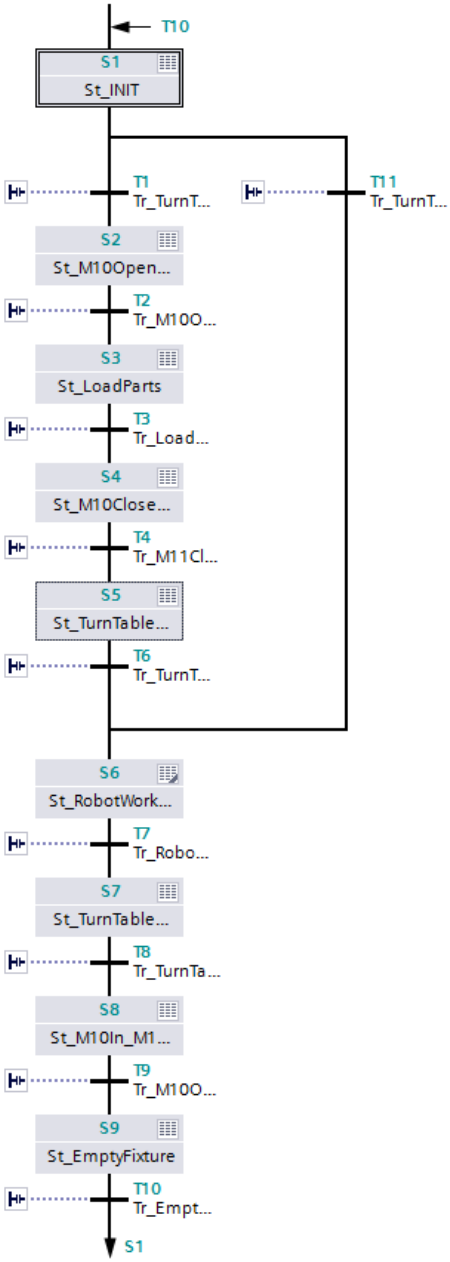
Förklaringen för hur produktionssystemet startas manuellt och sedan sätts i automatik visas i figur A.1.



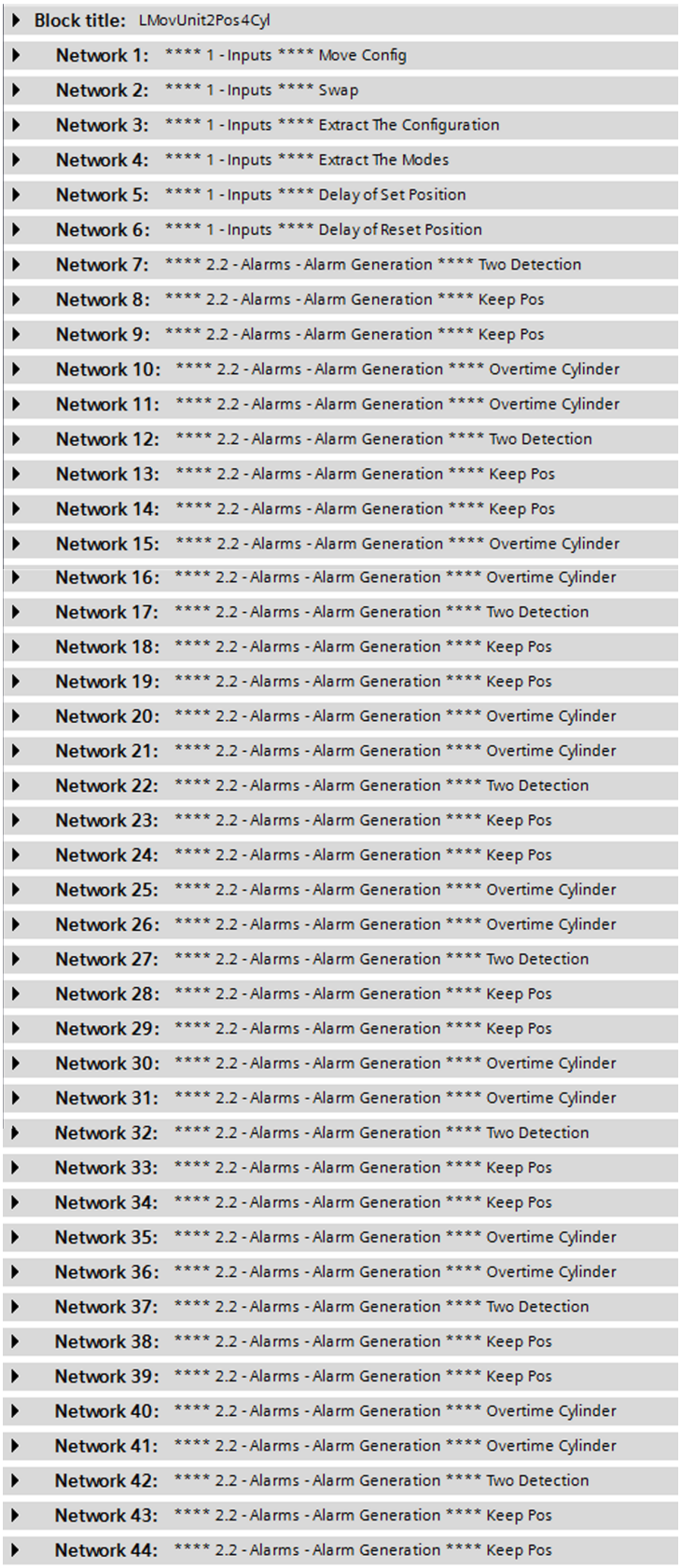
Figur A.1: *Beskrivning av hur produktionssystemet startas och sätts i automatik*

B

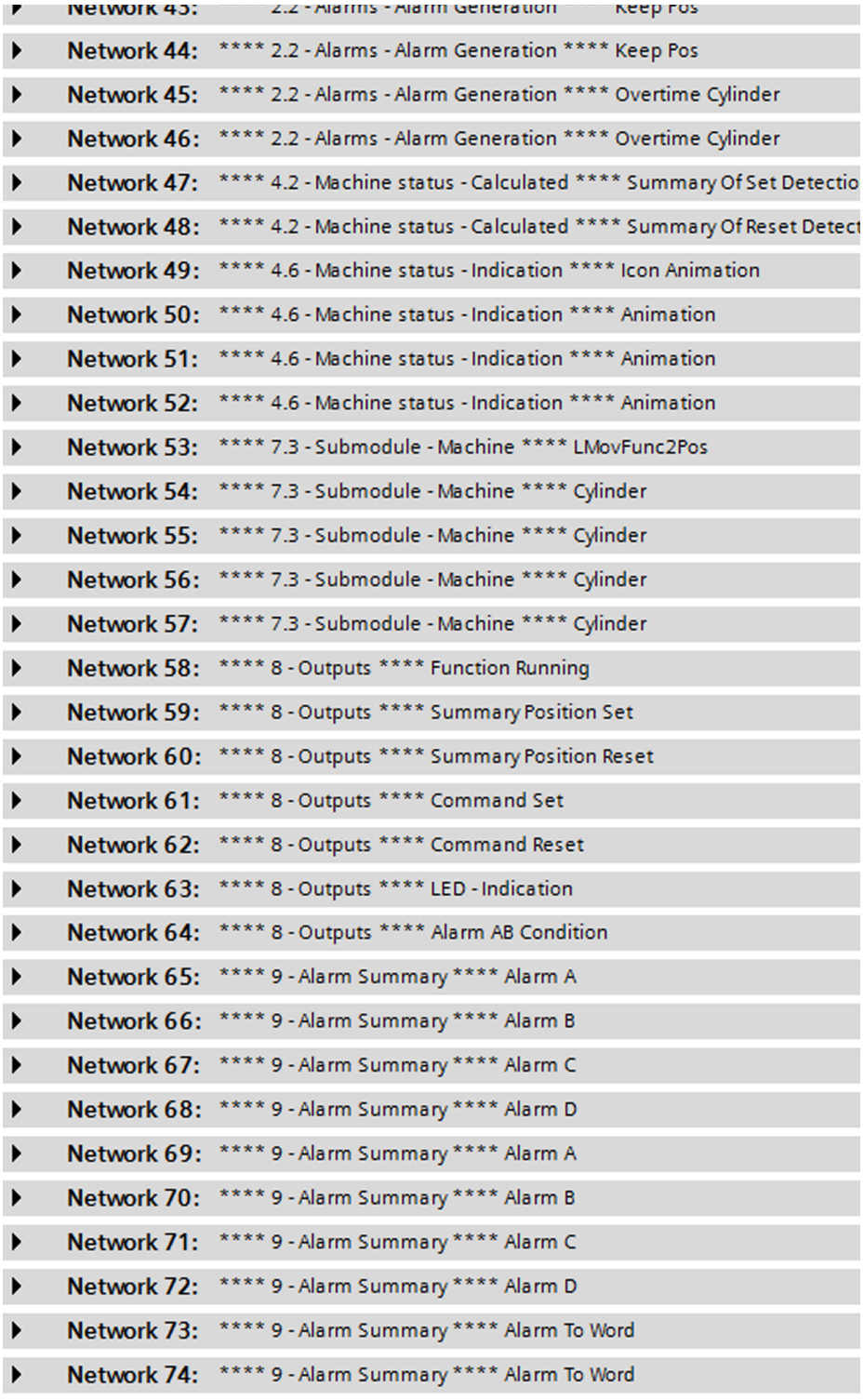
Exempelkod tagen ur Volvos kod



Figur B.1: *Programmet FixtureSequence skriven i SFC*



Figur B.2: *Operationsblocket LMovUnit2Pos4Cyl i kollapsad vy för överblick över alla Networks*



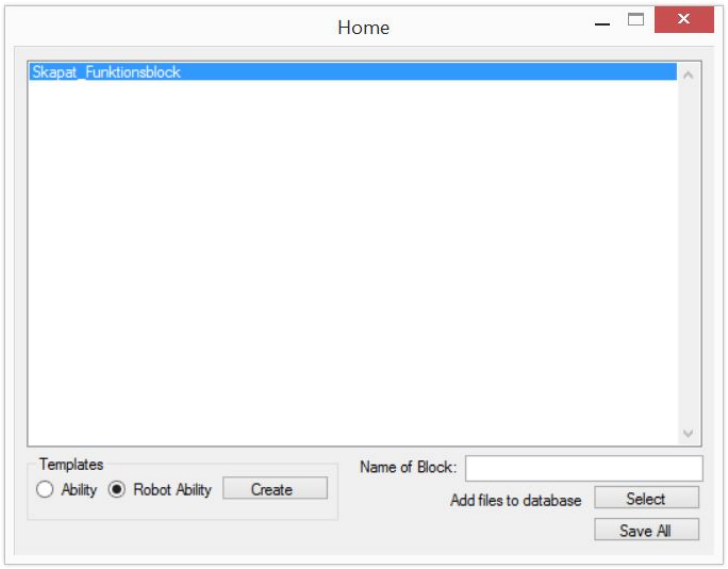
Figur B.3: *Forts. Operationsblocket LMovUnit2Pos4Cyl i kollapsad vy för överblick över alla Networks*

C

Manual för utvecklingsmiljön

Från programmets startfönster finns det en mängd olika funktionaliteter att välja mellan. Det som bör göras först för att underlätta arbetet i utvecklingsmiljön är att lägga till filer i databasen. Det görs genom att trycka på knappen ”Select” brevid labeln ”add files to database” som kan ses i figur C.1. Då ”Select” blir intryckt öppnas ett utforskarfönster filtrerat för XML-filer. Användaren får sedan lokalisera de filer som innehåller PLC-tagsen och markera dessa.

Programmet läser sedan av filerna och lagrar de innehållande variablerna i en statisk klass med global synlighet. Variablerna sorteras sedan efter datatyper för att underlätta användande av dessa i utvecklingsmiljön. När filerna har lagts till i databasen är det möjligt



Figur C.1: *Export & Importprogrammets hemskärm*

att skapa funktionsblock. Först anger användaren namnet på det funktionsblock som ska skapas samt sedan ange vilket template som ska användas. Som det synes i figur C.1 finns det två stycken templates att välja mellan ett för en Robot Ability samt ett för en vanlig Ability. Det template som är avsett för en Robot Ability kommer med ett antal färdiga objekt av *Network* som sedan autogenererar de delar rör robotkommunikationen. Efter att ovanstående kriterier är uppfyllda nås utvecklingsmiljön genom en knapptryckning på ”Create”.

Den nya utvecklingsmiljöns layout består primärt utav tre fält, se figur C.2. I det vänstra fältet deklareras alla övergångar mellan noderna, i det nedre av de högra fälten får

användaren välja vilket Network som ska vara aktuellt för redigering och i det övre av de högra fälten visas de noder som är tillgängliga som startpunkter för nästa övergång.

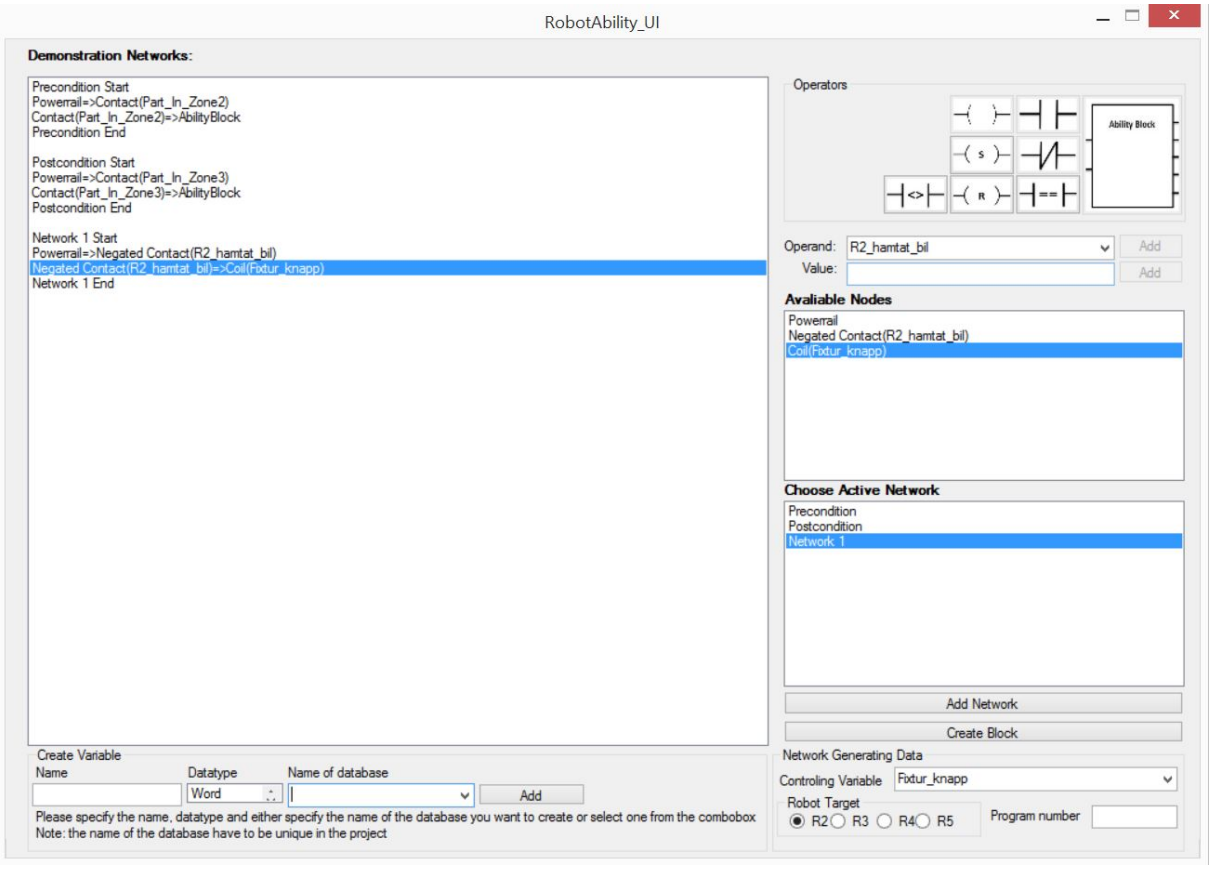
Vid start finns det två stycken Networks tillgängliga för redigering. De två som är tillgängliga är ingångarna *PreCondition* och *PostCondition* för Abilityn. De Network som blivit genererade listas i det vänstra fältet tillsammans med tillhörande övergångar mellan de olika noderna.

När en ny koppling genomförs får användaren välja en tillgänglig nod från fältet som håller dessa samlade. Målnoden nås genom att antingen välja från det samlade fältet eller genom att skapa en ny nod genom att en operator och operand specifieras av användaren. Vid konstruktion av en ny nod väljs först en av de tillgängliga operatorerna i det övre högre hörnet. Efter att operator har blivit vald matchas sedan operandfältet mot de variabler i databasen som är av godkänd datatyp för operatorn. Om operatortypen är en matematisk jämförelse behövs även en konstant anges för att fullborda det matematiska uttrycket. Då övergången är färdig listas den inom det Network som är aktuellt i den vänstra listan. Den eventuellt nya noden fylls även på i listan över tillgängliga noder.

Det går även att skapa och använda nya variabler och databaser i utvecklingsmiljön. Detta sker i den nedre delen av programfönstret och den nya databasen kommer generera en separat XML-fil som även den behöver importeras till TIA Portalen för den ska finnas tillgänglig i det befintliga projektet.

För utveckling inom det template som berör Robot Abilities behöver även ytterligare information anges innan det aktuella funktionsblocket kan genereras. Först behövs den aktuella roboten markeras för att sedan ange robotoperationens programnummer samt att välja en kontrollvariabel för att möjliggöra övergripande styrning i TIA-Portalen.

För att spara de funktionsblock som skapats trycks knappen ”Save All” ned som öppnar ett utforskarfönster där användaren får specificera en målmapp där de genererade XML-filerna lagras.



Figur C.2: *Nya Utvecklingsmiljön*

D

Manual för kommunikationsprogrammet

För att kommunikationen med TIA-Portalen ska fungera krävs det att följande installationer och konfigurationer ska vara uppfyllda:

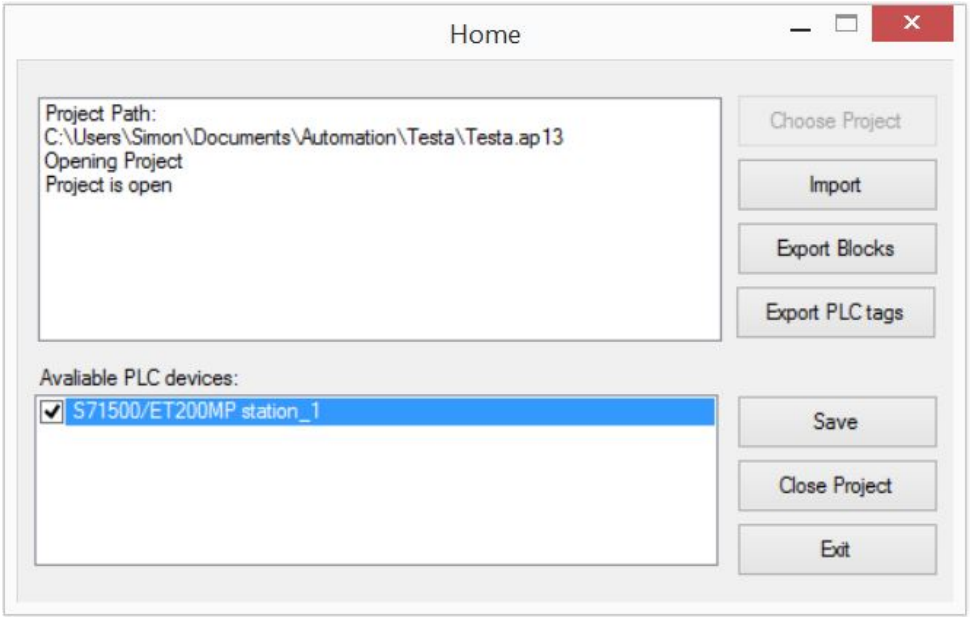
- TIA Portal Openness ska vara installerat på datorn
- Läg till den akutella användaren på operativsystemet i Siemens TIA Opennessänvändargrupp
- En auktoriseringsfil från Siemens, *SiemensTIAOpennessCustomerID.xml*, ska finnas i mappen för den exekverbara filen för kommunikationsprogrammet
- En användingsfil från Siemens, *SiemensTIAOpennessUsage.xml*, ska placeras på samma adress som installationsfilen för Siemens TIA Portal V13, om inget annat angavs vid installationen: `C:\ Program Files (x86)\ Siemens\ Automation\ Portal V13\ PublicAPI\ V13SP1\ SiemensTIAOpennessUsage.xml`

D.1 Programbeskrivning

Från programmets startfönster ombedes användaren att först specificera vilket projekt som kommunikationen ska riktas till. Detta sker genom ett utforskarfönster som är filtrerat för ap.13- filer, där användaren får bläddra igenom programbiblioteket för att finna det önskade projektet. Programmet itererar sedan genom projektets olika PLC-enheter och placerar de i en lista där användaren får välja vilken PLC-enhet kommunikationen ska rikta sig till. Efter det kan användaren välja fritt bland de funktionaliteter som finns tillgängliga i startfönstret (se figur D.1). Programmets olika funktioner nämns mer i detalj nedan.

D.1.1 Import

Import till TIA-Portalen av de funktionsblock och databaser som skapats i den nya utvecklingsmiljön sker via XML-filer. Alla funktioner som rör import av funktionsblock samt databaser sker via importfönstret som illustreras i figur D.2. Inne i importfönstret väljer användaren de filer som ska importeras genom att trycka på knappen ”Select Files”. Då öppnas ett utforskarfönster som är filtrerat för XML-filer för att uppnå en högre



Figur D.1: *Startfönster för programmet*

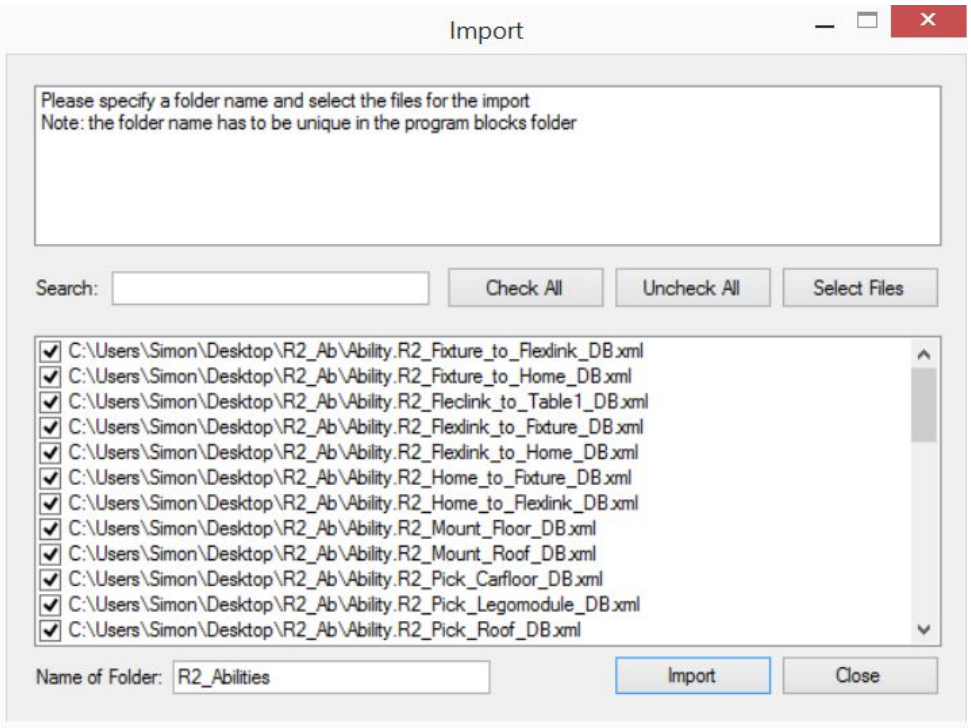
användarvänlighet. De valda filerna läggs sedan i en lista där användaren kan markera de filer som ska importeras genom att trycka i rutan till vänster om filnamnet. Alla filer blir automatiskt markerade vid import, men programmet har även utrustats med en knapp som markerar alla filer samt en som avmarkerar alla. Eventuella felmeddelanden som uppstår under importen loggas och skrivs sedan ut i det övre listfönstret för att delge användaren information för att åtgärda felet.

Under utvecklingen har det eftersträvats att upprätthålla en god användarvänlighet. Detta har lett till att programmet har utrustats med en sökfunktion. Filen som matchar sökningen bäst i listan markeras dynamiskt när användaren skriver in det önskade filnamnet i textrutan. Det här leder i sin tur till att det filnamn som motsvarar sökningen bäst markeras i listan dynamiskt. För att upprätthålla en god struktur i projektet i TIA Portalen skapar programmet mappar dit de nya filerna importeras. Det önskade mappnamnet specificeras i det nedre vänstra hörnet av programfönstret. Viktigt att notera är att det nya mappnamnet måste vara unikt inom projektet då Siemens ej tillhandahåller någon funktionalitet för att hantera dessa typer av konflikter. För att slutföra importen trycks knappen ”Import” in.

D.1.2 Export

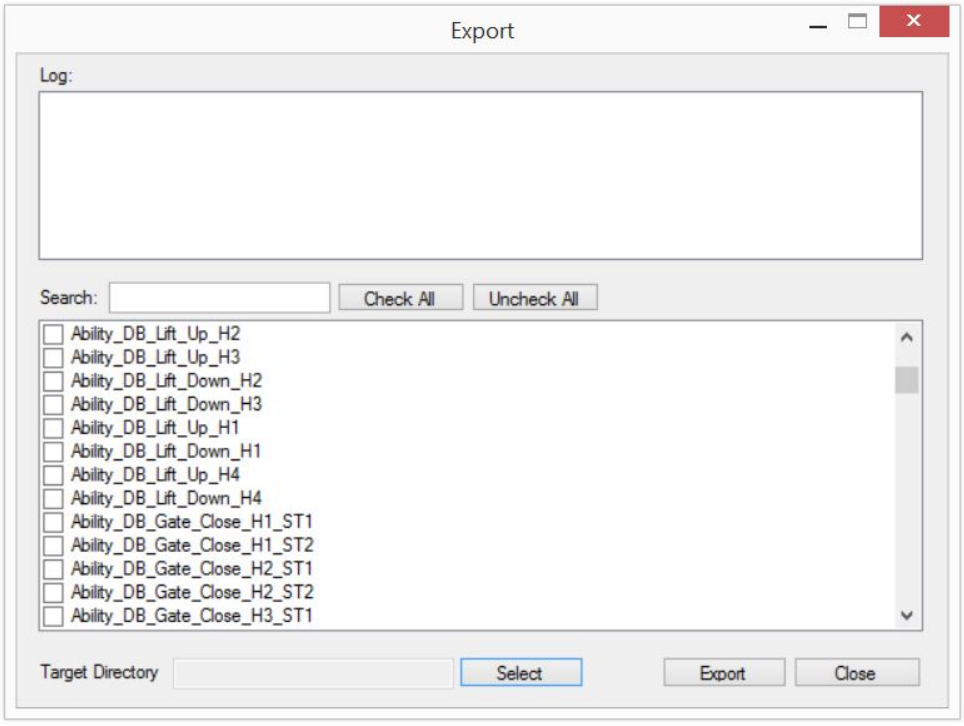
Export från TIA-Portalen sker även den via XML-filer vars struktur går att se i Appendix H. Programfönstret för exportfunktionen går att se i figur D.3. Då exportfunktionen anropas från startmenyn itererar programmet genom alla programmappar för den valda PLC:n och projektet. Programmet placerar sedan alla funktionsblock och databaser i listan som erhåller samma funktionalitet som den för importfunktionen.

Eventuella felmeddelanden som uppstår loggas även här och skrivs ut i det övre listfönstret för att delge användaren information om vad som gått fel. Innan filerna exporteras från TIA Portalen behöver användaren specificera en målmapp till de exporterade filerna. Detta



Figur D.2: *Överblick av importfunktionen för programmet*

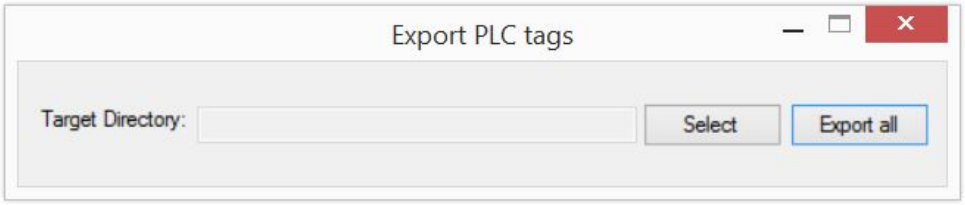
sker i det nedre vänstra hörnet inom programfönstret via knappen ”Select”.



Figur D.3: *Överblick av exportfunktionen för programmet*

Export av PLC-tags sker från programfönstret i figur D.4. Programfönstret har en något avskalad layout. Med endast två knappar, en för val av målmapp för exporten samt en knapp som exporterar alla tags.

Vid aktivering av exporten itereras alla mappar inom projektet för att finna alla tillgängliga taglistor och exporterar sedan en separat XML-fil per lista som innehåller all information som behövs för att variablerna ska kunna användas i den nya utvecklingsmiljön.



Figur D.4: *Överblick av exportfunktionen av PLC tags för programmet*

E

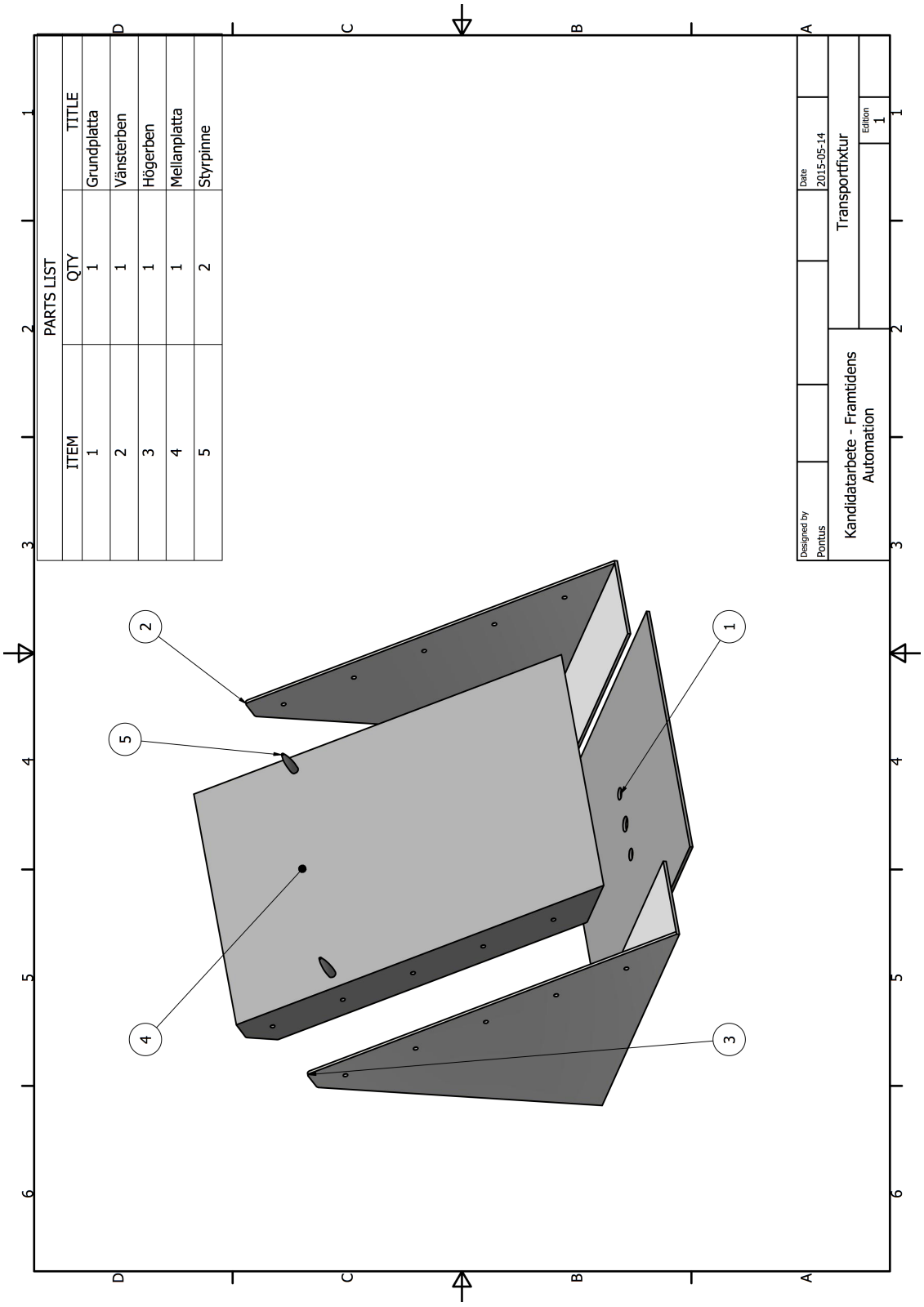
Ritningar

Här visas de ritningar på transportfixturen samt profilen som skapats vid utveckling. Först visas en översiktsritning över transportfixturen för att sedan visa ritningar på dess komponenter. Efter det visas ritningar på profilen, först en ritning på hur den såg ut efter konstruktion och därefter en ritning på hur den såg ut när den tillverkades. De olika ritningarna är listade i tabell E.1.

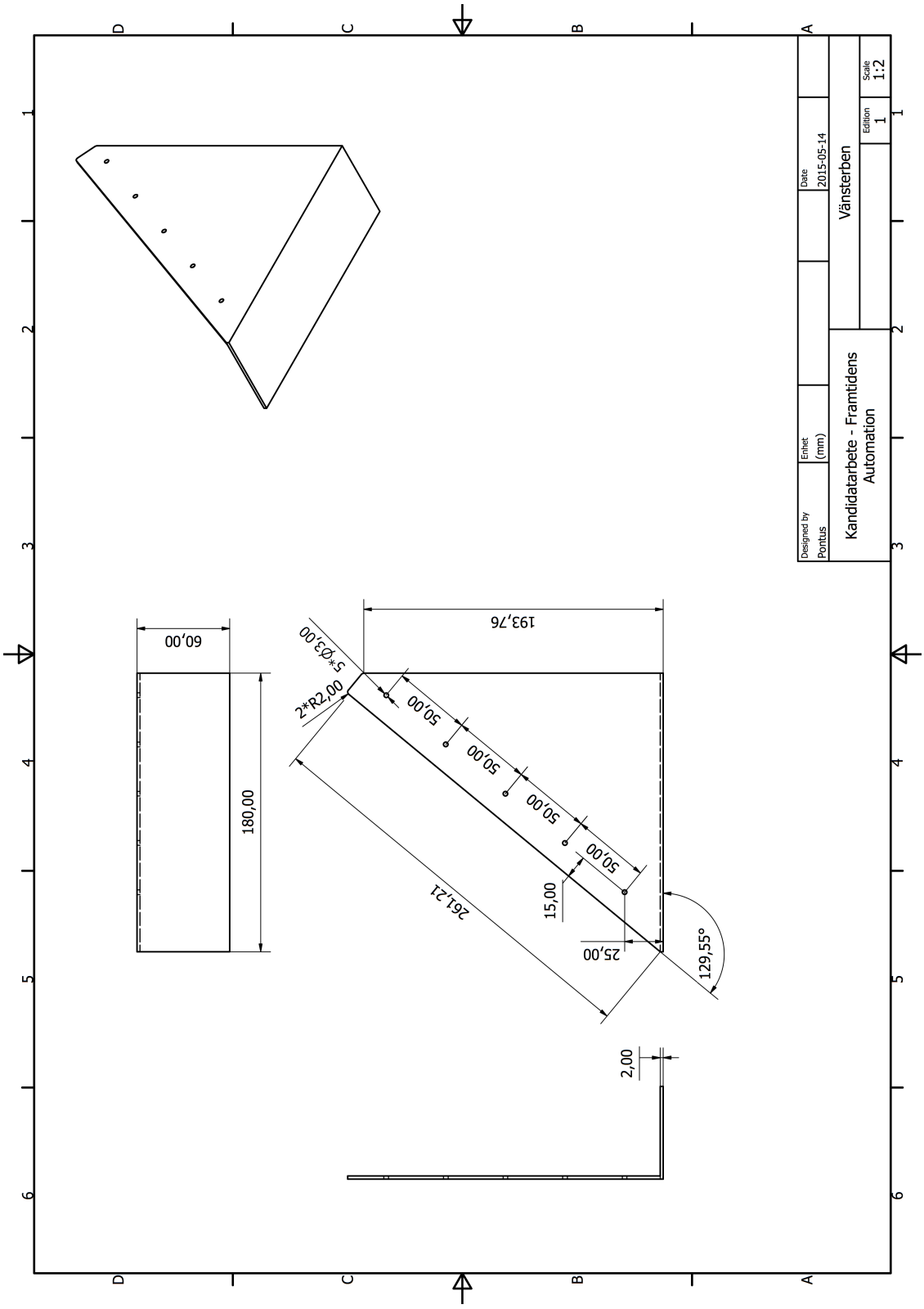
Tabell E.1: *Ritningar över ny och förbättrad hårdvara*

Ritningar	
Ritning A	Översiktritning över transportfixturen
Ritning B	Ben till transportfixturen
Ritning C	Grundplatta till transportfixturen
Ritning D	Mellanplatta till transportfixturen
Ritning E	Profil efter konstruktion
Ritning F	Profil efter justeringar inför tillverkning

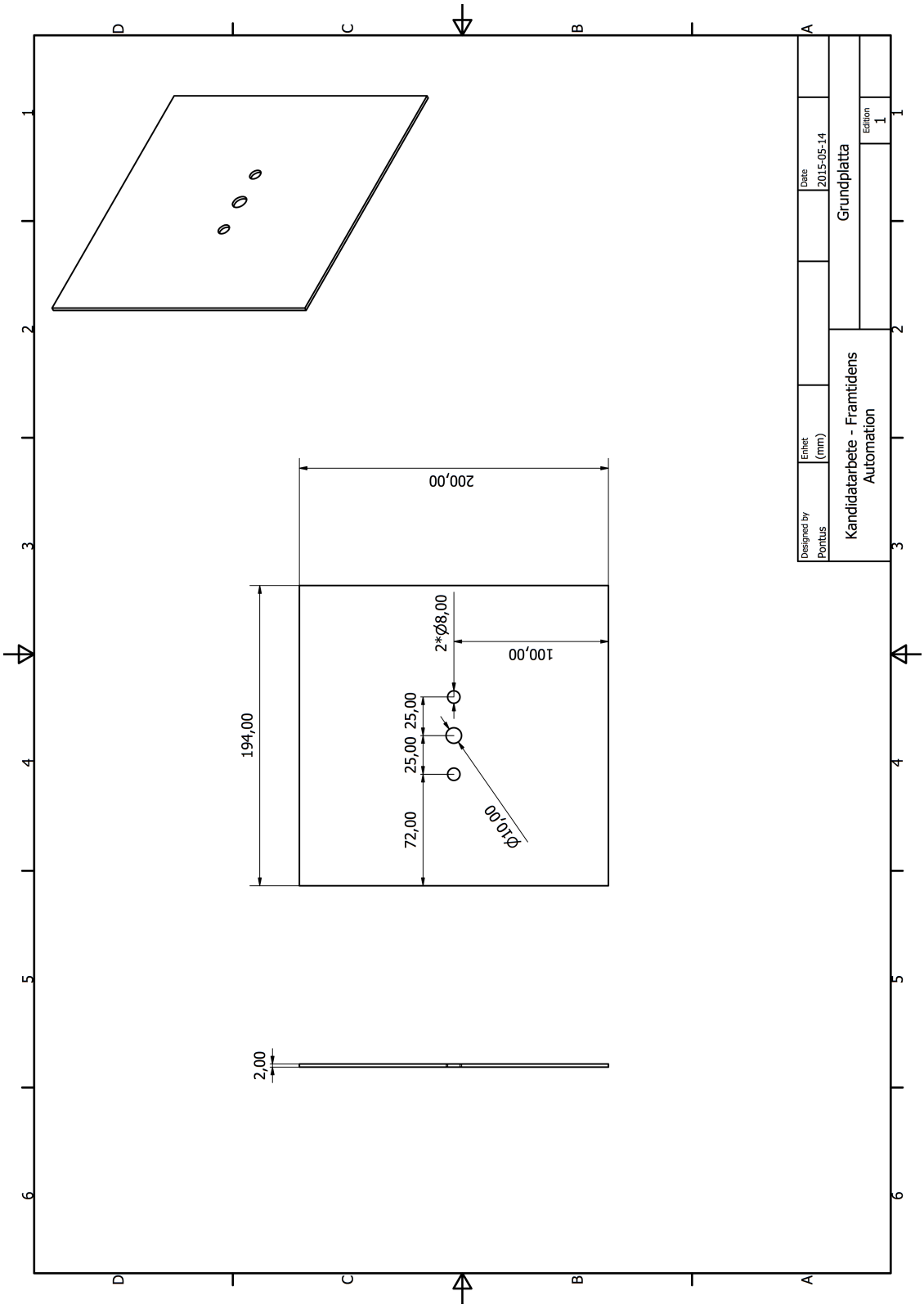
E.1 Ritning A - Översiktsritning över transportfixturen



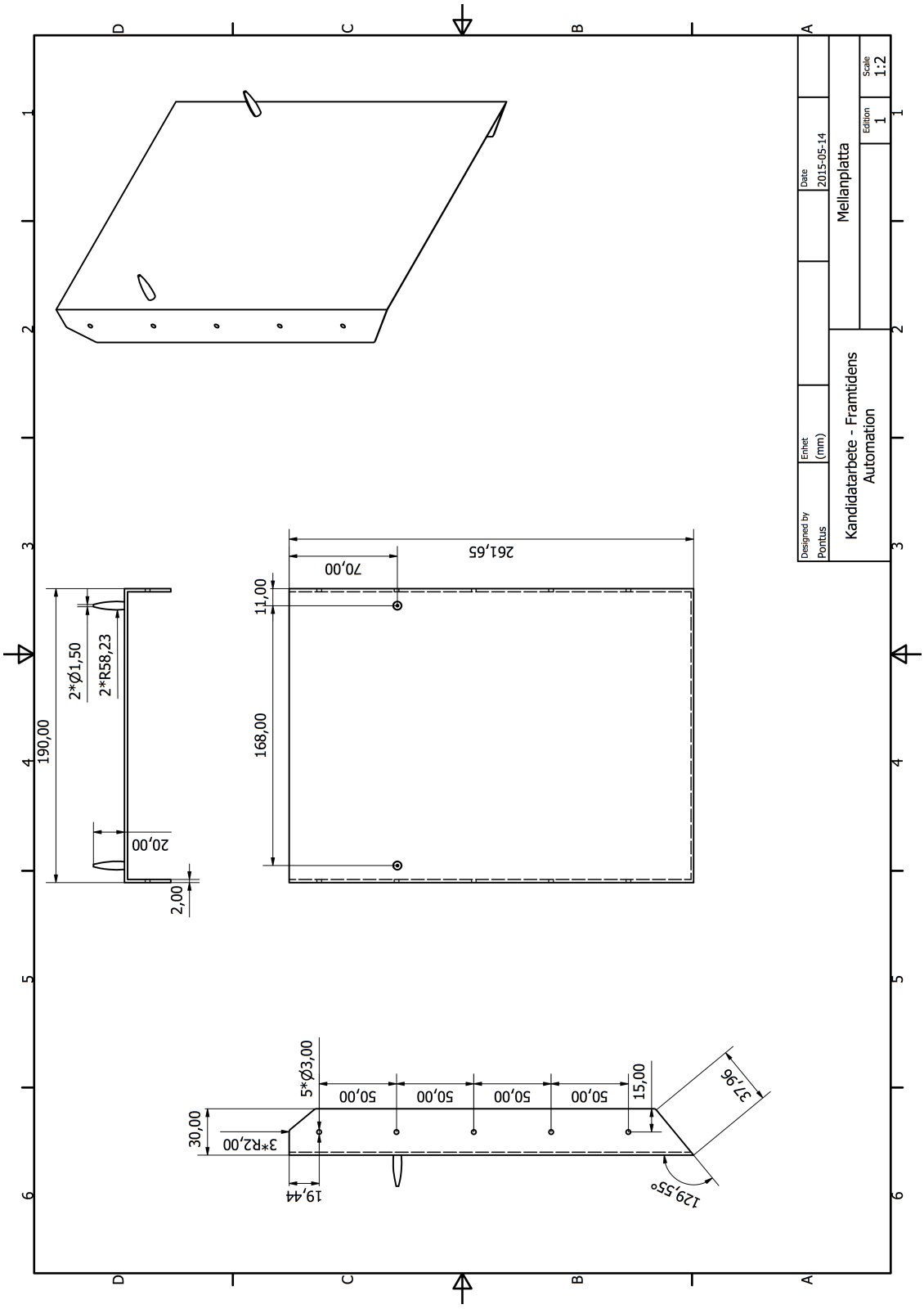
E.2 Ritning B - Ben till transportfixturen



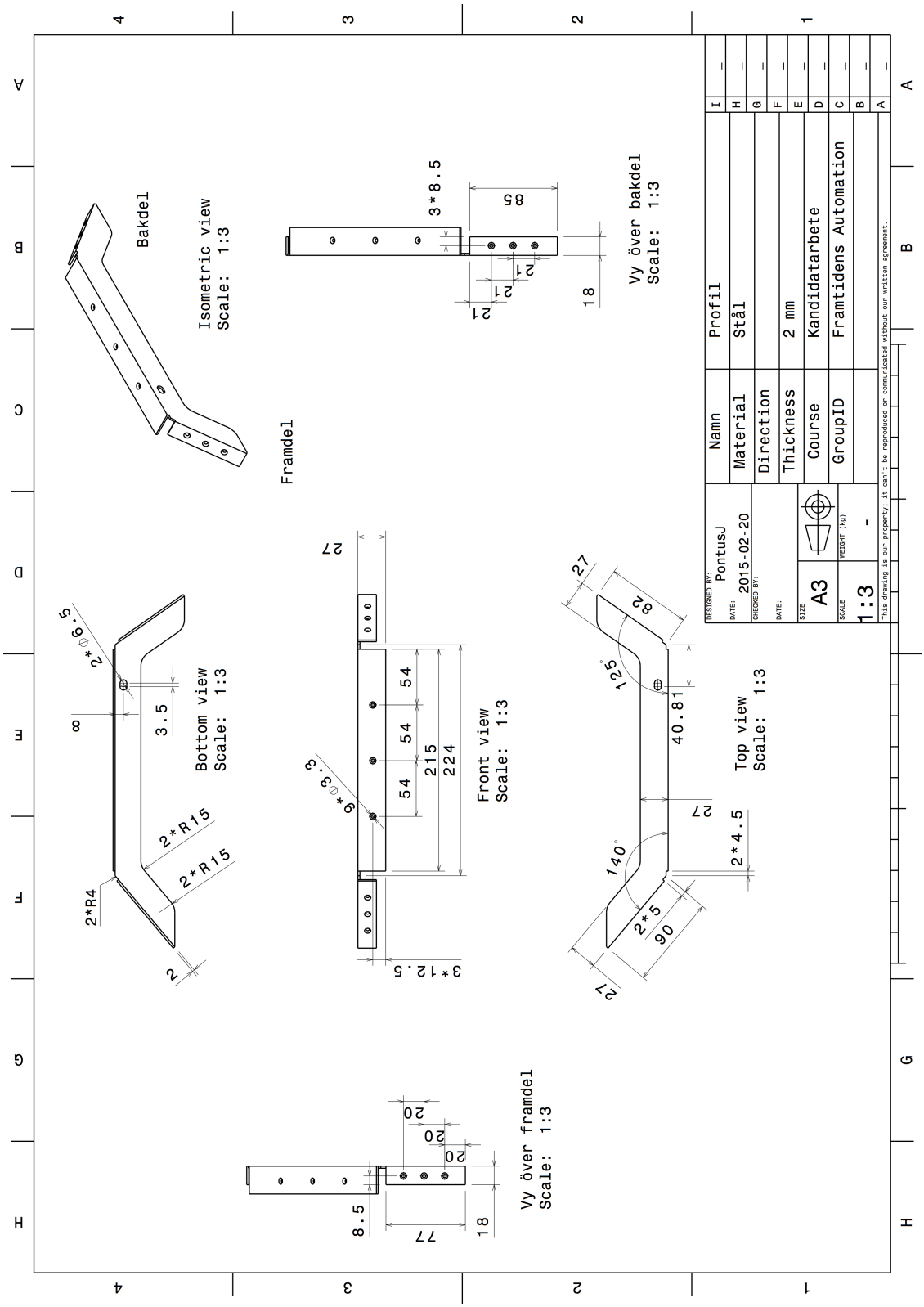
E.3 Ritning C - Grundplatta till transportfixturen



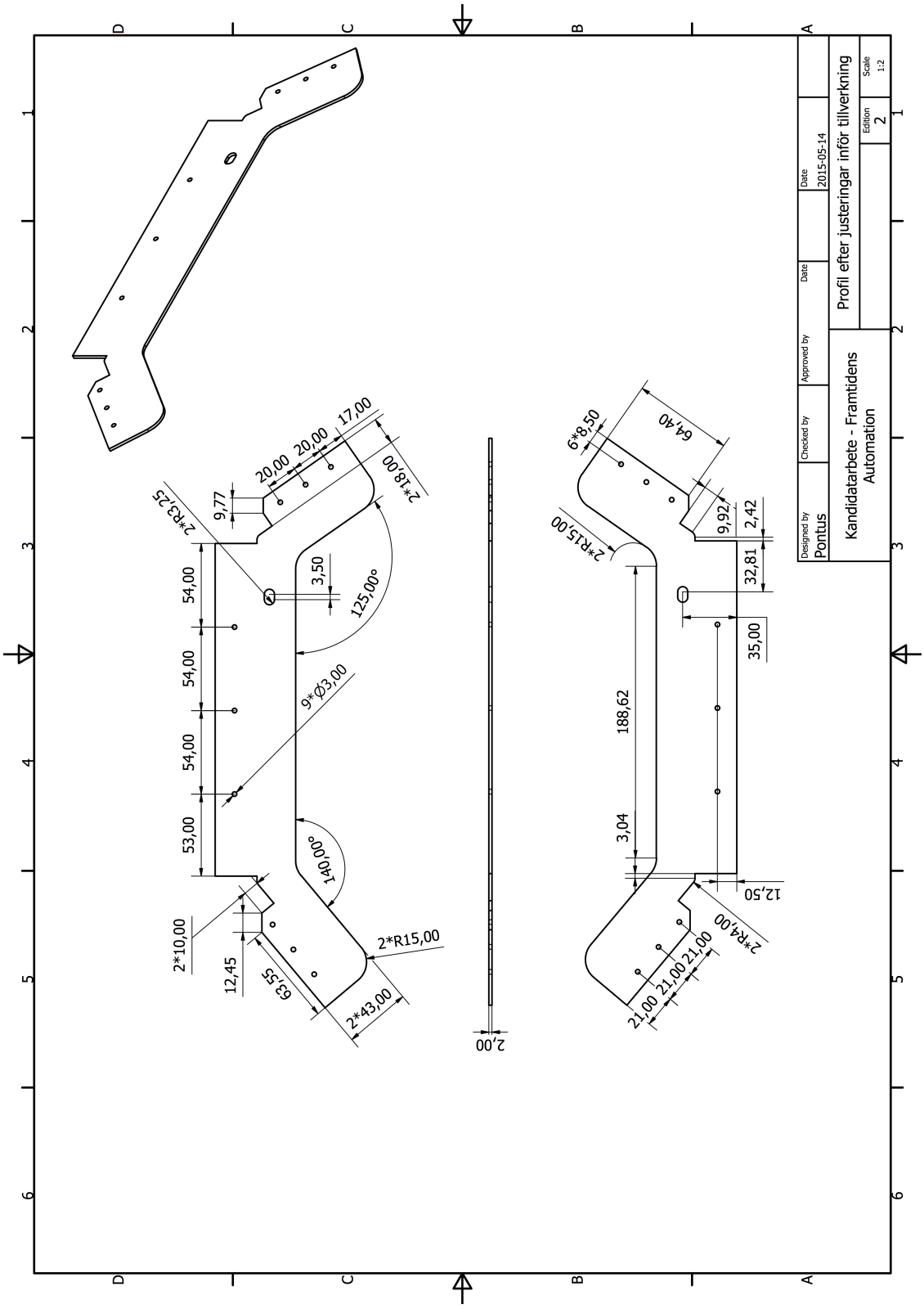
E.4 Ritning D - Mellanplatta till transportfixturen



E.5 Ritning E - Profil efter konstruktion



E.6 Ritning F - Profil efter justeringar inför tillverkning



F

Tidstudie av utvecklingstider

För att utvärdera den potentiella nyttan av den automatiska kodenereringen har en tidsstudie genomförts. Tidsstudien genomfördes under implementeringen av styrningen i *PSL* där 20 Robot Abilities konstruerades i utvecklingsmiljön medans åtta konstruerades manuellt i TIA-Portalen. Från tabell F.1 fås det att utvecklingstiden för en Robot Ability var 65,15% mindre i utvecklingsmiljön än i TIA-Portalen

Tabell F.1: *Jämförelse av utvecklingstider av Robot Abilities mellan utvecklingsmiljön för automatisk kodgenerering och TIA-Portalen*

Utvecklingsmiljön	**1260 s**	**20 st**
Konstruktion	1200 s	20 st
Import	60 s	20 st
TIA-Portalen	**1446 s**	**8 st**

G

Inuti operationsblocken

Figur G.1: *Kod inuti en Operation och en Ability*

H

XML-dokument för en Robot Ability

I detta avsnit listas ett XML-dokument som genererats från den konstruerade utvecklingsmiljön för automatisk kodgenerering, se avsnit 7.2.

```
<?xml version="1.0" encoding="utf-8"?>
<Document xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.
  <DocumentInfo>
    <Created>2015-05-01 13:17:49</Created>
    <ExportSetting>WithDefaults</ExportSetting>
    <InstalledProducts>
      <Product>
        <DisplayName>Totally Integrated Automation Portal</DisplayName>
        <DisplayVersion>V13 SP1 Update 1</DisplayVersion>
      </Product>
      <OptionPackage>
        <DisplayName>TIA Portal Openness</DisplayName>
        <DisplayVersion>V13 SP1</DisplayVersion>
      </OptionPackage>
      <Product>
        <DisplayName>STEP 7 Professional</DisplayName>
        <DisplayVersion>V13 SP1 Update 1</DisplayVersion>
      </Product>
      <Product>
        <DisplayName>WinCC Basic</DisplayName>
        <DisplayVersion>V13 SP1 Update 1</DisplayVersion>
      </Product>
    </InstalledProducts>
  </DocumentInfo>
  <SW.CodeBlock ID="0">
    <AttributeList>
      <AutoNumber>true</AutoNumber>
      <EnableTagReadback>false</EnableTagReadback>
      <EnableTagReadbackBlockProperties />
      <HeaderAuthor />
      <HeaderFamily />
      <HeaderName />
      <HeaderVersion>0.1</HeaderVersion>
      <Interface>
        <Sections xmlns="http://www.siemens.com/automation/Openness/SW/Interface/v1">
```

```
      <Section Name="Input" />
      <Section Name="Output" />
      <Section Name="InOut" />
      <Section Name="Temp" />
      <Section Name="Constant" />
      <Section Name="Return">
        <Member Name="Ret_Val" Datatype="Void" Accessibility="Public" />
      </Section>
    </Sections>
  </Interface>
  <IsIECCheckEnabled>false</IsIECCheckEnabled>
  <MemoryLayout>Optimized</MemoryLayout>
  <Name>R2_Home_to_Flexlink</Name>
  <Number>44</Number>
  <ProgrammingLanguage>LAD_CLASSIC</ProgrammingLanguage>
  <Type>FC</Type>
</AttributeList>
<ObjectList>
  <SW.CompileUnit ID="1" AggregationName="CompileUnits">
    <AttributeList>
      <NetworkSource>
        <FlgNet xmlns="http://www.siemens.com/automation/Openness/SW/NetworkSource
          <Parts>
            <Access Scope="GlobalVariable" Type="Word" UId="40">
              <Symbol>
                <Component Name="R2_Pos" />
              </Symbol>
            </Access>
            <Access Scope="GlobalConstant" Type="Word" UId="42">
              <Constant>
                <ConstantValue>0</ConstantValue>
              </Constant>
            </Access>
            <Access Scope="GlobalVariable" Type="Word" UId="50">
              <Symbol>
                <Component Name="R2_Pos" />
              </Symbol>
            </Access>
            <Access Scope="GlobalConstant" Type="Word" UId="52">
              <Constant>
                <ConstantValue>1</ConstantValue>
              </Constant>
            </Access>
            <Part Gate="Eq" UId="41">
              <TemplateValue Name="SrcType" Type="Type" Value="Word" Automatic="fa
            </Part>
            <Part Gate="Eq" UId="51">
              <TemplateValue Name="SrcType" Type="Type" Value="Word" Automatic="fa
```

```
  </Part>
  <CallRef UId="43" CallType="GlobalCall">
    <CallInfo Name="Ability" BlockType="FB">
      <Instance UId="44" Scope="GlobalVariable">
        <Component Name="Ability.R2_Home_to_Flexlink_DB" />
      </Instance>
    </CallInfo>
  </CallRef>
</Parts>
<Wires>
  <Wire>
    <Powerrail />
    <NameCon UId="43" Name="en" />
    <NameCon UId="41" Name="pre" />
    <NameCon UId="51" Name="pre" />
  </Wire>
  <Wire>
    <IdentCon UId="40" />
    <NameCon UId="41" Name="in1" />
  </Wire>
  <Wire>
    <IdentCon UId="42" />
    <NameCon UId="41" Name="in2" />
  </Wire>
  <Wire>
    <NameCon UId="41" Name="out" />
    <NameCon UId="43" Name="1" />
  </Wire>
  <Wire>
    <IdentCon UId="50" />
    <NameCon UId="51" Name="in1" />
  </Wire>
  <Wire>
    <IdentCon UId="52" />
    <NameCon UId="51" Name="in2" />
  </Wire>
  <Wire>
    <NameCon UId="51" Name="out" />
    <NameCon UId="43" Name="2" />
  </Wire>
  <Wire>
    <NameCon UId="43" Name="3" />
    <IdentCon UId="45" />
  </Wire>
  <Wire>
    <NameCon UId="43" Name="4" />
    <IdentCon UId="46" />
  </Wire>
```

```
            <Wire>
              <NameCon UId="43" Name="5" />
              <IdentCon UId="47" />
            </Wire>
            <Wire>
              <NameCon UId="43" Name="6" />
              <IdentCon UId="48" />
            </Wire>
            <Wire>
              <NameCon UId="43" Name="7" />
              <IdentCon UId="49" />
            </Wire>
          </Wires>
        </FlgNet>
      </NetworkSource>
      <ProgrammingLanguage>LAD_CLASSIC</ProgrammingLanguage>
    </AttributeList>
  </SW.CompileUnit>
  <SW.CompileUnit ID="2" AggregationName="CompileUnits">
    <AttributeList>
      <NetworkSource>
        <FlgNet xmlns="http://www.siemens.com/automation/Openness/SW/NetworkSource
          <Parts>
            <Access Scope="GlobalVariable" Type="Bool" UId="40">
              <Symbol>
                <Component Name="Ability.R2_Home_to_Flexlink_DB" />
                <Component Name="preAction" />
              </Symbol>
            </Access>
            <Access Scope="GlobalConstant" Type="Word" UId="43">
              <Constant>
                <ConstantValue>16#0014</ConstantValue>
              </Constant>
            </Access>
            <Access Scope="GlobalVariable" Type="Word" UId="44">
              <Symbol>
                <Component Name="Robot_R2_DB" />
                <Component Name="PrgNo" />
              </Symbol>
            </Access>
            <Part Gate="Contact" UId="41" />
            <Part Gate="Move" UId="42" DisabledENO="true" EN="true">
              <TemplateValue Name="Card" Type="Cardinality" Value="1" />
            </Part>
          </Parts>
          <Wires>
            <Wire>
              <Powerrail />
```

```
            <NameCon UId="41" Name="in" />
          </Wire>
          <Wire>
            <IdentCon UId="40" />
            <NameCon UId="41" Name="operand" />
          </Wire>
          <Wire>
            <NameCon UId="41" Name="out" />
            <NameCon UId="42" Name="en" />
          </Wire>
          <Wire>
            <IdentCon UId="43" />
            <NameCon UId="42" Name="in" />
          </Wire>
          <Wire>
            <NameCon UId="42" Name="out1" />
            <IdentCon UId="44" />
          </Wire>
        </Wires>
      </FlgNet>
    </NetworkSource>
    <ProgrammingLanguage>LAD_CLASSIC</ProgrammingLanguage>
  </AttributeList>
</SW.CompileUnit>
<SW.CompileUnit ID="3" AggregationName="CompileUnits">
  <AttributeList>
    <NetworkSource>
      <FlgNet xmlns="http://www.siemens.com/automation/Openness/SW/NetworkSource
        <Parts>
          <Access Scope="GlobalVariable" Type="Bool" UId="40">
            <Symbol>
              <Component Name="Ability.R2_Home_to_Flexlink_DB" />
              <Component Name="executing" />
            </Symbol>
          </Access>
          <Access Scope="GlobalVariable" Type="Byte" UId="42">
            <Symbol>
              <Component Name="R2IN_PrgNr_IB" />
            </Symbol>
          </Access>
          <Access Scope="GlobalConstant" Type="Byte" UId="44">
            <Constant>
              <ConstantValue>16#00</ConstantValue>
            </Constant>
          </Access>
          <Access Scope="GlobalConstant" Type="Word" UId="46">
            <Constant>
              <ConstantValue>16#0000</ConstantValue>
```

```
      </Constant>
    </Access>
    <Access Scope="GlobalVariable" Type="Word" UId="47">
      <Symbol>
        <Component Name="Robot_R2_DB" />
        <Component Name="PrgNo" />
      </Symbol>
    </Access>
    <Part Gate="Contact" UId="41" />
    <Part Gate="Ne" UId="43">
      <TemplateValue Name="SrcType" Type="Type" Value="Byte" Automatic="fa
    </Part>
    <Part Gate="Move" UId="45" DisabledENO="true" EN="true">
      <TemplateValue Name="Card" Type="Cardinality" Value="1" />
    </Part>
  </Parts>
  <Wires>
    <Wire>
      <Powerrail />
      <NameCon UId="41" Name="in" />
    </Wire>
    <Wire>
      <IdentCon UId="40" />
      <NameCon UId="41" Name="operand" />
    </Wire>
    <Wire>
      <NameCon UId="41" Name="out" />
      <NameCon UId="43" Name="pre" />
    </Wire>
    <Wire>
      <IdentCon UId="42" />
      <NameCon UId="43" Name="in1" />
    </Wire>
    <Wire>
      <IdentCon UId="44" />
      <NameCon UId="43" Name="in2" />
    </Wire>
    <Wire>
      <NameCon UId="43" Name="out" />
      <NameCon UId="45" Name="en" />
    </Wire>
    <Wire>
      <IdentCon UId="46" />
      <NameCon UId="45" Name="in" />
    </Wire>
    <Wire>
      <NameCon UId="45" Name="out1" />
      <IdentCon UId="47" />
```

```
          </Wire>
        </Wires>
      </FlgNet>
    </NetworkSource>
    <ProgrammingLanguage>LAD_CLASSIC</ProgrammingLanguage>
  </AttributeList>
</SW.CompileUnit>
<SW.CompileUnit ID="4" AggregationName="CompileUnits">
  <AttributeList>
    <NetworkSource>
      <FlgNet xmlns="http://www.siemens.com/automation/Openness/SW/NetworkSource
        <Parts>
          <Access Scope="GlobalVariable" Type="Bool" UId="40">
            <Symbol>
              <Component Name="Ability.R2_Home_to_Flexlink_DB" />
              <Component Name="postAction" />
            </Symbol>
          </Access>
          <Access Scope="GlobalVariable" Type="Bool" UId="42">
            <Symbol>
              <Component Name="R2_home_to_flexlink_done" />
            </Symbol>
          </Access>
          <Part Gate="Contact" UId="41" />
          <Part Gate="SCoil" UId="43" />
        </Parts>
        <Wires>
          <Wire>
            <Powerrail />
            <NameCon UId="41" Name="in" />
          </Wire>
          <Wire>
            <IdentCon UId="40" />
            <NameCon UId="41" Name="operand" />
          </Wire>
          <Wire>
            <NameCon UId="41" Name="out" />
            <NameCon UId="43" Name="in" />
          </Wire>
          <Wire>
            <IdentCon UId="42" />
            <NameCon UId="43" Name="operand" />
          </Wire>
        </Wires>
      </FlgNet>
    </NetworkSource>
    <ProgrammingLanguage>LAD_CLASSIC</ProgrammingLanguage>
  </AttributeList>
```

```
      </SW.CompileUnit>
    </ObjectList>
  </SW.CodeBlock>
</Document>
```