

# Metod för konvertering av SattLine till 800xA



---

**Joaquin Velozo Larrañaga**

Division of Industrial Electrical Engineering and Automation  
Faculty of Engineering, Lund University

# Metod för konvertering av SattLine till 800xA

Utveckling och verifiering av en Python-baserad verktygskedja för strukturell migrering via CB Open Interface



**LUNDS  
UNIVERSITET**

Lunds Tekniska Högskola

**LTH Ingenjörshögskola vid Campus Helsingborg**  
Avdelningen för Industriell elektroteknik och automation

Joaquin Velozo Larrañaga

Handledare: Erik Ottosson  
Examinator: Bernard Schmidt

® Copyright Joaquin Velozo Larrañaga

LTH vid Campus Helsingborg

Lunds universitet

Box 882

251 08 Helsingborg

LTH at Campus Helsingborg

Lund University

Box 882

SE-251 08 Helsingborg

Sweden

Tryckt i Sverige

Lunds universitet

Lund 2026

# Terminologi

**DCS** - Distribuerat styrsystem för industriella processer där styrning, övervakning och reglering är fördelad över flera sammankopplade enheter.

**SattLine** - ABB:s äldre distribuerade styrsystem (DCS) för industriella processer, föregångare till 800xA, använt för styrning, övervakning och reglering i exempelvis kraft- och processindustri.

**800xA** - ABB modern styrsystem (DCS) och automationsplattform för process- och industristyrning som kombinerar styrlogik, operatörsgränssnitt, historik och systemintegration.

**API** - Application Programming Interface. Gränssnitt som möjliggör kommunikation och datautbyte mellan olika program, system eller applikationer på ett kontrollerat och standardiserat sätt.

**Control Builder Open Interface** - Programmeringsgränssnitt (API) i ABB 800xA som möjliggör extern åtkomst och automatisering av projektdata, exempelvis för att skapa, ändra eller läsa moduler, typer och strukturer via skript eller externa verktyg.

**Strukturerade Datatyper** - Sammansatta datatyper som består av flera fält eller komponenter, till exempel signaler, parametrar eller tillstånd samlade i en gemensam struktur.

**Bibliotek** - Samling av återanvändbara typer, funktioner, moduler eller objekt som används för att bygga upp applikationer på ett standardiserat sätt.

**Modul** - Grundläggande programenhet i ett styrsystem som kan innehålla logik, parametrar, variabler och ibland även grafik eller submoduler.

**SFC** - Sequential Function Chart. Grafiskt programspråk enligt IEC 61131-3 för att beskriva sekvensstyrning med steg, övergångar och tillhörande åtgärder.

**ST** - Structured Text. Textbaserat programspråk enligt IEC 61131-3 som används för att skriva logik, beräkningar och styrfunktioner i automationssystem.

**CMT** - Control Module Type. Återanvändbar modultyp i 800xA som definierar struktur, logik och gränssnitt för en viss funktionsmodul i styrsystemet.

**FBT** - Function Block Type. Typdefinition för funktionsblock i 800xA/IEC 61131-3 som kapslar in logik med definierade in- och utgångar.

**P&ID** - Piping and Instrumentation Diagram. Teknisk ritning som visar processutrustning, rörsystem, ventiler, instrument och styrkopplingar i en industriell process.

**Python** – Högnivå kodnings språk.

**XML** - eXtensible Markup Language. Textbaserat format för strukturerade data.

**Hyper-V** - Microsofts virtualiseringsplattform för att skapa och köra virtuella maskiner.

# SAMMANFATTNING

Detta examensarbete undersöker hur befintliga styrsystem i ABB SattLine kan konverteras till ABB System 800xA på ett mer strukturerat, spårbart och delvis automatiserat sätt. Bakgrunden är att många industriella anläggningar fortfarande använder SattLine trots att systemet idag befinner sig i en sen livscykel, vilket skapar behov av modernisering till en plattform med bättre framtida stöd och utvecklingsmöjligheter. Arbetet genomfördes i samarbete med Granitor Systems AB och fokuserade på att ta fram en teknisk metod för att analysera SattLine-filer, översätta relevant information till ett format som kan användas i 800xA och därefter undersöka hur denna information kan importeras i målsystemet.

Arbetet inleddes med en förstudie av SattLine, 800xA och CB Open Interface för att identifiera vilka strukturer, datatyper och objekt som är centrala vid en migrering. Därefter byggdes en testmiljö med virtuella maskiner för båda systemen samt ett avgränsat referensprojekt som användes för analys, utveckling och verifiering. Utifrån denna analys utvecklades en Python-baserad verktygskedja som kan läsa SattLine-filer i textformat, tolka deras innehåll och generera XML-filer anpassade för CB Open Interface. Även verktyg för validering och import till 800xA togs fram, både som kommandoradsverktyg och som grafiska gränssnitt.

Resultatet visar att en strukturell konvertering från SattLine till 800xA är tekniskt möjlig att automatisera till stor del. Examensarbetet resulterade i en fungerande demonstrationslösning bestående av parser, XML-generator, importverktyg och tillhörande testmaterial. Verktygskedjan kunde användas för att läsa SattLine-bibliotek, generera importerbara XML-objekt och skapa objekt i 800xA via CB Open Interface. Samtidigt visar arbetet att en fullständig migrering inte enbart är en fråga om filformat och syntax. Skillnader i systemens objektmodeller, hantering av grafik, biblioteksberoenden och semantisk tolkning innebär att vissa delar fortfarande kräver ingenjörsmässig bedömning och manuell anpassning.

Examensarbetet har tagit fram en praktiskt fungerande proof of concept och en tydlig metod för hur konverteringsarbetet kan genomföras. Arbetet visar vilka delar av migreringen som lämpar sig för automatisering, vilka begränsningar som finns och hur ett internt verktygsstöd kan bidra till ökad reproducerbarhet och effektivitet vid framtida moderniseringsprojekt.

# ABSTRACT

This thesis investigates how existing control systems in ABB SattLine can be converted to ABB System 800xA in a more structured, traceable, and partly automated way. The background is that many industrial plants still rely on SattLine even though the system is now in a late lifecycle phase, creating a need to modernize to a platform with better long-term support and development possibilities. The work was carried out in collaboration with Granitor Systems AB and focused on developing a technical method for analyzing SattLine files, translating relevant information into a format that can be used in 800xA, and then examining how that information can be imported into the target system.

The project began with a pre-study of SattLine, 800xA, and CB Open Interface in order to identify the structures, data types, and object models that are central to a migration process. A test environment with virtual machines for both systems and a limited reference project was then established and used for analysis, development, and verification. Based on this analysis, a Python-based toolchain was developed that can read SattLine files in text format, interpret their contents, and generate XML files adapted for CB Open Interface. Tools for validation and import into 800xA were also developed, both as command-line utilities and graphical interfaces.

The results show that structural conversion from SattLine to 800xA is technically possible to automate to a significant extent. The thesis resulted in a working demonstration solution consisting of a parser, an XML generator, import tools, and supporting test material. The toolchain could be used to read SattLine libraries, generate importable XML objects, and create objects in 800xA through CB Open Interface. At the same time, the work shows that complete migration is not only a matter of file formats and syntax. Differences in the systems' object models, handling of graphics, library dependencies, and semantic interpretation mean that some parts still require engineering judgment and manual adaptation.

This thesis has produced a practical functioning proof of concept and a clear method for how such conversion work can be carried out. The work demonstrates which parts of the migration are suitable for automation, which limitations remain, and how an internal support tool can contribute to greater reproducibility, and efficiency in future modernization projects.

# FÖRORD

Jag vill tacka teamet på Granitor Systems i Malmö för att de gav mig den stora möjligheten att genomföra detta examensarbete tillsammans med dem. Den här erfarenheten har gett mig möjlighet att utveckla mina kunskaper i en professionell miljö och komma närmare det praktiska arbetet inom automation.

Jag vill rikta ett särskilt tack till min handledare och vän Wiktor Gullberg, som hjälpte och stöttade mig genom hela processen. Hans vägledning, engagemang och förtroende har varit mycket viktiga för att detta arbete skulle kunna genomföras. Jag vill också tacka Håkan Uddberg för tålamodet att besvara mina många frågor om SattLine och för att ha delat med sig av sina kunskaper under projektets gång.

Detta examensarbete markerar avslutningen på mina studier inom elektroteknik och automation vid LTH, Campus Helsingborg. Samtidigt har erfarenheten stärkt min övertygelse om min framtida yrkesväg och bekräftat mitt intresse för att utvecklas som automationsingenjör.

# Innehållsförteckning

|       |                                                |    |
|-------|------------------------------------------------|----|
| 1     | Inledning .....                                | 1  |
| 1.1   | Bakgrund .....                                 | 1  |
| 1.2   | Syfte.....                                     | 2  |
| 1.3   | Målformulering.....                            | 2  |
| 1.4   | Problemformulering .....                       | 2  |
| 1.5   | Motivering .....                               | 3  |
| 1.6   | Avgränsningar .....                            | 3  |
| 1.7   | Användning av AI .....                         | 3  |
| 2     | Teknisk bakgrund .....                         | 5  |
| 2.1   | Distribuerade styrsystem (DCS).....            | 5  |
| 2.2   | ABB SattLine .....                             | 5  |
| 2.2.1 | Historisk bakgrund.....                        | 5  |
| 2.2.2 | Systemöversikt och arkitektur.....             | 5  |
| 2.2.3 | Hårdvaruplattform - Series 200.....            | 6  |
| 2.2.4 | Programmeringsspråk och modulhierarki .....    | 7  |
| 2.2.5 | Bibliotek och Start Package .....              | 7  |
| 2.2.6 | Livscykelstatus.....                           | 7  |
| 2.3   | ABB 800xA .....                                | 8  |
| 2.3.1 | Historisk bakgrund.....                        | 8  |
| 2.3.2 | Systemöversikt och arkitektur.....             | 8  |
| 2.3.3 | Hårdvaruplattform - AC 800M .....              | 8  |
| 2.3.4 | Programmeringsmodell och Control Builder ..... | 9  |
| 2.3.5 | Processgrafik och operatörsgrenssnitt.....     | 9  |
| 2.3.6 | Ingenjörsmiljö och livscykel .....             | 9  |
| 2.4   | CB Open Interface .....                        | 10 |
| 2.4.1 | Syfte och placering i systemet.....            | 10 |
| 2.4.2 | Funktionalitet på projektnivå .....            | 10 |
| 2.5   | Tidigare lösningar och relaterat arbete .....  | 10 |
| 3     | Metod.....                                     | 13 |
| 3.1   | Förstudie .....                                | 14 |
| 3.2   | Testmiljö.....                                 | 14 |
| 3.3   | Testprojekt.....                               | 15 |
| 3.4   | Analys av SattLine och 800xA .....             | 15 |
| 3.5   | Utveckling och testning av Pythonverktyg.....  | 15 |
| 3.6   | Källkritik.....                                | 17 |

|       |                                                         |    |
|-------|---------------------------------------------------------|----|
| 4     | Analys.....                                             | 19 |
| 4.1   | Analys av SattLine .....                                | 19 |
| 4.1.1 | SattLines textfilformat och bibliotekens struktur ..... | 20 |
| 4.1.2 | Typdeklaration - RECORD .....                           | 20 |
| 4.1.3 | Moduldeklaration – MODULEDEFINITION.....                | 21 |
| 4.1.4 | Modul typer i SattLine .....                            | 22 |
| 4.1.5 | MODULEPARAMETERS. ....                                  | 22 |
| 4.1.6 | LOCALVARIABLES .....                                    | 23 |
| 4.1.7 | SUBMODULES.....                                         | 24 |
| 4.1.8 | ModuleCode.....                                         | 26 |
| 4.2   | Analys av 800xA med SattLine-mappning.....              | 30 |
| 4.2.1 | Systemhierarki och övergripande arkitektur .....        | 30 |
| 4.2.2 | Bibliotek .....                                         | 31 |
| 4.2.3 | Datatyper .....                                         | 32 |
| 4.2.4 | Moduler och submoduler - mappning mot 800xA .....       | 33 |
| 4.2.5 | Parametrar och variabler .....                          | 34 |
| 4.2.6 | Kodlogik och programspråk.....                          | 35 |
| 4.3   | Konvertering av SattLine-filer till XML .....           | 37 |
| 4.3.1 | Tvästegsflöde och kontrollpunkt.....                    | 37 |
| 4.3.2 | RECORD kontra MODULE - komplexitetsskillnaden .....     | 37 |
| 4.3.3 | XML-struktur för FunctionBlockType.....                 | 38 |
| 4.4   | Skapande av objekt i 800xA med CB Open Interface .....  | 41 |
| 4.4.1 | Förberedelse av utvecklingsmiljö.....                   | 41 |
| 4.4.2 | Python-till-COM-bryggan.....                            | 41 |
| 4.4.3 | Objektsökvägssyntax.....                                | 41 |
| 4.4.4 | Tillgängliga metoder .....                              | 42 |
| 4.4.5 | Felhantering: HRESULT och Message Bucket.....           | 42 |
| 4.4.6 | Importflödet i Python .....                             | 43 |
| 5     | Resultat .....                                          | 45 |
| 5.1   | Översikt över framtagna verktyg.....                    | 45 |
| 5.2   | Resultat från parser och analys av SattLine-filer.....  | 46 |
| 5.3   | Resultat från XML-generering .....                      | 48 |
| 5.4   | Resultat från import till 800xA.....                    | 50 |
| 5.5   | Resultat i form av en användbar arbetsmiljö .....       | 51 |
| 6     | Diskussion .....                                        | 53 |
| 6.1   | Samlad resultatbedömning .....                          | 53 |
| 6.2   | Modellskillnader som påverkar migreringen.....          | 53 |

|     |                                                          |    |
|-----|----------------------------------------------------------|----|
| 6.3 | Verktygens styrkor och begränsningar .....               | 53 |
| 7   | Slutsats .....                                           | 55 |
| 7.1 | Sammanfattning av de viktigaste resultaten .....         | 55 |
| 7.2 | Svar på problemformuleringens frågor .....               | 55 |
| 7.3 | Resultatets bidrag till syftet och hur det används ..... | 56 |
| 7.4 | Etiska aspekter .....                                    | 57 |
| 7.5 | Framtida utvecklingsmöjligheter .....                    | 57 |
| 8   | Referenser .....                                         | 59 |
|     | Appendix A – Viktiga kod bitar .....                     | 61 |
|     | Appendix B – P&ID .....                                  | 69 |
|     | Appendix C – CB Open Interface miljö .....               | 71 |

# 1 Inledning

Detta kapitel ger en övergripande beskrivning av examensarbetets bakgrund, syfte och mål. Här presenteras även problemformuleringen, motivering till det valda ämnet samt de avgränsningar som gjorts.

## 1.1 Bakgrund

Detta examensarbete har utförts på uppdrag av Granitor Systems AB, ett svenskt teknikföretag som levererar lösningar inom automation, energi och industriella IT-system till process- och tillverkningsindustrin. Företaget arbetar med hela livscykeln för industriella anläggningar, från konstruktion och driftsättning till modernisering och service [1].

Många fabriker inom processindustrin använder fortfarande SattLine, ett äldre DCS-system som utvecklades av Alfa Laval och senare övertogs av ABB [2]. Eftersom SattLine inte längre vidareutvecklas och befinner sig i sena livscykelfaser, med begränsat underhåll och reservdelstillgång, finns ett tydligt behov av att modernisera dessa system för att säkerställa driftsäkerhet, framtida support och möjlighet till vidareutveckling. En vanlig målplattform är ABB 800xA, som erbjuder en modern, integrerad och framtidssäker DCS-miljö.

Övergången från SattLine till 800xA är dock komplex, då systemen skiljer sig åt avseende arkitektur, programmeringsstruktur och datastruktur. Examensarbetet adresserar detta problem genom att undersöka och utveckla en metodik för effektiv och tillförlitlig konvertering mellan systemen.

Migrering från SattLine till System 800xA kan genomföras med hjälp av ABB:s egna uppgraderings- och migrationstjänster, där interna verktyg och etablerade arbetsprocesser används för att överföra befintliga applikationer till en modernare plattform. För Granitor Systems AB är detta dock inte alltid ett praktiskt eller ekonomiskt optimalt alternativ. Eftersom tjänsterna är kommersiella, beroende av ABB:s interna verktyg och i stor utsträckning genomförs av ABB:s egna ingenjörer, kan de innebära höga kostnader och begränsad insyn i själva konverteringsprocessen. Det skapar ett behov av att undersöka om delar av migreringen kan förstås, struktureras och automatiseras internt med hjälp av öppna eller dokumenterade gränssnitt.

Examensarbetet tar därför sin utgångspunkt i Granitors behov av ett mer självständigt och reproducerbart arbetssätt vid modernisering av äldre SattLine-anläggningar. Målet är inte att ersätta alla kommersiella migrationslösningar, utan att analysera vilka delar av konverteringsarbetet som kan göras spårbara och delvis automatiserade inom ramen för ett internt verktygsstöd.

Examensarbetet genomfördes under 2026 hos Granitor Systems AB:s avdelning i Malmö, där tillgång fanns till både SattLine- och 800xA-miljöer.

## 1.2 Syfte

Syftet med examensarbetet är att undersöka och utveckla en översiktlig metod för konvertering av befintliga SattLine-projekt till ABB System 800xA. Arbetet syftar till att stödja modernisering av äldre styrsystem inom processindustrin utan att kräva fullständig omprogrammering av befintliga anläggningar. Den förväntade nyttan med examensarbetet är att bidra till minskad omställningstid, reducerade kostnader och ökad kvalitet vid migrering från SattLine till 800xA. Det förväntade resultatet är en verifierad demonstrationslösning som visar hur en konverteringsprocess kan genomföras i praktiken, samt en identifiering av förbättringsområden och begränsningar som kan ligga till grund för framtida vidareutveckling och industriell implementering.

## 1.3 Målformulering

Målet med examensarbetet är att under en period om 15 veckor genomföra en konceptvalidering (proof of concept) för konvertering av styrsystem från SattLine till ABB System 800xA. Målen är avgränsade och utformade så att de är möjliga att verifiera vid arbetets slut. De konkreta målen är att:

- Identifiera och dokumentera centrala skillnader mellan SattLine och 800xA ur ett system-, struktur- och programmeringsperspektiv.
- Utveckla ett avgränsat testprojekt i SattLine som fungerar som referens för konvertering.
- Genomföra en praktisk konvertering av testprojektet till 800xA.
- Verifiera att funktionalitet, styrlogik och struktur bibehålls vid konverteringen.
- Ta fram och dokumentera en generell och reproducerbar metodik för hur liknande konverteringar kan genomföras i framtida projekt.

Om tiden tillåter undersöks även delvis automatiserad konvertering av mer komplexa projektstrukturer för att bedöma metodens skalbarhet.

## 1.4 Problemformulering

För att uppfylla syftet och de uppställda målen med examensarbetet formuleras följande avgränsade problemställningar, vilka examensarbetet avser att besvara:

- 1) Hur är ett SattLine-projekt strukturerat och vilka centrala komponenter måste beaktas vid en konvertering?
- 2) Vilka väsentliga skillnader finns mellan SattLine och ABB System 800xA avseende programmeringsmodell, datatyper och systemarkitektur?
- 3) Hur kan ett avgränsat SattLine-projekt konverteras till 800xA på ett effektivt och tillförlitligt sätt med hjälp av tillgängliga verktyg?
- 4) Vilka verifieringssteg krävs för att säkerställa att funktionalitet, styrlogik och systemstruktur bibehålls efter konvertering till 800xA?
- 5) Vilka begränsningar och risker kan identifieras vid konvertering av mer omfattande eller komplexa SattLine-system, och hur påverkar dessa möjligheten till skalning av den föreslagna metoden?

## 1.5 Motivering

Valet av examensarbete motiveras av att det behandlar ett aktuellt och industriellt relevant problemområde inom processautomation, nämligen modernisering av äldre styrsystem som fortfarande är i drift inom stora delar av processindustrin. Många anläggningar är idag beroende av föråldrade DCS-lösningar, vilket medför ökade risker avseende driftsäkerhet, underhåll och framtida support.

För Granitor Systems AB motiveras examensarbetet av behovet att ta fram en strukturerad och reproducerbar metod för konvertering från SattLine till ABB System 800xA. Ett sådant angreppssätt har potential att effektivisera framtida kundprojekt genom minskad projekttid, lägre kostnader och reducerad teknisk risk vid moderniseringsuppdrag. Examensarbetet utgör därmed ett första steg mot en mer standardiserad arbetsprocess inom detta område.

Ur ett samhälls- och industriperspektiv kan arbetet bidra till att förlänga livslängden på befintliga processanläggningar och underlätta en kontrollerad övergång till moderna styrsystem. På sikt kan sådana moderniseringsinsatser ge förbättrad driftsäkerhet, högre säkerhetsnivåer och bättre energieffektivitet, utan att hela anläggningar behöver byggas om från grunden.

Examensarbetet valdes för att det ger möjlighet att arbeta praktiskt med både äldre och moderna DCS-miljöer samt att fördjupa sina kunskaper inom systemintegration, automationsarkitektur och industriella migrationsprojekt, vilket är särskilt relevant för framtida yrkesverksamhet inom automation och processindustri.

## 1.6 Avgränsningar

Examensarbetet kommer att:

- Fokusera på konceptvalidering av konvertering mellan SattLine och 800xA.
- Begränsas till ett mindre referensprojekt. Inte omfatta fullständig migrering av stora anläggningssystem.
- Inte utvärdera ekonomiska aspekter utöver tidsbesparing och resursanvändning.

## 1.7 Användning av AI

Generativ AI har använts som ett stödverktyg under examensarbetet, främst vid programmering. AI-stödet användes i ett tidigt utvecklingsskede för att snabbt ta fram prototyper, jämföra olika kodstrukturer och identifiera rimliga tillvägagångssätt. AI användes också för att skapa figurer och grafer.



## 2 Teknisk bakgrund

Detta kapitel beskriver de tekniker, system och begrepp som är relevanta för examensarbetet. Syftet är att ge läsaren tillräcklig bakgrund för att kunna förstå de metoder, analyser och resultat som presenteras i senare kapitel.

### 2.1 Distribuerade styrsystem (DCS)

Distribuerade styrsystem (Distributed Control Systems, DCS) är datorbaserade automationssystem som används för att styra och övervaka industriella processer, särskilt i kontinuerliga och batchbaserade anläggningar inom exempelvis kemi, energi, vattenrening, läkemedels- och livsmedelsindustri. Till skillnad från en helt centraliserad styrning fördelas kontrollfunktionen i ett DCS mellan flera lokala styrenheter nära processen, medan övergripande övervakning, operatörsingripanden och ingenjörsarbete sker från centrala arbetsstationer [3].

Denna distribuerade arkitektur ger god skalbarhet och hög tillgänglighet eftersom olika delprocesser kan hanteras i separata kontrollslingsor samtidigt som hela anläggningen kan övervakas från kontrollrummet. Ett DCS omfattar typiskt fältinstrument, I/O-moduler, kontrollers, kommunikationsnätverk samt operatörs- och ingenjörstationer. En viktig styrka med denna typ av system är att redundans kan byggas in på flera nivåer, exempelvis i styrsystem, nätverk och servermiljöer, vilket minskar risken för driftavbrott och gör lösningen väl lämpad för processindustri där kraven på driftsäkerhet, överblick och långsiktig utbyggbarhet är höga [3].

### 2.2 ABB SattLine

#### 2.2.1 Historisk bakgrund

SattLine har sitt ursprung i det svenska företaget Elektronlund, grundat 1955 i Malmö. Genom en rad fusioner och namnbyten; Satt Electronics, Ahlsell Control, SattControl växte verksamheten till en ledande aktör inom europeisk processautomation och slogs 1986 samman med Alfa Laval till Alfa Laval Automation. Under början av 1990-talet lanserades SattLine som ett objektorienterat distribuerat styrsystem (DCS) för processautomation, med särskilt fokus på livsmedels-, kemi- och läkemedelsindustri.

År 1998 förvärvades Alfa Laval Automation av ABB, varefter SattLine blev en del av ABB:s portfölj av styrsystem. Systemet fick stor spridning och är fortfarande i drift i många anläggningar, trots att det idag betraktas som ett äldre system i sena livscyklifaser med begränsad vidareutveckling [2].

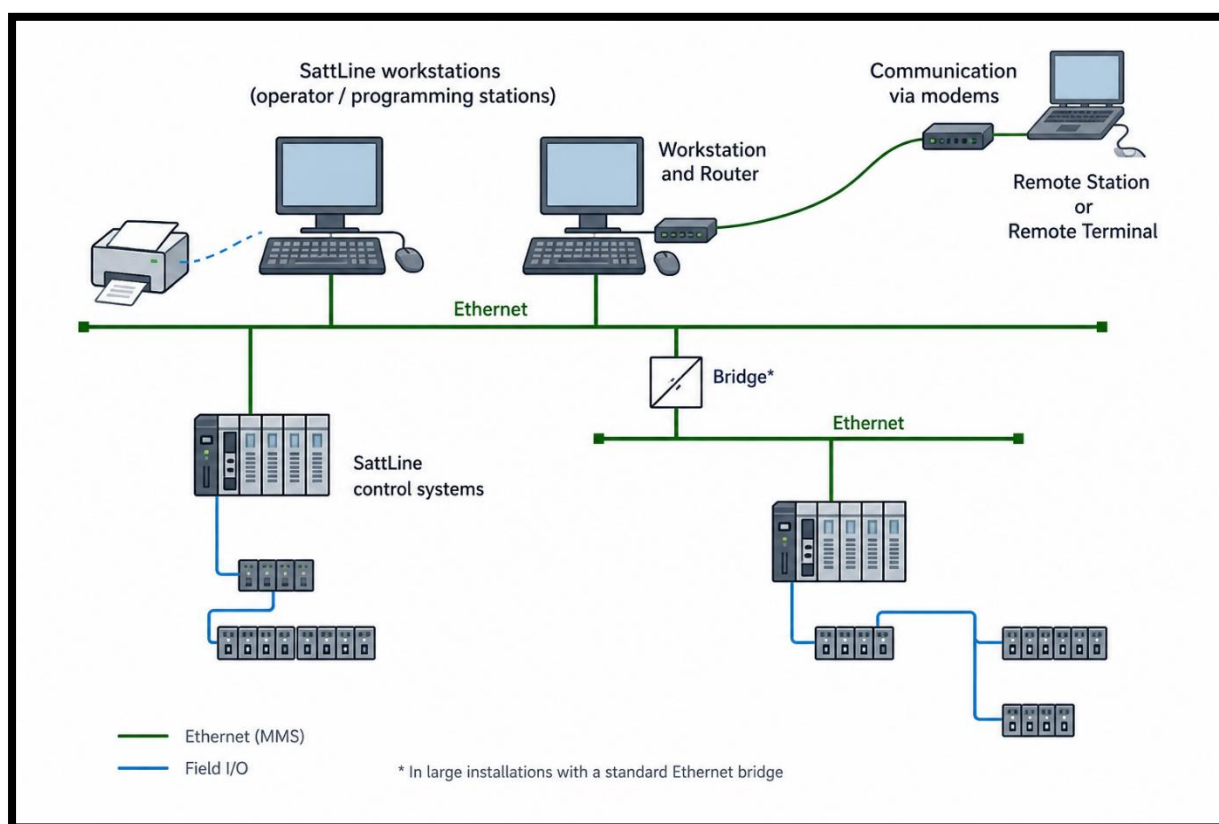
#### 2.2.2 Systemöversikt och arkitektur

SattLine är uppbyggt som ett distribuerat styrsystem där arbetsstationer och kontrollers samverkar över ett lokalt Ethernet-nätverk se Figur 2.1. Arbetsstationerna används för att utveckla, testa och dokumentera applikationer samt för operatörsövervakning med

processgrafik och dialoger, medan kontrollers utför regleringen och kommunicerar med fältinstrument via I/O-moduler.

En central designprincip är att ett gemensamt styrspråk (SattLine Language) används genom hela systemet: samma språk används för operatörsgränssnitt, sekvens- och logikstyrning samt överordnade funktioner. Programmen byggs upp av en hierarki av moduler som motsvarar funktioner och processobjekt, vilket gör det möjligt att återanvända färdiga lösningar och bygga upp applikationsspecifika bibliotek.

Kommunikationen mellan arbetsstationer och kontrollers sker via MMS-protokollet (Manufacturing Message Specification) över Ethernet med stöd för redundanta nät, och kompletteras med ControlNet för distribuerad I/O samt SattBus, COMLI och seriella länkar för fältkommunikation [4][5].



Figur 2.1: Översiktsdiagram av SattLine-systemarkitektur

### 2.2.3 Hårdvaruplattform - Series 200

Styrsystemen i SattLine baseras på hårdvaruplattformen Series 200, som omfattar CPU-enheter, kommunikationsmoduler, I/O-enheter och terminalbaser. SattLine-specifika CPU-enheter inkluderar modellerna 200-CPU30/40-SL och 200-CPU40/40-SL, med klockfrekvenser runt 16–29 MHz och 4 MB minne, samt senare CPU80 som möjliggör SattLine version 3.0/3.1. Samtliga CPU:er har flyttalsenhet, realtidsklocka och batteribackup för minne.

I/O-systemet kan konfigureras med centrala I/O-moduler i samma rack som CPU:n eller som fjärr-I/O via ControlNet, vilket är gynnsamt vid stora eller geografiskt utspridda anläggningar. Plattformen stödjer både analoga och digitala I/O-signaler [6].

#### 2.2.4 Programmeringsspråk och modulkierarki

SattLine använder ett gemensamt grafiskt och textuellt styrspråk där applikationer byggs upp av moduler som kan innehålla både programkod och grafik. Programmeringen sker i form av:

- **Equation blocks** för textbaserade uttryck, beräkningar och logik.
- **Sequence blocks** för sekvensstyrning baserad på Sequential Function Chart (SFC) med steg, övergångar och åtgärder.
- **Procedurer och funktioner** för systemtjänster såsom datalagring, historik, utskrift och kommunikation.

Moduler organiseras hierarkiskt i flera nivåer, där modultyper definierar struktur och beteende och kan instansieras till flera modulinstanser med egna parametrar och variabler. Exekveringen styrs av ScanGroups som anger vilken del av programmet som körs i vilken nod och med vilken cykeltid och prioritet [4][5].

#### 2.2.5 Bibliotek och Start Package

SattLine levereras med omfattande standardbibliotek, bland annat *NucleusLib* (mastermoduler och systemfunktioner), *ConfigLib* (globala datatyper), *EventLib* (larm- och händelsehantering) och *ControlLib* (regulatorer). Utöver dessa finns ett särskilt Start Package som innehåller färdiga programmallar och bibliotek avsedda att snabbt skapa en fungerande standardkonfiguration.

Start Package definierar bland annat hur larm, historik, printerhantering, tidsynkronisering och grundläggande kommunikation implementeras, samt hur flera styrsystem och arbetsstationer samverkar i en hel anläggning [7].

#### 2.2.6 Livscykelstatus

SattLine befinner sig idag i sena livscykelstadiet, där hårdvaruplattformen Series 200 och äldre Windows-baserade HMI-miljöer har klassificerats som Classic eller senare i ABB:s livscykelmodell (Active, Classic, Limited, Obsolete) [8]. Livscykelperspektivet är relevant eftersom industriella automationssystem förändras över tid och genomgår olika faser där kraven på drift, underhåll, vidareutveckling och förändringshantering ser olika ut [9]. När ett system befinner sig i en sen livscykelstadium blir därför frågor om modernisering, migrering och långsiktig förvaltning särskilt viktiga. För SattLine innebär detta begränsad vidareutveckling, ökande svårigheter med reservdelstillgång och ett tydligt behov av migrationslösningar till system som 800xA. ABB erbjuder därför särskilda evolutions- och uppgraderingskoncept där SattLine-applikationer kan integreras eller stegvis migreras till 800xA med hjälp av konverteringsverktyg och standardiserade metoder [10].

## 2.3 ABB 800xA

### 2.3.1 Historisk bakgrund

System 800xA (Extended Automation) introducerades av ABB som en nästa generations automationsplattform som kombinerar klassiska DCS-funktioner med IT-integration, informationshantering och asset management. Plattformen är avsedd att utgöra ett gemensamt nav för styrning, drift, teknik, historik, rapportering och integration mot affärs- och underhållssystem, och ersätter/omfattar äldre system som Advant, Master och SattLine genom olika evolutionslösningar.

Kärnidén är att alla delar av systemet AC 800M-kontrollers, I/O, operatörsarbetsplatser, servrar och externa klienter delar en gemensam objekt- och informationsmodell (Aspect Object-konceptet) och kommunicerar via standardiserade Ethernet-baserade nätverk samt stöd för ett stort antal fältbussar [11][12].

### 2.3.2 Systemöversikt och arkitektur

I en typisk 800xA-installation ingår följande nodtyper:

- Controller-noder (AC 800M) som utför den tidskritiska regleringen och hanterar lokala eller distribuerade I/O-system.
- Server-noder (Aspect, Connectivity, History, Batch, Information Management) som tillhandahåller databaser, historik, larm- och händelselistor, rapportering och externa gränssnitt.
- Client-noder (Operator och Engineering Workplaces) som visar processgrafik, larm, trender och ger åtkomst till engineering-verktyg som Control Builder och Graphics Builder.
- Informationsklienter för tunn åtkomst till process- och historikdata via exempelvis webbklienter eller rapportverktyg.

Noderna är anslutna till ett server-/klientnätverk för servrar och operatörsstationer samt till ett eller flera Ethernet-baserade kontrollnätverk för kommunikation med controllers. Nätverken byggs ofta upp med redundanta switchar och separerade nätverkssegment. I/O och fältenheter ansluts via ModuleBus (för S800 I/O) samt via fältbussar och industriprotokoll såsom PROFINET, PROFIBUS DP, FOUNDATION Fieldbus, HART, Modbus TCP och IEC 61850, antingen via kommunikationsmoduler i AC 800M eller via externa gateways [12].

### 2.3.3 Hårdvaruplattform - AC 800M

AC 800M-familjen består av modulära controllers med CPU-moduler, strömförsörjning, kommunikationsmoduler och lokala/fjärr-I/O-moduler. Varje CPU har normalt två Ethernet-portar för kommunikation mot servrar, klienter och andra controllers, samt plats för minneskort där applikationen lagras. Redundans kan införas på flera nivåer, exempelvis dubblade CPU:er, redundanta kommunikationsmoduler och redundanta I/O-vägar.

I/O-sidan domineras av S800 I/O, en distribuerad modulär I/O-familj som monteras på ModuleBus och stödjer ett brett spektrum av analoga och digitala signaltyper, samt S900 I/O

för Ex-klassade (explosionsklassade) miljöer. I/O-enheter kan placeras nära processen som fjärr-I/O, vilket reducerar kablage och ökar flexibiliteten vid större anläggningar [12][13].

### **2.3.4 Programmeringsmodell och Control Builder**

Styrlogiken i AC 800M programmeras i verktyget Control Builder, som följer IEC 61131-3 programmeringsstandarden och stödjer språken Function Block Diagram (FBD), Sequential Function Chart (SFC), Structured Text (ST) och i vissa fall Ladder Diagram (LD) [12][14]. Utöver dessa finns objektsorienterade kontrollmoduler (Control Modules) och Diagrams som används för att kapsla funktionalitet i återanvändbara byggblock.

Control Builder organiserar applikationen i projekt som kan innehålla flera kontrollers och bibliotek. Användaren kan skapa egna datatyper, funktionsblockstyper och kontrollmodultyper, samt paketera dem i bibliotek som delas mellan projekt. Kommunikation mellan kontrollers sker genom access-variabler, globala variabler och kommunikationsblock.

Systemet tillhandahåller omfattande standardbibliotek för kommunikation, larm- och händelsehantering, tidtagning, matematik, diagnostik och reglering [12].

### **2.3.5 Processgrafik och operatörsgränssnitt**

Operatörsgränssnittet i 800xA bygger på Process Graphics, där grafiska element kopplas till objekt i Aspect Object-modellen. Processbilder visas i Operator Workplace och innehåller dynamiska symboler för exempelvis ventiler, motorer, tankar och mätare, med koppling till larm, trender och faceplates.

Grafiken konfigureras i Graphics Builder, som ger stöd för standardiserade symbolbibliotek och så kallade "solution libraries" med branschspecifika byggblock. Det finns också möjlighet att skapa egna byggblock med definierade parametrar, larm och faceplates. Med hjälp av expressions och expression variables kan grafiska egenskaper kopplas till olika processtillstånd.

Jämfört med SattLine är processgrafiken i 800xA mer integrerad med objektmodellen och ger bättre stöd för sen bindning, återanvändning samt hantering av larm, historik och navigering [12].

### **2.3.6 Ingenjörsmiljö och livscykel**

800xA har en integrerad ingenjörsmiljö där Control Builder, Graphics Builder, larm- och historikkonfiguration, batch och asset management hanteras inom samma system. Flödet omfattar typiskt dimensionering och design, konfiguration av hårdvara och styrlogik, skapande av processgrafik, konfiguration av larm/historik och testning, inklusive simulering och online-ändringar.

Plattformen följer ABB:s livscykelmodell med kontinuerliga uppdateringar och tydliga vägar för uppgradering mellan versioner. Detta är betydelsefullt ur ett livscykelperspektiv eftersom modernare automationsplattformar ger bättre förutsättningar för vidareutveckling, underhåll och långsiktig förvaltning än äldre systemmiljöer. System 800xA fungerar därför som ett naturligt målsystem för migration från äldre DCS-plattformar, inklusive SattLine [8][9][10].

## 2.4 CB Open Interface

### 2.4.1 Syfte och placering i systemet

CB Open Interface (Control Builder Open Interface) är ett programmeringsgränssnitt runt Control Builder/AC 800M-miljön som möjliggör att externa applikationer kan läsa och skriva projektrelaterade data samt processtillstånd utan att gå via operatörsgränssnittet. Gränssnittet riktar sig till utvecklare som vill bygga egna verktyg för exempelvis automatiserad dokumentation, generering av Control Builder-objekt eller online kommunikation mot körande applikationer [15].

CB Open Interface använder samma interna objektmodell som Control Builder, vilket gör att exempelvis projekt, bibliotek, applikationer, controllers, typer, etc. kan behandlas som programmerbara objekt. Åtkomsten till dessa sker via ett COM (Component Object Model) -baserat API (Application Programming Interface), alltså genom att ett externt program anropar metoder och egenskaper i Control Builder som vanliga programobjekt. När objektens innehåll ska beskrivas eller överföras används en XML-representation, där information om objektens uppbyggnad, parametrar, kopplingar m.m. skrivs i ett strukturerat textformat. Det betyder att COM står för själva kommunikationen med systemet, medan XML används för att beskriva den data och de objekt som kommunikationen gäller.

### 2.4.2 Funktionalitet på projektnivå

CB Open Interface ger åtkomst till att:

- Skapa, öppna och stänga Control Builder-projekt.
- Skapa, kopiera, modifiera och ta bort bibliotek, applikationer och kontrollerkonfigurationer.
- Skapa nya datatyper, funktionsblockstyper och kontrollmodultyper från XML-mallar.
- Skapa och konfigurera instanser (typsinstanser) och koppla dem till I/O-kanaler och variabler.
- Köra validering, bygga (compile) och nedladdning (download) av applikationer.

På detta sätt kan exempelvis ett externt konverteringsverktyg översätta ett SattLine-projekt till motsvarande Control Builder-struktur genom att generera XML-beskrivningar av moduler, variabler och signaler och sedan låta CB Open Interface skapa motsvarande objekt i ett 800xA-/AC 800M-projekt [15].

## 2.5 Tidigare lösningar och relaterat arbete

Tidigare industriella migrationsprojekt visar att övergång från SattLine till System 800xA är tekniskt möjlig och redan har genomförts i större anläggningar. Dessa lösningar bygger dock i huvudsak på ABB:s egna verktyg och tjänster, vilket innebär att detaljerade metoder och konverteringsverktyg inte är publikt tillgängliga [16][17][18].

Ett närliggande akademiskt arbete från LTH behandlar modernisering av ett äldre Uniview-baserat styrsystem till ABB 800xA. Även om arbetet inte behandlar SattLine visar det att

migrering till 800xA i praktiken omfattar mer än överföring av styrlogik, exempelvis integration av äldre utrustning, utveckling av operatörsgränssnitt, testning och driftsättning [19].

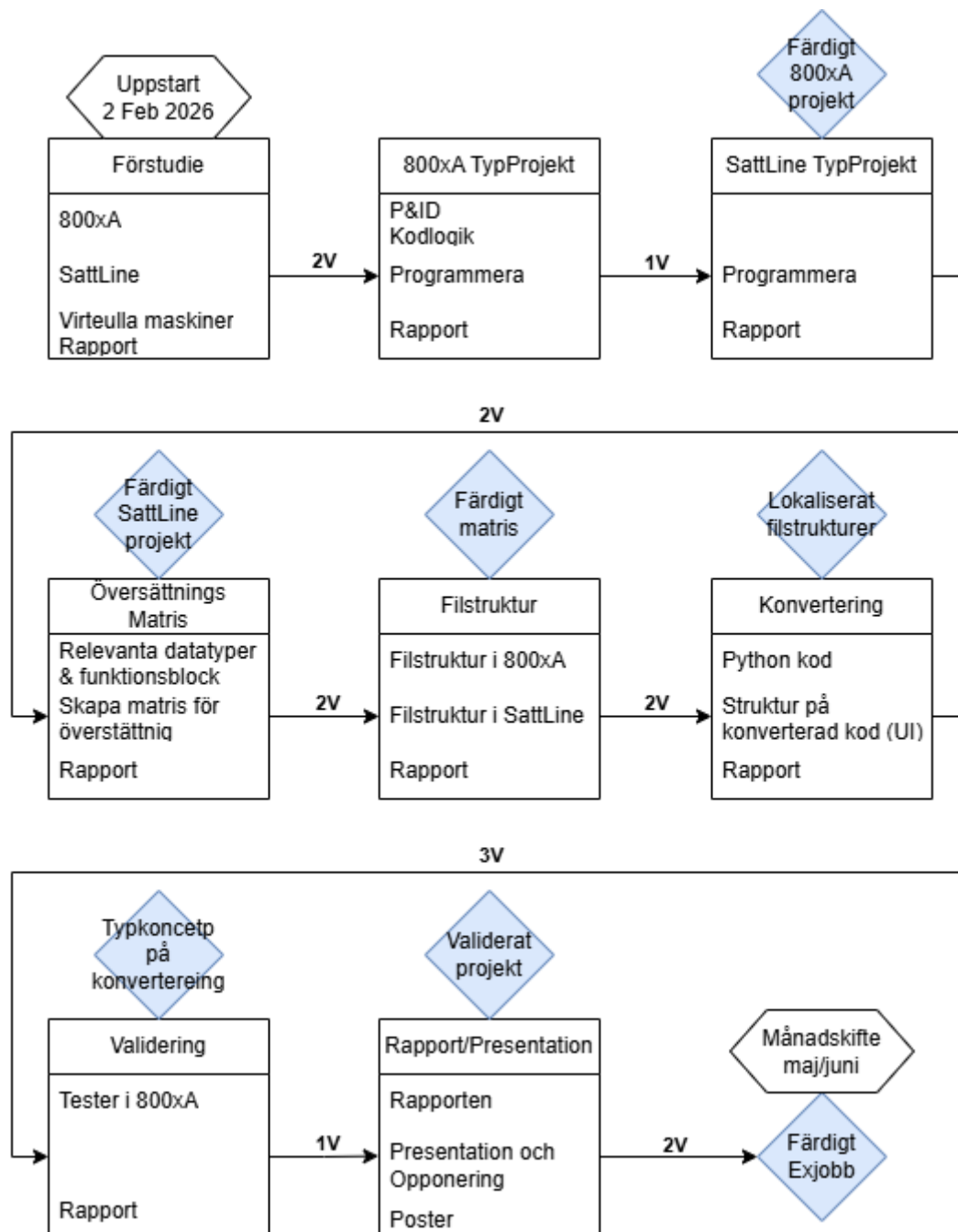
För övriga aktörer återstår i praktiken antingen att köpa ABB:s tjänsten eller att manuellt återskapa applikationen i 800xA med omfattande handpåläggning. Det saknas publikt beskrivna metoder och öppna verktyg som visar hur konverteringen kan utföras av tredje part.

Det är denna uppmaning examensarbetet adresserar på uppdrag av Granitor Systems AB: att utforska om konvertering är möjlig in-house, att identifiera ett sätt att utföra den med öppna gränssnitt, och att demonstrera metoden genom ett proof-of-concept.



### 3 Metod

Arbetet genomfördes som en konceptvalidering med syfte att undersöka och demonstrera genomförbarheten av en automatiserad migrering från SattLine till 800xA. Figur 3.1 visar ett WBS-schema, Work Breakdown Structure, som under examensarbetets gång användes för att strukturera och planera de olika faserna. En WBS används inom projektledning för att dela upp projektets omfattning i hanterbara arbetsmoment som kan planeras, följas upp och utvärderas systematiskt. I detta arbete användes strukturen för att dela upp examensarbetet i sex huvudfaser med specificerade tidsramar och uppgifter kopplade till varje fas [20].



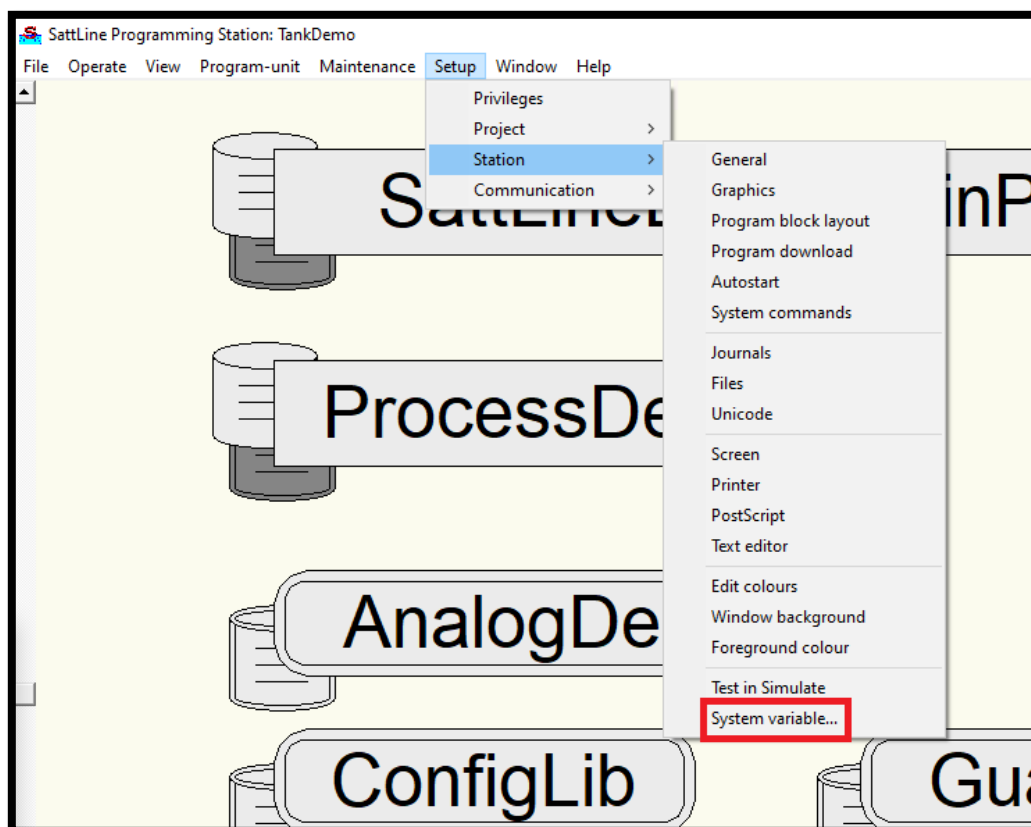
Figur 3.1: WBS för examensarbete som visar projektets delmoment och tillhörande mål.

## 3.1 Förstudie

Den inledande fasen av examensarbetet utgjordes av en förstudie i syfte att bygga en kunskapsgrund kring de två styrsystemen. Arbetet innebar granskning av teknisk dokumentation och manualer för SattLine respektive 800xA, se introduktion i avsnitt 2.2 och 2.3, samt av CB Open Interface-manualen, se avsnitt 2.4. Målet var att identifiera motsvarande strukturer och funktioner mellan systemen som underlag för den översättningsmatris som krävs för att kartlägga relevanta datatyper och funktionsblock.

## 3.2 Testmiljö

Testmiljön sattes upp i Hyper-V och bestod av två separata virtuella maskiner, där den ena hade 800xA installerat och den andra SattLine. Den virtuella miljön var avgörande för att kunna arbeta med systemen på ett säkert och kontrollerat sätt, samt för att möjliggöra repeterbara tester, export och import av projektdata samt felsökning. En förutsättning för att kunna analysera SattLines filformat i textform var även att aktivera systemvariabeln *SaveInPrettyPrint* i SattLine-miljön se Figur 3.2, så att programenheter och bibliotek sparades som strukturerad, läsbar text istället för i ett internt binärt format.



Figur 3.2: Bild på var man kan ställa in en systemvariabel

### 3.3 Testprojekt

Som underlag för den fortsatta konverteringsstudien skapades ett avgränsat testprojekt i både SattLine och 800xA. Testprojektet baserades på ett enkelt tanksystem enligt tillhörande P&ID [21], med cistern, inloppsventil, utloppsventil och nivågivare, se Appendix B. Syftet med testprojektet var att skapa ett gemensamt referensfall där samma processfunktion kunde analyseras i båda systemen. Därmed blev det möjligt att jämföra hur signaler, datatyper, styrlogik, moduler och objektstruktur representeras i SattLine respektive 800xA.

I 800xA kunde testapplikationen byggas med Control Builders standardstruktur genom att skapa ett nytt projekt, definiera applikation, signaler, datatyper och styrlogik samt koppla dessa till den process struktur som motsvarade P&ID:t. I SattLine behövde utvecklingen istället ske inom ramen för det mall-/startsystem (Start Package, se avsnitt 2.2.5) som redan innehöll generella programenheter, bibliotek och systemfunktioner. Testprojektet implementerades därför genom att anpassa befintliga programmallar samt utnyttja de bibliotek för IO och processobjekt som ingår i Start Package.

### 3.4 Analys av SattLine och 800xA

Arbetsmomentet utgjordes av en systematisk kartläggning av SattLines underliggande kodarkitektur och filformat, samt av 800xA/Control Builders objekt- och typmodell. Metodiskt skedde analysen i tre steg:

- 1) Praktisk användning av respektive miljö: bibliotek och moduler skapades, sparades och öppnades i både SattLine och Control Builder för att observera vilken struktur systemen exponerar för användaren.
- 2) Granskning av exporterade textfiler från SattLine (aktiverade via *SaveInPrettyPrint*) respektive XML-scheman för CB Open Interface (se avsnitt 2.4).
- 3) Jämförande kartläggning där motsvarigheter mellan systemen identifierades på tre nivåer: systemhierarki, datatyper samt programmeringsmodell (moduler, parametrar, variabler och kodblock).

### 3.5 Utveckling och testning av Pythonverktyg

Baserat på analysen, som presenteras i kapitel 4, utvecklades en verktygskedja i Python i två distinkta steg med ett granskningsskede emellan. Det första steget tar in SattLine-filer som indata och producerar en XML-fil per datatyp och modul som hittas i SattLine-filen. Dessa XML-filer granskas, och kompletteras eller redigeras manuellt vid behov, innan de förs vidare till det andra steget som automatiserar importen till Control Builder via CB Open Interface, introducerat i avsnitt 2.4. Uppdelningen i två moment är ett medvetet designval som skapar en kontrollpunkt där konverteringsresultatet kan verifieras innan något faktiskt skrivs till Control Builder.

Utvecklingen skedde iterativt: parser och XML-emitter byggdes upp stegvis genom att börja med att konvertera enstaka datatyper och därefter successivt utökas till att hantera hela bibliotek. Motsvarande iterativa ansats användes för importdelen, där anrop till CB Open

Interface-metoderna testades. Som stöd i utvecklingen användes även AI-assisterad programmering i ett tidigt skede för snabb konceptualisering och framtagning av prototypkod. Syftet med detta var att snabbare kunna undersöka om textparsning, XML-generering och import via CB Open Interface var en genomförbar väg för migrering från SattLine till 800xA. AI användes därmed som ett utvecklingshjälpmedel, medan mappningsbeslut, verifiering av kod och tolkning av resultat genomfördes manuellt utifrån systemdokumentation, testmiljö och praktiska körningar. Varje genererad XML validerades mot CB Open Interface-schemat innan den skickades in. Testningen omfattade både isolerade moduler och det kompletta tankesystemstestprojektet, och resultaten av dessa körningar redovisas i kapitel 5.

Testningen genomfördes i flera nivåer för stegvis verifiera verktygskedjan. Först testades parsern isolerat genom att enskilda SattLine-filer lästes in och kontrollerades mot förväntade RECORD-definitioner, moduldefinitioner, parametrar, variabler, submoduler och kodblock. Därefter testades XML-genereringen genom att de extraherade objekten serialiserades till CB Open Interface-format och granskades manuellt med avseende på namn, typpreferenser, komponenter och struktur. I nästa steg validerades XML-filerna mot CB Open Interface-schemat innan import. Valideringen användes som en kontrollpunkt för att upptäcka felaktig XML-struktur, saknade attribut eller objekt som inte kunde tolkas av Control Builder. Slutligen testades importflödet genom att XML-filerna skickades till ett öppet 800xA-projekt via CB Open Interface. Testerna omfattade både enskilda datatyper och mer sammansatta modultyper för att bedöma hur långt den automatiserade kedjan fungerade.

Testningen kan sammanfattas enligt Tabell 3.1.

Tabell 3.1: Teststeg för verifiering av verktygskedjan.

| Teststeg                          | Syfte                                                                                                   | Verifiering                                                                                         |
|-----------------------------------|---------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------|
| 1) Parser - RECORD                | Kontrollera att SattLine-datatyper identifieras korrekt                                                 | Förväntade datatyper jämfördes med källfilen                                                        |
| 2) Parser - MODULEDEFINITION      | Kontrollera att moduler samt tillhörande parametrar, variabler, kod och submoduler identifieras korrekt | Förväntade moduler jämfördes med källfilen                                                          |
| 3) XML- generering och validering | Säkerställa att interna datastrukturer omformas till korrekt CB Open Interface XML schemat              | XML-filer granskades manuellt för att säkerställa att de följde önskad XML-schemat.                 |
| 5) Import till 800xA              | Kontrollera att objekt kan skapas i Control Builder                                                     | Objekt verifierades i Control Builder genom att deras struktur och innehåll jämfördes med källfilen |
| 6) Kompletta arbetsflöde          | Kontrollera hela kedjan från SattLine-fil till importerat objekt                                        | GUI/CLI användes för att genomföra hela flödet steg för steg                                        |

### 3.6 Källkritik

[1] - Denna källa är Granitor Systems egen webbplats och används för att beskriva uppdragsgivaren. Eftersom informationen gäller företagets egen verksamhet, bransch och organisation kan Granitor ses som en primärkälla för grundläggande företagsfakta. Samtidigt är det en företagshemsida och informationen kan därför vara formulerad på ett positivt sätt. Källan anses därför pålitlig för neutrala fakta om företaget, men inte som oberoende bedömning av företagets prestationer.

[2], [8], [10], [11], [14] - Dessa källor kommer också från ABB, men är mer övergripande webbsidor, datablad eller produktinriktade dokument. Eftersom de behandlar ABB:s egna system, historik, standardkopplingar, Satt-produkter, 800xA och livscykelhantering är de relevanta och i huvudsak pålitliga som bakgrundskällor. Däremot kan de ha ett visst leverantörs- och marknadsföringsperspektiv. Därför används de främst för allmän bakgrund och sammanhang, medan mer detaljerade tekniska påståenden bör stödjas av manualerna.

[3], [20] - Dessa källor är böcker respektive standardlitteratur som används för att ge ett mer generellt teoretiskt stöd till rapportens tekniska bakgrund och metodavsnitt. Källorna bedöms som trovärdiga för övergripande begrepp, struktur och metodstöd. Samtidigt är de inte skrivna specifikt för SattLine eller 800xA, vilket innebär att de främst används för generella resonemang och inte som stöd för systemspecifika tekniska detaljer.

[4], [5], [6], [7] - Dessa källor är officiella ABB-manualer för SattLine. Eftersom ABB är den leverantör som förvaltar SattLine-dokumentationen anses dessa källor vara mycket pålitliga för tekniska uppgifter om systemets språk, I/O, kommunikation, grafik och grundläggande uppbyggnad. Källorna är dock versionsbundna, vilket innebär att informationen främst gäller de produktversioner som manualerna behandlar.

[9] - Denna källa är en vetenskaplig artikel som behandlar livscykelperspektivet inom industriell automation. Källan är relevant eftersom den ger ett mer generellt och akademiskt stöd till resonemanget om att olika faser i ett automationssystem livslängd medför olika krav på drift, underhåll, vidareutveckling och förändringshantering. Artikeln bedöms därför som trovärdig för att motivera varför livscykelperspektivet är relevant i rapporten. Samtidigt behandlar artikeln inte specifikt ABB SattLine eller System 800xA, vilket innebär att den används som teoretiskt stöd för det övergripande livscykelresonemanget och inte som direkt källa för ABB:s produktspecifika livscykelklassificering.

[12], [13], [15] - Dessa källor är officiell ABB-dokumentation för System 800xA, AC 800M och Control Builder Open Interface. Då ABB utvecklar och tillhandahåller dessa system är deras tekniska manualer de mest relevanta källorna för funktioner, arkitektur, gränssnitt, importmetoder och systemkrav. Källorna bedöms därför som trovärdiga för produktnära tekniska fakta. Samtidigt är de leverantörsdokumentation och bör inte tolkas som oberoende utvärderingar av systemens fördelar eller praktiska projektutfall.

[16], [17] - Dessa källor är ABB:s egna fallstudier om genomförda uppgraderingar till System 800xA. De är relevanta eftersom de visar att migration från äldre system, däribland SattLine, har genomförts i verkliga industriella miljöer. Eftersom fallstudierna kommer från ABB finns

det dock ett tydligt intresse av att framhäva positiva resultat. De anses därför trovärdiga som exempel på att sådana projekt har genomförts, men inte som oberoende bevis för att alla migrationer är enkla, riskfria eller direkt jämförbara med examensarbetets metod.

[18] - Denna källa kommer från Batenburg Industrial Automation och beskriver ett konkret projekt där ABB SattLine uppgraderats till ABB System 800xA hos Olam Cocoa. Källan är relevant eftersom Batenburg är en systemintegratör inom industriell automation och sidan beskriver ett praktiskt migrationsprojekt. Samtidigt är det en företagsreferens med kommersiellt syfte, vilket innebär att den kan vara selektiv och positivt vinklad. Den används därför som exempel på ett genomfört industriprojekt, inte som en oberoende teknisk analys.

[19] - Denna källa är ett examensarbete från Lunds universitet, LTH, inom industriell elektroteknik och automation. Källan är relevant eftersom den behandlar modernisering av ett äldre styrsystem till ABB 800xA i en verklig industriell miljö. Eftersom arbetet är publicerat via LUP Student Papers och har genomförts inom en akademisk utbildningsmiljö bedöms det vara mer akademiskt förankrat än företagskällor och webbartiklar. Samtidigt är det ett examensarbete och inte en vetenskapligt peer-reviewad artikel. Dessutom behandlar det Uniview och inte SattLine, vilket innebär att det bör användas som jämförande exempel snarare än som direkt bevis för SattLine-konvertering.

[21] - Denna källa är en allmän webbkälla som används för en grundläggande begreppsförklaring av P&ID. Creately tillhandahåller pedagogiskt material om diagram och dokumentation, vilket gör källan användbar för en översiktlig introduktion till ämnet. Samtidigt är det inte en vetenskapligt granskad källa eller en officiell standard. Den bör därför läsas kritiskt och endast användas för enklare bakgrundsförklaringar, inte för centrala tekniska slutsatser.

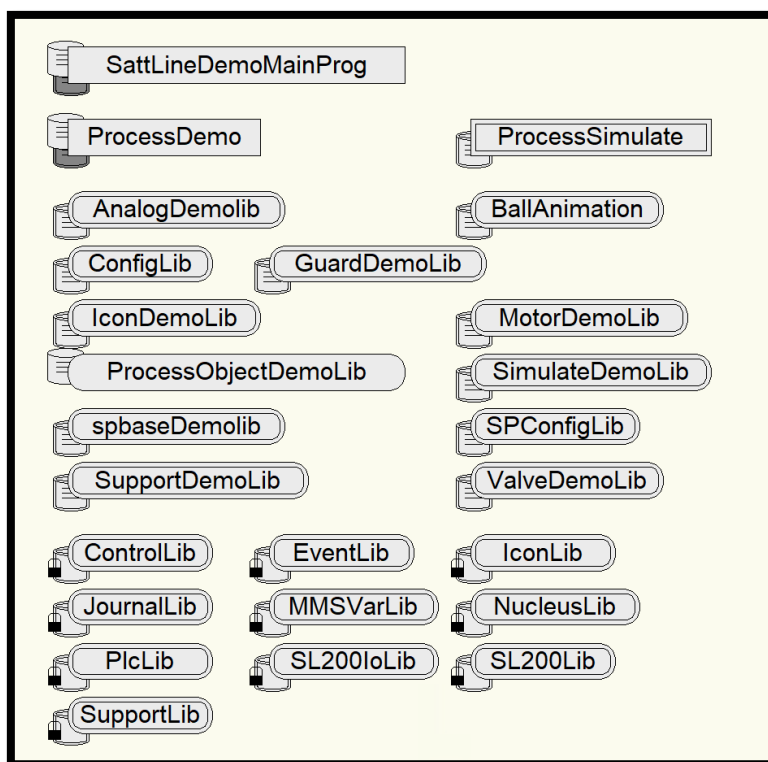
[22] - Denna källa kommer från Sveriges Ingenjörer, som är en etablerad professionsorganisation för ingenjörer i Sverige. Källan används för yrkesetiska resonemang och är relevant eftersom den beskriver ingenjörens ansvar, saklighet och professionella omdöme. Den bedöms som trovärdig för etiska principer inom svensk ingenjörskontext. Däremot är den inte en teknisk källa och används därför inte som stöd för tekniska påståenden.

## 4 Analys

Detta kapitel redovisar vad analysen av SattLine och 800xA resulterade i, och hur en konvertering mellan systemen bör ske. Avsnittet är uppdelat i en analys av SattLines interna struktur och textfilsformat, en jämförande analys av 800xA med tillhörande mappning av centrala koncept mot SattLine, samt en analys av hur konverteringen kan realiseras via XML och CB Open Interface.

### 4.1 Analys av SattLine

SattLines applikationsstruktur är strikt hierarkisk och objektorienterad (jfr. översikten i avsnitt 2.2.2). Toppnivån kallas *configuration* och innehåller ett antal programenheter se Figur 4.1. En programenhet är antingen ett exekverbart *Program*, som kompileras och laddas ned till en controller, eller ett *bibliotek* som innehåller återanvändbara modultyper och datatyper.



Figur 4.1: Konfiguration-flik i SattLine.

All kontrollogik och grafik kapslas in i moduler. Moduler struktureras hierarkiskt i godtyckligt antal nivåer; en modultyp definierar beteende, parametrar och grafik och kan instansieras ett valfritt antal gånger som modulinstanser. Exekveringen styrs av *ScanGroups*, som definierar cykeltid och prioritet för modulernas beräkningar. En modul ärver som standard sin omgivande moduls *ScanGroup*, men kan kopplas till en specifik grupp.

### 4.1.1 SattLines textfilformat och bibliotekens struktur

Filerna som genereras av SattLine i *Pretty Print*-läge (se avsnitt 3.2) är blockstrukturerade och hierarkiska, vilket direkt speglar modulernas logiska uppbyggnad. Varje fil inleds med ett filhuvud innehållande versionsdeklaration, tidsstämplar och en andringslogg. Hela fil innehålllet är insvept i en *BasePicture*-invocation, som alltid utgör det yttersta lagret:

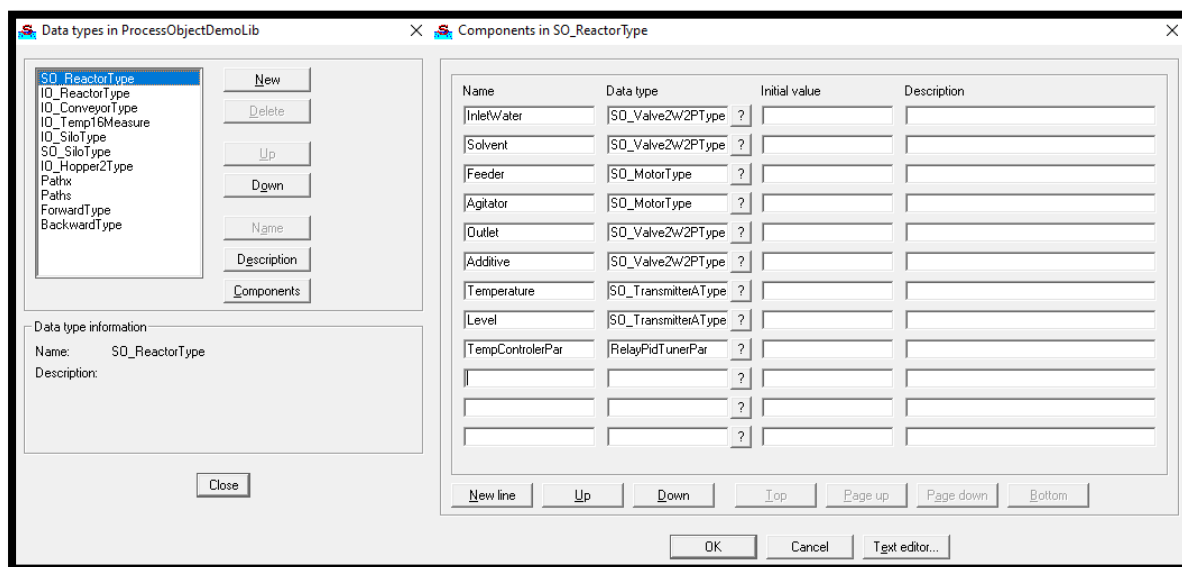
```
"Syntax version 2.23, date: YYYY-MM-DD-HH:MM:SS.mmm N"
"Original file date: YYYY-MM-DD-HH:MM:SS.mmm"
"Program date: YYYY-MM-DD-HH:MM:SS.mmm, name: LibraryName"
(* andringslogg med datum, signatur och kommentar *)
BasePicture
  Invocation
    ( x, y, z, w, h
      IgnoreMaxModule ) : MODULEDEFINITION DateCode_ nnnnnnn
    ( GroupConn = expr ) *
TYPEDEFINITIONS
...
```

- *GroupConn*: valfri, kopplar rotnoden till en ScanGroup.
- TYPEDEFINITIONS förekommer som nyckelord flera gånger i samma fil, oftast två gånger; första blocket innehåller vanligtvis **typdeklARATIONER** och andra blocket **moduler**.

Filen avslutas med:

```
ENDDEF (*BasePicture*);
```

### 4.1.2 TypdeklARATION - RECORD



Figur 4.2: Tillgängliga datatyper i ProcessObjectLib samt componenterna i SO\_ReactorType datatyp

Sammanfattade datatyper i SattLine deklarerar som RECORD-typer se figur 4.2. En RECORD-definition i textfilen följer monstret:

```
TypeName ["kommentar"*] = RECORD DateCode_ nnnnnnn
  [namn] ["kommentar"*] : [datatyp] := [initialvärde*];
ENDDEF
(*TypeName*);
```

Asterisk (\*) indikerar valfria fält; en typ behöver inte ha kommentar eller initialvärde. Varje fält består av ett namn eller flera kommaseparerade namn, en valfri beskrivningssträng och en datatyp, med möjlighet att ange ett initialvärde. Beskrivningssträngen kan innehålla riktningsprefix som "IN ...", "OUT ..." eller "IN/OUT ...", vilka används som dokumentation men inte påverkar typens semantik.

### 4.1.3 Moduldeklaration – MODULEDEFINITION

En modul i SattLine utgör en samlad enhet av parametrar, interna variabler, submoduler, kodlogik och presentationsrelaterade objekt. Varje modul inleds med en *MODULEDEFINITION*-sektion som innehåller metadata såsom intern tidsstämpel, eventuell direkt koppling till en ScanGroup via *GroupConn* samt ifall den är en *Frame Module*. Modulnamnet förekommer som ett fristående namn före *MODULEDEFINITION*, som en sektionsrubrik, och blocket avslutas med *ENDDEF*:

```
[Module namn]
(* *)
= MODULEDEFINITION DateCode_ [num] (GroupConn = expr | Frame_Module)*
MODULEPARAMETERS
  ...
LOCALVARIABLES
  ...
SUBMODULES
  ...
ModuleDef
  [ClippingBounds = (x1, y1) (x2, y2)]
  [ZoomLimits = max min]
  [Two_Layers_LayerLimit_ = value]
  [GraphObjects : ...]
  [InteractObjects : ...]
[ModuleCode ...]
ENDEF (* Module namn *)
```

Kodexemplet visar hur modulens delar ligger ordnade i textfilen. *MODULEPARAMETERS* beskriver modulens externa gränssnitt och behandlas i avsnitt 4.1.5. *LOCALVARIABLES* beskriver modulens interna tillstånd och behandlas i avsnitt 4.1.6. *SUBMODULES* visar hur modulen byggs upp hierarkiskt av underliggande moduler och behandlas i avsnitt 4.1.7. *ModuleCode* innehåller modulens exekverbara logik och behandlas i avsnitt 4.1.8.

Blocket *ModuleDef* innehåller presentationsrelaterad information. Dit hör exempelvis *ClippingBounds*, *ZoomLimits*, *GraphObjects* och *InteractObjects*. Dessa delar beskriver grafisk presentation, dynamiska grafikkopplingar och operatörshändelser. De är viktiga för att förstå hur modulen visas i SattLine, men de har inte samma direkta betydelse för logikkonverteringen som parametrar, variabler, submoduler och kodblock.

## 4.1.4 Modultyper i SattLine

SattLine skiljer mellan flera typer av moduler. Denna klassificering beskriver hur en modul är tänkt att användas.

- En **Module Type** är avsedd för återanvändning. Den får ett eget namn och kan användas som mall för flera modulinstanser i en programenhet eller i ett bibliotek. Instanserna delar samma typlogik, men kan ha egna parameterkopplingar och egna variabelvärden.
- En **Single Module** är i stället avsedd att användas en gång. Den kan gruppera grafiska objekt, variabler, parametrar och andra moduler i en sammanhållen enhet, men är inte tänkt att återanvändas på samma sätt som en namngiven modultyp.
- En **Frame Module** fungerar främst som en strukturell eller grafisk ram. Den saknar eget definitionsnamn och ärver variabler och parametrar från den omslutande icke-rammodulen. Den används därför framför allt för att gruppera presentationsobjekt eller programdelar utan att behöva skapa ett separat gränssnitt med egna parametrar.

## 4.1.5 MODULEPARAMETERS.

Parametrar anger modulens externa gränssnitt se figur 4.3. Varje parameter deklareraras med namn, valfri kommentar, datatyp, eventuellt standardvärde och eventuellt startattribut. Syntaxen följer i huvudsak följande mönster:

```
[Name] [Description*] : [Data type] := [Default value*] [Startattribut*];
```

*Name* parameter från *Reactor* parametrar som visas i figur 4.3 kan se ut på följande sätt:

```
MODULEPARAMETERS
  Name "In Module name": string := "Module name";
  ...
```

| Name            | Data type           | Default value | Description      |
|-----------------|---------------------|---------------|------------------|
| Name            | string              | ?             | "Module name"    |
| IO              | IO_ReactorType      | ?             | IN/OUT IO        |
| SO              | SO_ReactorType      | ?             | Default          |
| EventClass      | integer             | ?             | IN Part of Plant |
| EventText       | TextEventDataType   | ?             |                  |
| HistoryGroup    | HistoryGroupType    | ?             |                  |
| HistoryControl  | HistoryControlType  | ?             |                  |
| DemoProgramInfo | DemoProgramInfoType | ?             | IN for Demo      |
|                 |                     | ?             |                  |
|                 |                     | ?             |                  |
|                 |                     | ?             |                  |
|                 |                     | ?             |                  |

New line Up Down Top Page up Page down Bottom OK Cancel Text editor...

Figur 4.3: Reactor parametrar vyn i SattLine

Ibland kan det förekomma ett prefix i *[default value]*, ett exempel på ett prefix är *Global* som indikerar att variabeln refererar till ett globalt objekt utanför den aktuella modulens räckvidd. Ett så kallat *startattribut* kan anges efter initialvärdet och styr hur parameterens värde hanteras vid omstart och programändringar. Tabellen 4.1 visar tillgängliga startattribut.

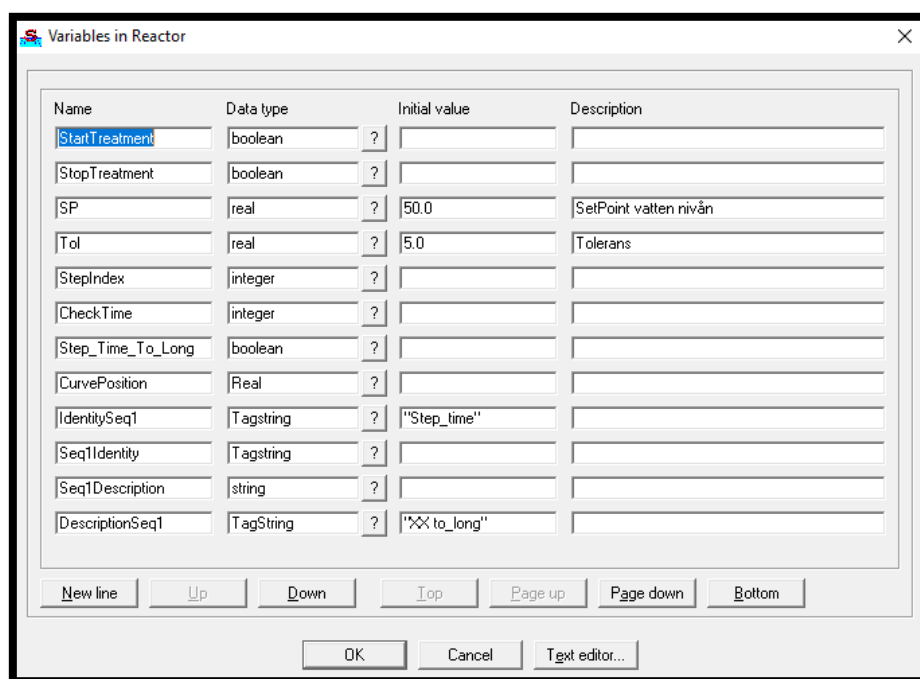
Tabell 4.1: Startattribut för variabler i SattLine och deras funktion vid omstart och exekvering.

| Attribut             | Kommentarer                                                                                                                                                          |
|----------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b><i>State</i></b>  | Används när variabeln ska bära med sig sitt tidigare värde mellan exekveringscykler, vilket innebär att systemet skapar separata gamla och nya värden för variabeln. |
| <b><i>Const</i></b>  | Används för variabler som alltid ska behålla sitt initialvärde                                                                                                       |
| <b><i>OpSave</i></b> | Används för operatörsjusterade värden som ska sparas och återställas vid omstart                                                                                     |
| <b><i>Secure</i></b> | Används för kritiska processvärden som också ska bevaras vid omstart.                                                                                                |

## 4.1.6 LOCALVARIABLES

*LOCALVARIABLES* beskriver modulens interna tillstånd. Till skillnad från *MODULEPARAMETERS* är dessa variabler inte synliga som en del av modulens externa gränssnitt, utan används internt av modulens kod och submoduler.

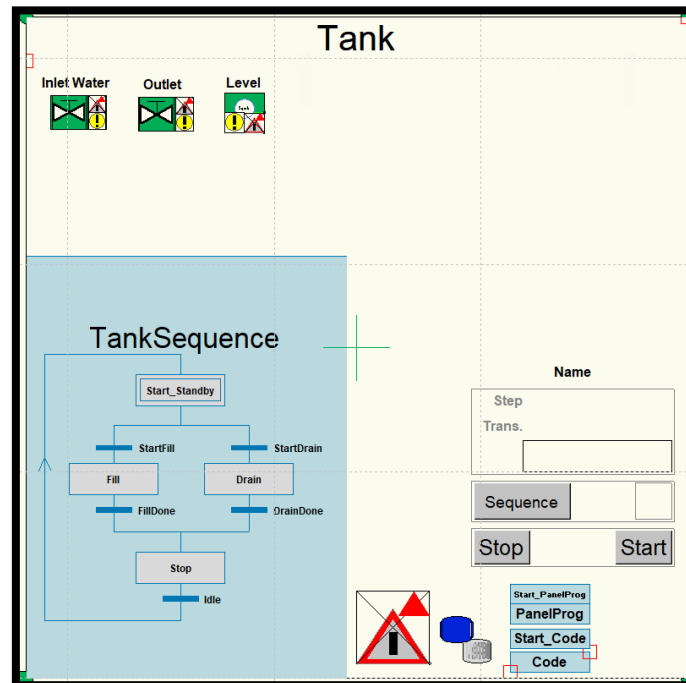
Syntaxen är i stort sett densamma som för *MODULEPARAMETERS*. Skillnaden är att värdet fungerar som ett initialvärde, *Initial value* i figur 4.4, för modulens interna variabel i stället för som ett standardvärde för en extern parameter. Därför är *LOCALVARIABLES* viktigt vid konvertering, eftersom dessa variabler behöver följa med som modulens interna data även om de inte exponeras utåt.



Figur 4.4: Reactor variabler vyn i SattLine

## 4.1.7 SUBMODULES

*SUBMODULES* beskriver modulens hierarkiska uppbyggnad. Submoduler i SattLine är alltid inkapslade inom en omgivande modul, se figur 4.5. Vilket innebär att den omgivande modulen äger submodulens livscykel, parameterkopplingar och placering i modulstrukturen.



Figur 4.5: Tank modul samt ventil-, sekvens-, kod-, faceplace- och ikonsubmoduler.

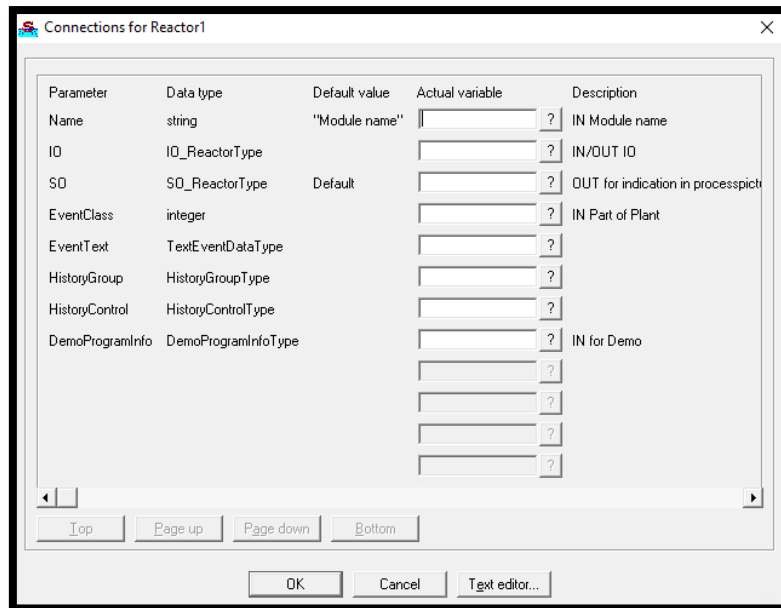
I SattLines textformat kan submoduler instansieras på två huvudsakliga sätt. Antingen skapas en instans av en redan namngiven modultyp eller så deklarerar submodulen direkt i *SUBMODULES* alltså som single module.

### Instans av namngiven modultyp

När submodulen är en instans av en namngiven modultyp anges först instansens namn och grafiska egenskaper, följt av vilken modultyp som ska användas. Därefter följer en lista över parameterkopplingarna, benämnd Connections.

```
SUBMODULES
[namn] Invocation
  (x , y , rotation , bredd , höjd
   övriga params) : [Modultyp] (
[Parameter1] => Global* [Kopplat värde/variabel]
[Parameter2] => [Kopplat värde/variabel]
... )
```

I connections-listan i figur 4.6 kopplas varje parameter i modultypen till en källa i den omgivande modulen. Källan kan exempelvis vara en lokal variabel, ett fält i en strukturerad IO- eller SO-record, eller en global variabel. Den omgivande modulen äger därmed de värden som skickas in i submodulen, exempelvis identiteter, IO-signaler och händelseklassning.



Figur 4.6: Connections lista för Reactor 1

## Lokal moduldefinition i SUBMODULES

En submodul kan även definieras direkt i SUBMODULES genom en MODULEDEFINITION, Då definieras som en icke återanvändbar modul som bara existerar inom den omgivande modulen.

```

SUBMODULES
  SubName Invocation
    ( x, y, z, w, h
      [övriga attribut]
    ) : MODULEDEFINITION DateCode_ [num] [(Frame_Module)]
    [SUBMODULES
      ...
    ]
  ModuleDef
    [grafik och attribut]
    [ModuleCode
      ...
    ]
  ENDDDEF (*SubName*);

```

Denna form används när modulens struktur eller presentation behöver kapslas in lokalt. I fallet där submodulen också är en Frame Module, då fungerar sen som en grafisk eller strukturell ram runt andra objekt. Den återanvänder den omgivande modulens variabler och parametrar och kan användas för att gruppera grafiska objekt eller programdelar, till exempel för att visa eller dölja en grupp satser.

Den grafiska representationen är inbäddad direkt i modulfilerna under blocken *GraphObjects* (geometri och dynamiska kopplingar till variabler) och *InteractObjects* (operatorshandelser kopplade till booleska variabler). Dessa delar, liksom visuella/zoom-relaterade attribut: *ClippingBounds*, *ZoomLimits*, *Two\_Layers\_LayerLimit\_*, *Zoomable*, *Grid*, hör till SattLines presentationslogik och har ingen direkt motsvarighet i den logik som ska migreras, de tas därför inte med i konverteringen.

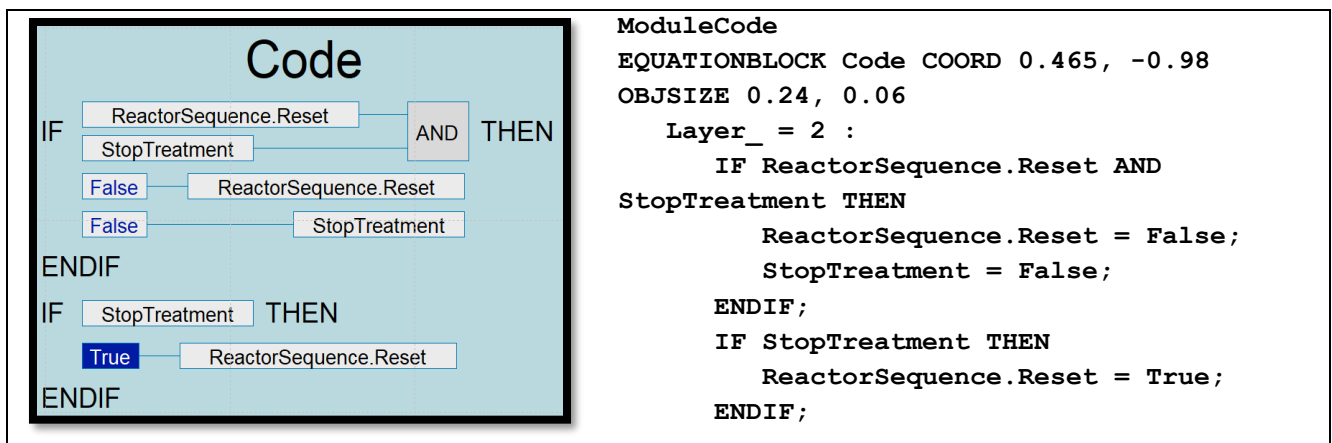
### 4.1.8 ModuleCode

*ModuleCode* är den del av en SattLine-modul där den exekverbara logiken samlas. Avsnittet är centralt för konverteringen eftersom det är här modulens beteende uttrycks. I textformatet delas kodlogiken främst upp i *EQUATIONBLOCK* och *SEQUENCE* eller *OPENSEQUENCE*.

#### Equationblock

Ett *EQUATIONBLOCK* är ett kodblock som exekveras kontinuerligt varje scancykel, ett exempel på det finns i figur 4.8. Det finns även en särskild variant, *StartEquation*, där blocknamnet börjar med *Start*. Ett sådant block exekveras endast en gång vid programstart eller nedladdning, exempelvis *Start\_Code*. Equationblock följer mönster:

```
ModuleCode
EQUATIONBLOCK [blocknamn] COORD x, y OBJSIZE m, n
    [övriga grafiska parameter]:
    [KOD]
EQUATIONBLOCK [...]
```



Figur 4.7: Kod *EQUATIONBLOCK* i SattLine med kodrepresentation i textfilen

Koden i ett ekvationsblock liknar syntaktiskt Structured Text. Den kan innehålla booleska uttryck med *AND*, *OR* och *NOT*, aritmetiska operatorer som +, −, \* och /, jämförelser som =, <>, >, ≥, < och ≤, samt *IF*-uttryck.

## SEQUENCE och OPENSEQUENCE

Sekvensblock är SattLines grafiska verktyg för processkontroll och motsvarar i huvudsak SFC i IEC-standardens och ett exempel på det finns i figur 4.9. En *SEQUENCE* är ett komplett sekvensblock med en inbyggd ovillkorlig loop där sista steget återaktiverar initialsteget. En *OPENSEQUENCE* är i stället ett sekvensfragment som kopplas till resten av sekvensen genom förgreningar eller hopp.

Ett sekvenssteg, *SEQSTEP*, kan innehålla tre kodsektioner. *ENTERCODE* körs en gång när steget aktiveras, *ACTIVECODE* körs kontinuerligt medan steget är aktivt och *EXITCODE* körs en gång när steget deaktiveras.

## Strukturerade förgreningar

SattLine använder särskilda konstruktioner för att beskriva alternativa och parallella sekvensflöden. En alternativ förgrening representeras av *ALTERNATIVESEQ* och *ALTERNATIVEBRANCH*. Den används när endast en av flera möjliga grenar ska väljas. Varje gren börjar och slutar med en *SEQTRANSITION*, och om flera övergångsvillkor är sanna väljs den vänstra grenen först.

En parallell förgrening representeras av *PARALLELSEQ* och *PARALLELBRANCH*. Den används när flera grenar ska köras samtidigt och oberoende av varandra. Varje parallell gren börjar och slutar med ett *SEQSTEP*, och den efterföljande transitionen aktiveras först när samtliga grenars sista steg är aktiva. Skillnaden mellan dessa två förgreningstyper är viktig vid konvertering, eftersom en alternativ gren beskriver ett val medan en parallell gren beskriver samtidighet och synkronisering.

## Sekvenshopp

Sekvensflödet kan också styras med hopp. Ett hopp kan placeras efter ett steg eller efter en transition och anges med *SEQFORK* följt av destinationsnamn och *SEQBREAK*.

Hopp efter ett steg:

```
SEQSTEP [steg namn]
  ENTERCODE
  ...
  ACTIVECODE
  ...
  SEQFORK [destinations namn] SEQBREAK
```

Hopp efter transition:

```
SEQTRANSITION [transitionnamn] [villkor]
  SEQFORK [destinations namn] SEQBREAK
```

## Sammanfattande sekvensmönster

Ett sekvensblock kan sammanfattas med följande övergripande mönster i SattLines textformat:

```
[OPENSEQUENCE / SEQUENCE] [blocknamn] (SeqControl*, SeqTimer*) COORD [x,y]
OBJSIZE [x,y]

    SEQINITSTEP [stegnamn]      (* Initialsteg - aktiveras vid start *)
        ENTERCODE              (* Körs EN gång när steget aktiveras *)
        ...
        ACTIVECODE             (* Körs KONTINUERLIGT medan steget är aktivt *)
        ...
        EXITCODE*              (* Körs EN gång när steget deaktiveras *)
        ...

    ALTERNATIVESEQ              (* Alternativgren; väljer EN av flera grenar *)
        SEQTRANSITION [namn] WAIT_FOR [villkor]
        SEQSTEP [stegnamn]
            ENTERCODE
            ...
            ACTIVECODE

        SEQTRANSITION [namn] WAIT_FOR [villkor]
    ALTERNATIVEBRANCH
        SEQTRANSITION [namn] WAIT_FOR [villkor]
        SEQSTEP [stegnamn]
            ENTERCODE
            ...
            ACTIVECODE

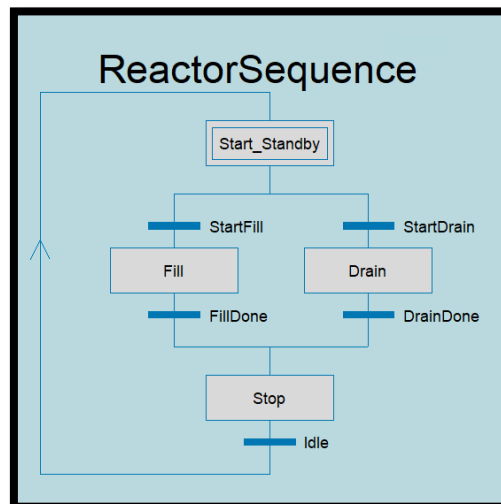
        SEQTRANSITION [namn] WAIT_FOR [villkor]
    ENDALTERNATIVE

    PARALLELSEQ                (* Parallellgren; kör FLERA grenar samtidigt *)
        SEQSTEP [stegnamn]
    PARALLELBRANCH
        SEQSTEP [stegnamn]
    ENDPARALLEL

    SEQTRANSITION [namn] WAIT_FOR [villkor]
    SEQSTEP Stop
        ENTERCODE
        ...
        ACTIVECODE

    SEQTRANSITION [namn] WAIT_FOR [villkor]
[ENDOPENSEQUENCE / ENDSEQUENCE]

ENDDEF (*blocknamn*);
```



```

SEQUENCE ReactorSequence (SeqControl,SeqTimer) COORD -0.755001, -1.01
  OBJSIZE 1.525, 2.01
  SEQINITSTEP Start_Standby
    ENTERCODE
      SO.InletWater.OrderOpen = False;
      SO.InletWater.OrderClose = True;
      SO.Outlet.OrderOpen = False;
      SO.Outlet.OrderClose = True;
    ACTIVECODE

  ALTERNATIVESEQ
    SEQTRANSITION StartFill WAIT_FOR StartTreatment AND NOT
      StopTreatment AND IO.Level.Value < SP
    SEQSTEP Fill
      ENTERCODE
        SO.InletWater.OrderOpen = True;
        SO.InletWater.OrderClose = False;
        SO.Outlet.OrderOpen = False;
        SO.Outlet.OrderClose = True;
      SEQTRANSITION FillDone WAIT_FOR IO.Level.Value >= SP
    ALTERNATIVEBRANCH
      SEQTRANSITION StartDrain WAIT_FOR StartTreatment AND NOT
        StopTreatment AND IO.Level.Value > SP
      SEQSTEP Drain
        ENTERCODE
          SO.InletWater.OrderOpen = False;
          SO.InletWater.OrderClose = True;
          SO.Outlet.OrderOpen = True;
          SO.Outlet.OrderClose = False;
        ACTIVECODE

      SEQTRANSITION DrainDone WAIT_FOR IO.Level.Value <= SP
    ENDALTERNATIVE
  SEQSTEP Stop
    ENTERCODE
      SO.InletWater.OrderOpen = False;
      SO.InletWater.OrderClose = True;
      SO.Outlet.OrderOpen = False;
      SO.Outlet.OrderClose = True;
    ACTIVECODE

  SEQTRANSITION Idle WAIT_FOR IO.Level.Value > SP + Tol OR IO.Level.
    Value < SP - Tol
  ENDSEQUENCE

```

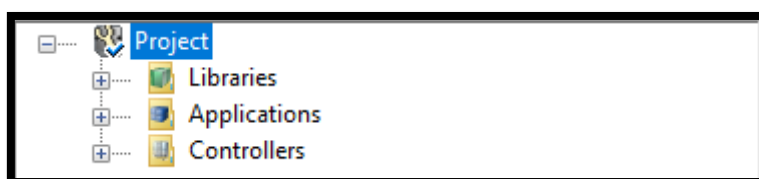
Figur 4.8: ReactorSequence SEQUENCE i SattLine med kodrepresentation i textfilen

## 4.2 Analys av 800xA med SattLine-mappning

SattLine och 800xA systemen bygger på en hierarkisk, objektorienterad struktur, men 800xA introducerar en tydligare separation mellan projekt, applikationer och styrenheter (jfr. översikten i avsnitt 2.3 och 2.4).

### 4.2.1 Systemhierarki och övergripande arkitektur

I 800xA organiseras projektet i tre huvudmappar: *Libraries* (standardbibliotek + användardefinierade), *Applications* (all exekverbar kod) och *Controllers* (hårdvarukonfiguration och tasks).



Figur 4.9: 800xA Project Explorer Interface

Figur 4.10 visar den övergripande projektstrukturen i 800xA. För att tydliggöra hur denna struktur kan relateras till motsvarande enheter i SattLine presenteras en sammanställning i tabell 4.2. Tabellen visar centrala mappningar mellan SattLines systemhierarki och 800xA:s projektstruktur.

Tabell 4.2: Mappning mellan SattLines systemhierarki och 800xA:s projektstruktur

| SattLine               | 800xA        | Kommentarer                                                                                                                                                                                                                                                                                                          |
|------------------------|--------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Configuration          | Project      | Toppnivåbehållare; i 800xA hanteras hela konfigurationen inom ett Project.                                                                                                                                                                                                                                           |
| Program Unit (Program) | Applications | Exekverbar enhet som kompileras och laddas ned till en styrenhet; en 800xA Application kan innehålla Diagrams, Programs, Control Modules                                                                                                                                                                             |
| Program Unit (Library) | Libraries    | Återanvändbara typer; i 800xA skapas bibliotek under Libraries-mappen med Data Types, Function Block Types, Control Module Types och Diagram Types                                                                                                                                                                   |
| Program Unit           | Controllers  | I 800xA mappen Controllers innehåller den eller de fysiska styrenheter som används i projektet. I SattLine motsvaras detta inte av en separat controllers-mapp, utan av en programenhet som är knuten till ett kontrollsystem. Kontrollsystemet nås genom att öppna programenheten och zooma in i dess modulkierarki |

## 4.2.2 Bibliotek

Båda systemens standardbibliotek täcker samma övergripande funktionella domäner, men deras innehåll och uppbyggnad skiljer sig åt, vilket sammanfattas i tabell 4.3:

Tabell 4.3: Funktionell kategorisering av SattLine och 800xA System Bibliotek

| Funktionskategori         | SattLine                                         | 800xA                                            |
|---------------------------|--------------------------------------------------|--------------------------------------------------|
| Grundfunktioner           | ConfigLib, NucleusLib, IconLib, SupportLib       | Basic Library, Icon Library                      |
| I/O-Koppling              | SL800Lib, SL200Lib, PioRackLib, m.fl.            | Communication Libraries, hårdvaruspecifika typer |
| Larm och händelser        | EventLib                                         | Alarm och Event Library, Signal Library          |
| Digital/supervisory logik | PlcLib, TccLib                                   | Process Object Libraries, Supervision Library    |
| Reglering                 | ControlLib, FuzzyLib                             | Control Libraries (PID m.m.)                     |
| Sekvensstyrning           | NucleusLib (ScanGroup) + sekvensblock            | Sequence Start Library, Group Start Library      |
| Datainsamling             | JournalLib, SQLLib, ReportLib                    | 800xA History / IM Server                        |
| Kommunikation             | MMSVarLib, PBLib, Comli1–5Lib, Fieldbusvar1–2Lib | Communication Libraries (Profibus, HART, FF)     |
| Batch                     | BatchLib                                         | Batch Library                                    |
| VT100/terminal            | FormsLib                                         | Saknas                                           |

En central skillnad är hur de två systemen hanterar programexekvering och schemaläggning. I SattLine definieras exekveringsbeteendet (cykeltid, prioritet, fas och systemidentitet) direkt i programmet via *NucleusLib*-modulerna *ScanGroupControl* och *ScanGroupExtension*. I System 800xA är detta istället en konfigurationsparameter på kontrollersnivå via AC 800M:s Task-system i Control Builder, inte ett bibliotek som ingår i applikationskoden. Detta innebär en strukturell skillnad som påverkar hur ett SattLine-program behöver tolkas och representeras i 800xA.

En annan skillnad gäller datainsamling och rapportering. SattLine hanterar journalföring, SQL-kommunikation och rapportgenerering via kontrollib-moduler (*JournalLib*, *SQLLib*, *ReportLib*). I System 800xA är dessa funktioner separerade till en dedikerad informationshanteringsserver (IM Server / Process Data), vilket innebär att de inte har direkta motsvarigheter på kontrollibnivå.

Vad gäller I/O-hantering är SattLine:s bibliotek nära kopplade till specifik hårdvara (SL200, SL800, PIO-rack), medan 800xA:s hårdvarutyper är mer abstraherade och konfigureras via ett separat hårdvarulager i Control Builder. Principen är densamma att koppla en programvariabel till en fysisk signal men abstraktionsnivå och konfigurationsmetod skiljer sig.

### 4.2.3 Datatyper

SattLine använder ett begränsat antal inbyggda enkla datatyper. Som framgår av tabell 4.4 mappas de grundläggande till motsvarande datatyper i 800xA.

Tabell 4.4: Mappning mellan SattLines enkla datatyper och motsvarande datatyper i 800xA.

| SattLine    | 800xA       | Kommentarer                                                                                                                                                                                               |
|-------------|-------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| boolean     | bool        | Direkt mappning.                                                                                                                                                                                          |
| integer     | dint        | SattLines integer är 32-bitars heltal; dint i 800xA är likvärdig (32 bitar, signerad). Observera att 800xA även har int (16 bitar) och uint (16 bitar, osignerad) som saknar direkt SattLine-motsvarighet |
| real        | real        | Direkt mappning; båda är 32-bitars                                                                                                                                                                        |
| String      | String      | Båda har samma längd, direkt mappning                                                                                                                                                                     |
| IdentString | String[10]  | Specificera längd explicit i 800xA                                                                                                                                                                        |
| TagString   | String[30]  | Specificera längd explicit i 800xA                                                                                                                                                                        |
| LineString  | String[80]  | Specificera längd explicit i 800xA                                                                                                                                                                        |
| MaxString   | String[140] | Maxlängden är 140 tecken i båda systemen                                                                                                                                                                  |

800xA:s string-typ har en standardlängd på 40 tecken om ingen längd anges, men kan deklarerar upp till 140 tecken med syntaxen *String[length]*. Eftersom SattLine har fem separata string-typer med fasta längder behöver varje string-variabel vid migrering analyseras individuellt och tilldelas rätt längd i 800xA-deklarationen.

#### Strukturerade systemdatatyper

Utöver de enkla datatyperna finns även strukturerade systemdatatyper, vilka jämförs med motsvarande lösningar i 800xA i tabell 4.5.

Tabell 4.5: Jämförelse av några av SattLines inbyggda strukturerade systemdatatyper och deras närmaste motsvarigheter.

| SattLine                                       | 800xA                | Kommentar                                                                 |
|------------------------------------------------|----------------------|---------------------------------------------------------------------------|
| Time                                           | date_and_time        | kräver constant-variabel i 800xA                                          |
| Duration                                       | time                 | time i 800xA motsvarar duration, inte absolut tid                         |
| TimeRecord                                     | CalendarStruct       | Båda bryter ner datum/tid i komponenter, olika fältnamn/struktur          |
| DurationRecord                                 | –                    | Får lösas med egen struct och beräkningar                                 |
| CalendarRecord                                 | CalendarStruct       | Veckonummer m.m. behöver verifieras mot aktuell 800xA-version             |
| Timer                                          | TON/TOF/TP-block     | Ingen fristående Timer-typ i 800xA, ersätts med IEC-timerblock            |
| OnOffTimerType                                 | TON/TOF/TP + logik   | Funktion ersätts med IEC-timerblock, Hold hanteras i logik                |
| BooleanSignal/IntegerSignal/AnalogSignalIn/Out | BoolIO/DintIO/RealIO | IO-bindning flyttas till Hardware Configuration, variabeln blir enkel typ |

## IO-signaltyper

SattLines *BooleanSignal*, *IntegerSignal* och *AnalogSignalIn/Out* definierade i *NucleusLib* kopplar en programvariabel direkt till ett fysiskt IO-kort och innehåller komponenter som *Value*, *IoValue* och *Forced*; analoga typer har även *Valid*, *Inverted* och skalningsparametrar via *SignalPar*. De representerar ett designmönster där IO-mappning och programlogik är tätt integrerade på datatypsnivå. I 800xA finns motsvarande IO-strukturtyper *BoolIO*, *DintIO/DwordIO* och *RealIO* med komponenter som *Value*, *IOValue*, *Forced* och *Status*, men IO-kopplingen konfigureras separat i Hardware Configuration: variabeln deklarerar med typ *BoolIO* och kopplas sedan till en fysisk IO-kanal i Hardware Editor. Applikationskoden använder sedan *Signal.Value*, vilket strukturellt liknar SattLines *BooleanSignal.Value*, men med skillnaden att diagnostik och skalning hanteras av systemet via *Status*-komponenten i stället för att vara inbyggd i datatypsdefinitionen.

### 4.2.4 Moduler och submoduler - mappning mot 800xA

Grundpelaren i SattLine är modulen: den primära struktureringsenheten som kan innehålla programkod, variabler, parametrar, grafik och submoduler. Som framgår av tabell 4.6 motsvaras SattLines moduler av flera olika objektstrukturer i 800xA beroende på modulens funktion, återanvändbarhet och hierarkiska uppbyggnad.

Tabell 4.6: Översiktlig mappning mellan SattLine-moduler och motsvarande struktur i 800xA.

| SattLine                                   | 800xA                              | Kommentarer                                                                                   |
|--------------------------------------------|------------------------------------|-----------------------------------------------------------------------------------------------|
| Module Type (med instanser)                | Control Module Type                | Återanvändbar typ                                                                             |
| Module Type (enkel logik, inga submoduler) | Function Block Type                | Återanvändbar beräknings/logikblock utan grafisk hierarki                                     |
| Single Module                              | Single Control Module              | Typ och instans är sammansatta i ett enda objekt; kan inte återanvändas som en fristående typ |
| Frame module                               | (ingen direkt motsvarighet)        | 800xA har inte samma typ av ”ärvda synlighet” för parametrar som rammodulerna i SattLine      |
| Submoduler (modulinstanser inuti modultyp) | Control Module instanser inuti CMT | CMT-instanser som placeras inuti andra CMT skapar en hierarkisk struktur.                     |

En viktig distinktion i 800xA är skillnaden mellan *FunctionBlockType* och *ControlModuleType*:

- En *FunctionBlockType* är ett återanvändbart beräkningsselement med parametrar, variabler och kodblock (ST, IL, FBD, ladder). Den saknar grafisk processbild och processinteraktion, huvudrollen består av ren beräknings- och styrologik.
- En *ControlModuleType* är en mer komplex enhet som kan innehålla submoduler (instanser av andra CMT:er och FBT:er), CodeBlocks, parametrar, variabler,

ExternalVariables, interaktionsfönster och grafisk layout i CMD-editorn (Control Module Diagram).

Vid mappning av SattLine-projekt till 800xA behöver man alltså ta ställning till vilken 800xA-typ varje SattLine-modul bör konverteras till. En grov tumregel är:

- Module Type med submoduler och processvisualisering → ControlModuleType
- Module Type med enbart beräkningslogik (PID, filter, timers) → FunctionBlockType
- Single Module → Single Control Module
- Frame Module → kan i de flesta fall absorberas i den omslutande ControlModuleType

## 4.2.5 Parametrar och variabler

SattLine och 800xA har likartade koncept för parametrar och variabler, men 800xA erbjuder fler typer och attribut. Centrala skillnader i hur parametrar deklarerar i de två systemen sammanfattas i tabell 4.7.

### Parameterdeklaration

En parameter i SattLine deklarerar inuti en modul och används för att koppla samman modulen med sin omgivning. Antingen till en variabel i den omslutande modulen, till ett fast värde, eller via grafisk koppling till ett annat objekt. Den har ett namn, en datatyp, ett eventuellt standardvärde och en beskrivning. Riktningen anges informellt i beskrivnings fältet; det finns inget riktningsattribut i deklarationssyntaxen. I 800xA har parametrar ett explicit *Direction*-attribut som är en formell del av typdeklarationen (*In*, *Out* eller *In\_Out*) och valideras vid kompilering. 800xA introducerar även *FD Port* (Function Diagram port), som styr om parametern ska exponeras som en grafisk kopplingsport; parametrar utan FD-port förblir logiskt tillgängliga men visas inte som portar i det grafiska flödesschemat.

Tabell 4.7: Mappning mellan parametrars attribut i SattLine och motsvarande fält i 800xA.

| SattLine                                                          | 800xA         | Kommentarer                                                                                    |
|-------------------------------------------------------------------|---------------|------------------------------------------------------------------------------------------------|
| Name                                                              | Name          | Direkt mappning.                                                                               |
| Data type                                                         | Data Type     | Se avsnitt 3.3.2 om datatyper.                                                                 |
| <i>(Riktning framgår oftast i beskrivningen, men inte alltid)</i> | Direction     | 800xA har formellt In/Out/In_Out; i SattLine anges riktningen informellt i description-fältet. |
| <i>(ej tillämpligt)</i>                                           | FD Port       | Avgör om parametern syns som grafisk port.                                                     |
| Default Value                                                     | Initial Value | Direkt mappning.                                                                               |
| Description                                                       | Description   | Direkt mappning.                                                                               |

Vid migrering innebär avsaknaden av formell riktning i SattLine ett tolkningssteg: IN/OUT-deklarerationer i beskrivnings fältet måste parsas och översättas till korrekt *Direction* i 800xA-deklarerationen.

## Variabeldeklaration

I SattLine deklareras variabler på medelnivå och är tillgängliga för alla ekvationsblock och sekvenser inom samma modul. En variabel har namn, datatype och eventuellt initialvärde, samt attributen *State*, *Const*, *OpSave* och *Secure*. I 800xA hanteras motsvarande funktionalitet via två separata mekanismer: *Start Attribute* och *Attributes*. Start Attribute styr hur värdet hanteras vid omstart: *coldretain* bevarar värdet vid både varm och kall omstart (närmast SattLines *Secure*), medan *retain* enbart bevarar värdet vid varm omstart. Attributet *constant* hindrar operatören från att ändra värdet online, vilket motsvarar SattLines *Const*. En variabel utan *Start Attribute* återgår till sitt initialvärde vid omstart. Attribut-fältet styr i stället beteenden under körning: *state* ger variabeln ett gammalt och ett nytt värde mellan scan-cykler, precis som SattLines *State*-attribut. Övriga attribut som *hidden*, *nosort*, *signalin* och *signalout* saknar direkta SattLine-motsvarigheter. Motsvarande skillnader mellan variabeldeklaration i SattLine och attributhantering i 800xA sammanfattas i tabell 4.8.

Tabell 4.8: Jämförelse mellan variabeldeklaration i SattLine och motsvarande attributhantering i 800xA.

| SattLine                                            | 800xA           | Kommentarer                    |
|-----------------------------------------------------|-----------------|--------------------------------|
| Name                                                | Name            | Direkt mappning.               |
| Data type                                           | Data Type       | Se avsnitt 3.3.2 om datatyper. |
| Initial Value                                       | Initial Value   | Direkt mappning.               |
| Description                                         | Description     | Direkt mappning.               |
| Vid variabel deklaration<br>(:= [namn] [Attribute]) | Start Attribute |                                |
| Vid variabel deklaration<br>(:= [namn] [Attribute]) | Attribute       |                                |

En viktig skillnad är att variabler i 800xA som ingår i en Control Module automatiskt kan göras tillgängliga för trendvisning, larmsättning och faceplate-visning via Aspect Object-systemet, något som saknar direkt motsvarighet i SattLines variabeldeklaration.

### 4.2.6 Kodlogik och programspråk

SattLines programspråk är ett proprietärspråk som är syntaktiskt likt *Structured Text (ST)* enligt IEC 61131-3, men med vissa avvikelser som måste hanteras vid migrering. Programkoden i SattLine är organiserad i två typer av programblock: *Equation Blocks* och *Sequence Blocks*, vilka i 800xA motsvaras av ST-sektioner respektive SFC-diagram placerade inuti en *Control Module Typ* eller *Function Block Typ*.

#### ***Equation Block* → *Structured Text (ST)***

Ett Equation Block i SattLine består av en lista satser som exekveras cykliskt och kan innehålla tilldelningar, villkor, proceduranrop och kommentarer. Strukturen liknar i många avseenden vanlig ST-kod i 800xA. Det finns även en särskild variant, Start Equation Block, som används för kod som endast ska köras vid initiering, exempelvis efter programnedladdning eller kallstart. Någon direkt motsvarighet till detta finns inte i 800xA, men liknande funktion kan implementeras med uppstartslogik, exempelvis via FirstScan eller villkor som endast är sanna vid start.

Skillnader och likheter mellan SattLines Equation Block och motsvarande ST-konstruktioner i 800xA sammanfattas i tabell 4.9.

Tabell 4.9: Mappning mellan SattLine Equation Block och Structured Text i 800xA.

| SattLine                        | 800xA                            | Kommentarer                                                                                                  |
|---------------------------------|----------------------------------|--------------------------------------------------------------------------------------------------------------|
| Equation Block                  | ST                               | Går att migrera, men med några förbehåll                                                                     |
| Grafiska koordinater            | <i>Finns ej</i>                  | Layoutmetadata i SattLine; migreras ej                                                                       |
| IF/ THEN/ ELSIF/<br>ELSE/ ENDIF | IF/ THEN/ ELSIF/<br>ELSE/ END_IF | Nästan identisk syntax men identisk logik;<br>notera END_IF istället för ENDIF och<br>tilldelning notationen |
| Tilldelning (=)                 | Tilldelning (: =)                |                                                                                                              |

För att konkret illustrera dessa skillnader visas nedan ett enkelt syntaxexempel i SattLine och motsvarande uttryck i 800xA.

#### ***SattLine syntaxexempel:***

```
IF Timeout THEN
    Out1 = true;
    Out2 = true;
ELSIF NOT SumAlarm THEN
    Out1 = AutoMode AND NOT LS1;
ELSE
    Out1 = false;
ENDIF;
```

#### ***800xA syntaxexempel:***

```
IF Timeout THEN
    Out1 := TRUE;
    Out2 := TRUE;
ELSIF NOT SumAlarm THEN
    Out1 := AutoMode AND NOT LS1;
ELSE
    Out1 := FALSE;
END_IF;
```

### ***Sequence Block → Sequential Function Chart (SFC)***

Ett Sequence Block i SattLine är en sekvens baserad på SFC-standarden med steg, transitioner, alternativa och parallella grenar samt kodsektioner per steg. SattLines sekvenssemantik överensstämmer med IEC-SFC-beteendet i 800xA och kan realiseras direkt utan logikändring. Övergripande motsvarigheter mellan SattLines Sequence Block och SFC-strukturen i 800xA sammanfattas i tabell 4.10.

Tabell 4.10: Mappning mellan SattLine Sequence Block och SFC i 800xA.

| SattLine                                                                                                           | 800xA           | Kommentarer                                                     |
|--------------------------------------------------------------------------------------------------------------------|-----------------|-----------------------------------------------------------------|
| Sequence Block                                                                                                     | SFC block       | Logiken mappas direkt                                           |
| Grafiska koordinater                                                                                               | <i>Finns ej</i> | Layoutmetadata i SattLine; migreras ej                          |
| Steps, transitions, alternativ/parallell<br>sekvens, hopp/förgrening,<br>(P1,N,P0)/(Entercode/Activecode/Exitcode) |                 | Nästan identisk syntax men identisk logik;<br>se syntax för ST. |

## 4.3 Konvertering av SattLine-filer till XML

Analysen av SattLine-strukturen (avsnitt 4.1) och 800xA-mappningen (avsnitt 4.2) utgör tillsammans underlaget för hur en automatiserad konvertering kan realiseras via CB Open Interface (se avsnitt 2.4). CB Open Interface exponerar metoder som *NewDataType()*, *NewFunctionBlockType()* och *NewControlModuleType()*, vilka vardera tar ett XML-formaterat innehåll som inparameter. Det innebär att XML-filer som genereras vid konverteringen, i ett format som CB Open Interface kan tolka, direkt kan läsas in och skickas till Control Builder.

### 4.3.1 Tvåstegsflöde och kontrollpunkt

Som beskrivs i avsnitt 3.5 delas konverteringen upp i två steg med ett granskningsskede emellan. Uppdelningen ger en kontrollpunkt där resultatet kan verifieras innan något skrivs till 800xA-projektet, vilket minskar risken för att felaktig information propageras genom hela flödet.

### 4.3.2 RECORD kontra MODULE - komplexitetsskillnaden

Vid analys av biblioteksfilerna (se avsnitt 4.1.2) identifierades två huvudsakliga kategorier av typdefinitionsblock:

- A. **RECORD-typdefinitionsblock** deklarerar sammansatta strukturerade datatyper med ett fast, platt schema
- B. **MODULE-typdefinitionsblock** beskriver modulers parametrar, variabler, eventuella submoduler, grafiska aspekter samt programkod i form av strukturerade kodblock eller sekvenser

Analysen visade att detta mönster mappas naturligt till ett *<DataType>*-element i CB Open Interface-schemat, där varje komponent i RECORD-blocket representeras som ett *<Component>*-underelement., och importeras med ett enda anrop till *NewDataType()*:

```
<?xml version="1.0" encoding="UTF-16"?>
<DataType xmlns="CBOpenIFSchema3_0"
  Name="" Protected="0" Hidden="0" Scope="public">
  <Components>
    <Component Name="" Type="" Attribute=""
      InitialValue="" ReadPermission="" WritePermission=""
      AuthenticationLevel="None" TypePath="">
      <Description>...</Description>
    </Component>
  </Components>
  <Description>...</Description>
</DataType>
```

En MODULE-definition är däremot en sammansatt enhet (parametrar, lokala variabler, submoduler, grafik, kodblock) och kräver de mer komplexa typkonstrukterna *FunctionBlockType* eller *ControlModuleType*.

### 4.3.3 XML-struktur för FunctionBlockType

CB Open Interface-manualen definierar ett fullständigt XML-schema för FunctionBlockType som rymmer parametrar, variabler, externa variabler, inbäddade funktionsblock och kodblock i en och samma XML-sträng:

```
<?xml version="1.0" encoding="UTF-8"?>
<FunctionBlockType xmlns="CBOpenIFSchema3.0" Name="MyFBType" Protected="0"
Scope="public" SILLevel="NonSIL">

  <!-- Parametrar (motsvarar SattLine MODULEPARAMETERS) -->
  <Parameters>
    <Parameter Name="Par1" Type="dint" Attribute="retain" Direction="in"
InitialValue="7" TypePath="System.dint">
      <Description>Description of Par1</Description>
    </Parameter>
    <Parameter Name="Par2" Type="CalendarStruct" Direction="inout"
TypePath="System.CalendarStruct"/>
  </Parameters>
  <ExtensibleParameters/>

  <!-- Lokala variabler (motsvarar SattLine LOCALVARIABLES) -->
  <Variables>
    <Variable Name="Var1" Type="real" Attribute="retain" InitialValue="7.0"
TypePath="System.real">
      <Description>Description of Var1</Description>
    </Variable>
  </Variables>

  <!-- Externa variabler (konstanter fran omgivande bibliotek) -->
  <ExternalVariables>
    <ExternalVariable Name="ExtVar1" Type="dint" Attribute="constant"
TypePath="System.dint"/>
  </ExternalVariables>

  <!-- Inbäddade funktionsblock (motsvarar SattLine SUBMODULES) -->
  <FunctionBlocks>
    <FunctionBlock Name="FBInst1" Type="AnotherFBType"
TaskConnection="NewCont.Fast" TypePath="MyApplication 1.0-0.AnotherFBType">
      <!-- ... -->
    </FunctionBlock>
  </FunctionBlocks>

  <!-- CodeBlocks (motsvarar SattLine ModuleCode) -->
  <!-- ... -->
</FunctionBlockType>
```

Mappningen mellan SattLines MODULE-sektioner och 800xA:s FunctionBlockType-element är därmed:

- MODULEPARAMETERS → <Parameters>,
- LOCALVARIABLES → <Variables>
- SUBMODULES → <FunctionBlocks> (vid FBT) resp. <ControlModules> (vid CMT)
- ModuleCode -> <CodeBlocks>

De delar av SattLine-modulen som inte migreras *GraphObjects*, *InteractObjects* och *zoom-/layoutattribut* har ingen motsvarighet i XML-schemat och hoppas därför över i konverteringen.

## XML-struktur för ControlModuleType

Schemat är strukturellt likvärdigt med *FunctionBlockType*.

```
<?xml version="1.0" encoding="UTF-8"?>
<ControlModuleType xmlns="CBOpenIFSchema3.0" Name="MyCMType" Protected="0"
    Scope="public" SILLevel="NonSIL">

    <!-- CM-parametrar (motsvarar SattLine MODULEPARAMETERS) -->
    <CMPParameters>
        <CMPParameter Name="Par1" Type="dint" InitialValue="7"
TypePath="System.dint">
            <Description>Description of Par1</Description>
        </CMPParameter>
    </CMPParameters>

    <Variables/>
    <ExternalVariables/>
    <FunctionBlocks/>

    <!-- Inbäddade kontrollmoduler (submoduler av CM-typ) -->
    <ControlModules>
        <ControlModule Name="AnotherCMType1" Type="AnotherCMType"
            VisibilityInGraphics="invisible"
            TypePath="MyApplication 1.0-0.AnotherCMType">
            <CMConnections/>
            <GraphPos Xpos="0.0" Ypos="0.0" Rotation="0.0"
                Xscale="1.0" Yscale="1.0"/>
        </ControlModule>
    </ControlModules>

    <!-- Kodblock -->
    <CodeBlocks>
        <STCodeBlock Name="Code">
            <STCode>FbInst1.ParA := Par1;</STCode>
        </STCodeBlock>
    </CodeBlocks>

    <Description>Description of MyCMType</Description>

    <!-- Grafisk koordinatsystem-definition -->
    <GraphSize>
        <Point X="-1.0" Y="-1.0"/>
        <Point X="1.0" Y="1.0"/>
    </GraphSize>
</ControlModuleType>
```

## XML-struktur: ST och SFC

CB Open Interface stöder flera typer av kodblock. För konvertering från SattLine är de två relevanta typerna ST och SFC, då SattLines equation blocks mappas till ST och SattLines sekvensblock mappas till SFC.

Structured Text representeras i XML som ett *<STCodeBlock>*-element där själva koden bäddas in som ett textnodvärde i *<STCode>*-elementet:

```
<CodeBlocks>
  <STCodeBlock Name="CodeBlock1">
    <STCode>if Var1 &gt; 0 then FBInst1.Par1 := Var1;endif</STCode>
  </STCodeBlock>
  <STCodeBlock Name="CodeBlock2">
    <STCode>Var1 := Var2 + 1;</STCode>
  </STCodeBlock>
</CodeBlocks>
```

Sequential Function Chart (SFC) representeras med en mer hierarkisk XML-struktur där steg, transitioner, parallella grenar och alternativa grenar har egna element. Varje stegs action och varje transitions villkor skrivs som inbäddad ST-kod:

```
<CodeBlocks>
  <SFCCodeBlock Name="SeqControll1" StepElapsedTime="0" SFCViewerAspect="0">
    <SFCStep Name="S1" InitialStep="1">
      <PlAction><STCode>a1 := 1;</STCode></PlAction>
    </SFCStep>
    <SFCTransition Name="Tr1"><True/></SFCTransition>

    <!-- Parallella grenar -->
    <SFCSimultaneous>
      <SFCBranch>
        <SFCStep Name="S2" InitialStep="0"/>
      </SFCBranch>
      <SFCBranch>
        <SFCStep Name="S3" InitialStep="0"/>
      </SFCBranch>
    </SFCSimultaneous>
    <SFCTransition Name="Tr2"><True/></SFCTransition>

    <!-- Alternativa grenar -->
    <SFCStep Name="S4" InitialStep="0"/>
    <SFCSelection>
      <SFCBranch>
        <SFCTransition Name="Tr3"><True/></SFCTransition>
      </SFCBranch>
      <SFCBranch>
        <SFCTransition Name="Tr4" Dest="S1"><True/></SFCTransition>
      </SFCBranch>
    </SFCSelection>

    <!-- Delsekvens -->
    <SFCSubSequence SubSeqName="Subseq">
      <SFCStep Name="S5" InitialStep="0"/>
      <SFCTransition Name="Tr5"><True/></SFCTransition>
    </SFCSubSequence>
  </SFCCodeBlock>
</CodeBlocks>
```

## 4.4 Skapande av objekt i 800xA med CB Open Interface

CB Open Interface är ett COM-baserat gränssnitt som öppnar upp Control Builders funktionalitet för externa applikationer. Gränssnittet gör det möjligt att programmässigt skapa, modifiera och ta bort objekt i ett 800xA-projekt utan att interagera manuellt med Control Builders grafiska gränssnitt. I konverteringskontexten används gränssnittet som det sista steget i tvåstegsflödet: de XML-filer som genererats och granskats i steg ett läses in och skickas till Control Builder via CB Open Interface-metoderna.

### 4.4.1 Förberedelse av utvecklingsmiljö

Importflödet testades i en Windows-miljö med Control Builder, registrerat CB Open Interface och 32-bitars Python med pythonnet. Detaljer redovisas i Appendix C.

### 4.4.2 Python-till-COM-bryggan

Bryggan mellan Python och CB Open Interface byggs upp i tre steg. Först laddar `clr.AddReference()` *Interop.CONTROLBUILDERLib.dll* från disk, vilket registrerar .NET-interop-wrappern för COM-typbiblioteket. Därefter instansieras *CBOpenIFClass* via *Activator.CreateInstance()*. Om direktinstansiering misslyckas faller skriptet tillbaka på ProgID-aktivering (*CONTROLBUILDERLib.CBOpenIF*) och slutligen CLSID-aktivering, vilket ger robusthet mot variationer i installationsmiljön. Det resulterande *cb*-objektet exponerar alla CB Open Interface-metoder som Python-attribut.

```
clr.AddReference(INTEROP_DLL_PATH)
import CONTROLBUILDERLib
cb = Activator.CreateInstance(CONTROLBUILDERLib.CBOpenIFClass)
```

En viktig begränsning är att COM-körtidsmiljön är x86-orienterad. Skriptet verifierar processorarkitekturen vid uppstart och avbryter med ett tydligt felmeddelande om det körs i 64-bitars Python, eftersom ett *BadImageFormatException* annars uppstår vid DLL-laddningen.

### 4.4.3 Objektsökvägssyntax

För att identifiera vilket objekt i 800xA-projektet en CB Open Interface-metod ska operera på används en punktseparerad sökvägssyntax. Enkla toppnivåobjekt adresseras med enbart sitt namn; objekt längre ner i hierarkin adresseras med fullständig sökväg, se tabell 4.11.

Tabell 4.11: Exempel på objektsökvägar i CB Open Interface.

| Objekt                                      | Exempelsökväg           |
|---------------------------------------------|-------------------------|
| Bibliotek Library1                          | Library1                |
| DataType DT1 i Library1                     | Library1.DT1            |
| FunctionBlockType med versionsspecificering | Library1 2.0-3.MyFBType |
| ControlModule CM1 i CMT CMT1 i Application1 | Application1.CMT1.CM1   |

Om inget versionsnummer anges väljer CB Open Interface automatiskt den högsta tillgängliga versionen. Parametern *appOrLibName* i *NewDataType()*, *NewFunctionBlockType()* och liknande metoder följer samma syntax. Det är alltså biblioteksnamnet som styr var den nya typen placeras i projektträdet.

#### 4.4.4 Tillgängliga metoder

CB Open Interface exponerar metoder för att skapa och hantera objekt i ett Control Builder-projekt. De metoder som är mest relevanta redovisas i tabellen 4.11.

Tabell 4.12: Relevanta CB Open Interface-metoder för konvertering av SattLine-bibliotek.

| Metod                    | Beskrivning                     |
|--------------------------|---------------------------------|
| NewLibrary()             | Skapar ett nytt bibliotek       |
| NewApplication()         | Skapar en ny applikation        |
| NewDataType()            | Skapar en ny datatyp från XML   |
| NewFunctionBlockType()   | Skapar en ny FBT från XML       |
| NewControlModuleType()   | Skapar en ny CMT från XML       |
| NewFunctionBlock()       | Skapar en FB-instans            |
| NewControlModule()       | Skapar en CM-instans            |
| NewSingleControlModule() | Skapar en SCM                   |
| NewProgram()             | Skapar ett program              |
| SetTaskConnection()      | Kopplar ett objekt till en task |
| GetProjectTree()         | Läser projektträdet som XML     |

New- och Set-metoderna tar en XML-sträng som parameter. Strängen måste följa *CBOpenIFSchema3\_0*-schemat, som Control Builders interna MSXML-parser validerar mot vid varje anrop.

#### 4.4.5 Felhantering: HRESULT och Message Bucket

CB Open Interface rapporterar fel på två separata nivåer.

Den första nivån är HRESULT, ett COM-standardvärde som returneras av varje metदानrop. Ett fel på denna nivå kastar ett undantag i Python och fångas med try/except. Vanliga HRESULT-fel är *OI\_E\_MODE* (Control Builder är i online- eller testläge), *OI\_E\_XML* (ogiltig XML-struktur) och *OI\_E\_ALREADYEXIST* (typen finns redan).

Den andra nivån är Message Bucket, en XML-kodad sträng som returneras som retur från New- och Set-metoderna när *HRESULT = S\_OK*. En tom sträng indikerar framgång utan anmärkning. En icke-tom sträng innehåller ett *<MessageBucket>*-element med *<Error>*- och *<Warning>*-underelement, vart och ett med felnummer, meddelandetext och positionsinformation. Message Bucket-mekanismen finns till för att Control Builder ska kunna lagra och rapportera applikationslogikfel som till exempel inkonsekventa initialvärden eller syntaxfel i ST-kod, utan att avbryta metदानropet med ett HRESULT-fel.

I importskriptet loggas returvärdet från varje New-anrop. Om strängen inte är tom skrivs innehållet ut som en varning:

```
msg = cb.NewDataType(datatype_name, libname, xml_block)
if msg:
    print(f"[{datatype_name}] Message bucket: {msg}")
```

#### 4.4.6 Importflödet i Python

Importskriptets flöde för ett genererat XML-dokument består av följande steg:

1. Läs XML-filen filen öppnas med UTF-8-kodning med fallback till UTF-16, det format som CB Open Interface självt använder vid export.
2. Namnrymdsinjicering varje `<DataType>`- eller `<FunctionBlockType>`-block extraheras med reguljärt uttryck. Om blocket saknar `xmlns`-attributet injiceras `xmlns="CBOpenIFSchema3_0"` i öppningstaggen. CB Open Interface kräver korrekt namnrymdsdeklaration för att MSXML-parsern internt ska kunna validera mot schemat.
3. Extraktion av typnamn `Name`-attributet läses ur XML-blocket och används som namnparameter i metदानropet.
4. Anrop till New-metoden det bearbetade XML-blocket skickas som sträng till `NewDataType()` eller `NewFunctionBlockType()`. Returvärdet (Message Bucket) loggas om det inte är tomt.
5. Iteration steg 2–4 upprepas för varje typblock i filen, i den topologiska ordning som bibliotekshierarkin kräver.

Control Builder måste vara i offline-läge under hela importflödet. Skriptet kopplar inte heller upp något projekt automatiskt. Det ansluter till det projekt som redan är öppet i Control Builder-gränssnittet vid körningens start. Koden finns under *Appendix A – Python kod: XML till 800xA konvertering*.



## 5 Resultat

Resultaten omfattar framför allt den verktygskedja som utvecklades för att gå från en SattLine-textfil till CB Open Interface XML och vidare till objekt i ABB 800xA. Kapitlet fokuserar på vad som faktiskt togs fram, hur verktygen användes i praktiken och vilket resultat de gav.

En viktig utgångspunkt är att examensarbetets mest centrala resultat inte blev en fullständigt automatisk migreringsmotor för alla typer av SattLine-objekt. Det centrala resultatet blev i stället en fungerande, användbar och dokumenterad arbetskedja för konvertering och import – en lösning som uppfyllde det praktiska mål som företaget i första hand efterfrågade. Resultatet kan sammanfattas i följande huvudpunkter:

- SattLine-filer i textformat kan läsas, tolkas och delas upp i meningsfull struktur.
- Den extraherade strukturen kan omvandlas till XML i CB Open Interface-format.
- XML-filerna kan granskas, valideras och importerar till 800xA.
- Ett komplett arbetsflöde med både skript och GUI togs fram.
- Tanksystem användes som jämförelsefall för att visa varför fullständig automatisk migrering inte är ett rimligt mål när käll- och målsystem inte är uppbyggda enligt samma objektmodell.

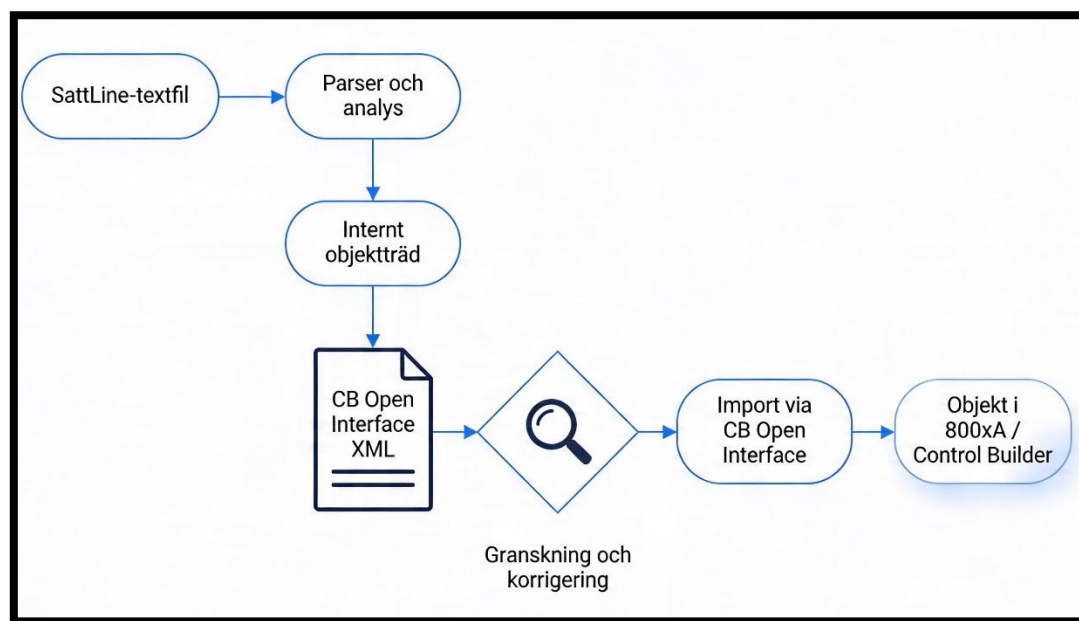
### 5.1 Översikt över framtagna verktyg

Arbetet resulterade i flera konkreta program vilka tillsammans bildar en komplett verktygskedja. Dessa delar motsvarar de steg som beskrevs i metodkapitlet: förstudie, uppsättning testmiljö, testprojekt, analys av SattLine och 800xA System och utveckling och testning av Pythonverktyg. De framtagna verktygen och deras huvudsakliga funktioner sammanfattas i tabell 5.1.

Tabell 5.1: Översikt över framtagna verktyg och deras funktion.

| Resultatkomponent           | Funktion                                            | Resultat i arbetet                                                                      |
|-----------------------------|-----------------------------------------------------|-----------------------------------------------------------------------------------------|
| sattline_parser.py          | Läser SattLine-textfiler och bygger intern struktur | Kunde extrahera RECORD-, MODULEDEFINITION-, Equation Block- och Sequence Block-struktur |
| sattline_gui.py             | GUI för val av filer och körning av parsern         | Gav ett användbart arbetsgränssnitt för konvertering utan manuell kommandoradshantering |
| cboi_cli.py                 | CLI mot CB Open Interface                           | Kunde validera XML, importera XML och skapa objekt i 800xA                              |
| cboi_import_gui.py          | GUI för import och validering                       | Förenklade praktisk användning i Control Builder                                        |
| Genererade XML-filer        | Mellanresultat för granskning och import            | Gav både samlade granskningsfiler och separata objektfiler                              |
| Output-kataloger och loggar | Dokumentation av körningar                          | Gjorde arbetsflödet reproducerbart och lätt att följa                                   |

Figur 5.1 illustrerar det samlade arbetsflödet från SattLine-källfil till importerat objekt i 800xA.



Figur 5.1: Översikt över verktygskedjan från SattLine-textfil till objekt i 800xA.

## 5.2 Resultat från parser och analys av SattLine-filer

Det första huvudresultatet var att SattLine-filer kunde tolkas systematiskt i textform. Detta var en direkt följd av metodkapitlets analysdrivna angreppssätt, där arbetet inleddes med att förstå hur bibliotek, datatyper, moduler och kodblock representeras i SattLines textfiler. Analysen visade att SattLine-filer blandar deklarationer, grafikdata, modulkopplingar och kod i samma fil, en heterogen struktur som ändå gick att separera tillräckligt väl för vidare behandling.

Parseern kunde identifiera och extrahera följande delar:

- RECORD-definitioner och deras komponenter
- MODULEDEFINITION med parametrar och lokala variabler
- submoduler och kopplingar
- Equation Blocks och Sequence Blocks
- typnamn och typpreferenser
- standardvärden och attribut

Parseern kunde även hantera praktiska problem i källfiler, till exempel olika teckenkodningar, beskrivningssträngar och andra specialfall, variationer i deklarationssätt mellan olika objekt, samt typmappning mellan SattLine-typer och 800xA-nära typer till exempel *BooleanSignal* till *BoolIO*.

Ett exempel på hur parseern hanterade variationer i SattLine-filerna visas i Kodexempel 5.1. Utdraget är hämtat från *ProcessObjectDemoLib.txt* och innehåller flera typiska fall: flera fält på samma rad, initialvärden, SattLine-specifika strängtyper och attribut.

Kodexempel 5.1: Exempel på SattLine-deklarationer som parsern behövde tolka.

```
SearchFor2, SearchFor3, SearchFor4: Tagstring := "None";
ObjectOrder: integer := -1;
Found: boolean State;
ObjectRunning, Changed: boolean;
```

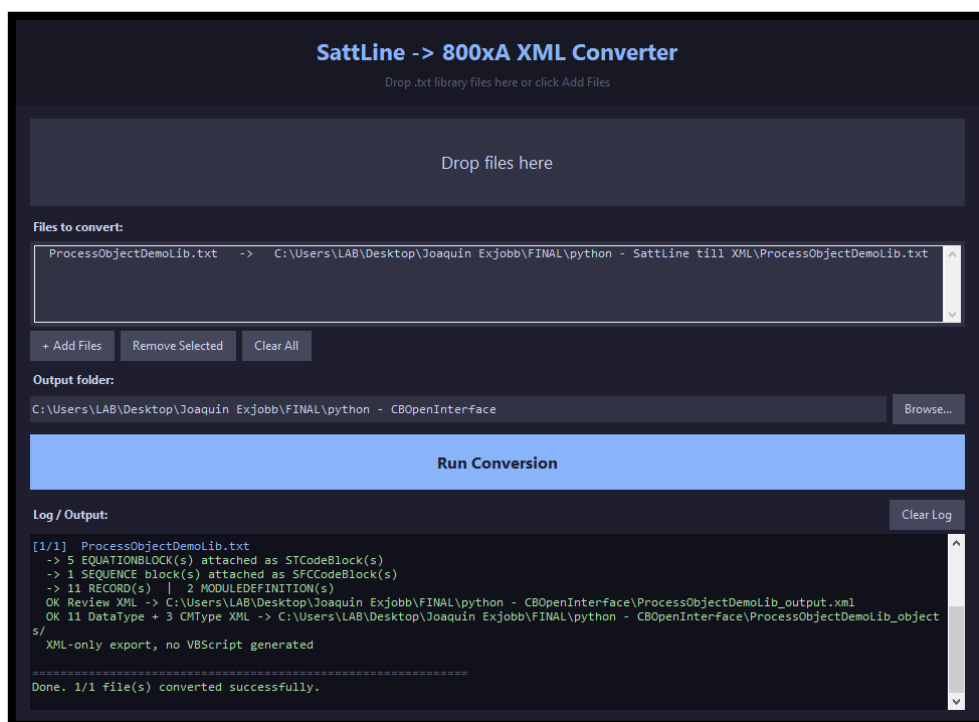
Parsern delade upp deklARATIONERNA i separata fält och sparade datatyp, initialvärde och attribut var för sig. Raden *SearchFor2, SearchFor3, SearchFor4: Tagstring := "None";* blev tre separata fält med samma datatyp och initialvärde, medan *Found: boolean State;* tolkades som ett booleskt fält med attributet *State*. På så sätt kunde verktyget hantera verkliga SattLine-deklarationer och inte bara enkla exempel med ett fält per rad.

Ett annat exempel är att SattLine ibland anger information om parameterriktning i beskrivningssträngen eftersom det inte går att ange det som ett eget attribut. Detta visas i Kodexempel 5.2.

Kodexempel 5.2: Riktning angiven i SattLine-beskrivning.

```
Name "IN Module name": string := "Module name";
IO "IN/OUT IO": IO_ReactorType;
SO "OUT for indication in processpicture": SO_ReactorType := Default;
```

I dessa rader finns riktningen inte som ett formellt SattLine-attribut, utan som prefix i beskrivningssträngen. Parsern kunde därför tolka *IN* som *in*, *IN/OUT* som *in\_out* och *OUT* som *out*, samtidigt som resten av texten sparades som beskrivning. Den strukturerade informationen från parsern användes därefter som indata till XML-genereringen. I avsnitt 5.3 visas hur de extraherade fälten omformades till CB Open Interface XML. Figur 5.2 visar GUI-verktyget som användes för att köra parsern och granska output, logg och genererade filer i den praktiska testmiljön.



Figur 5.2: Skärmdump av sattline\_gui.py med filval, output-katalog och logg.

## 5.3 Resultat från XML-generering

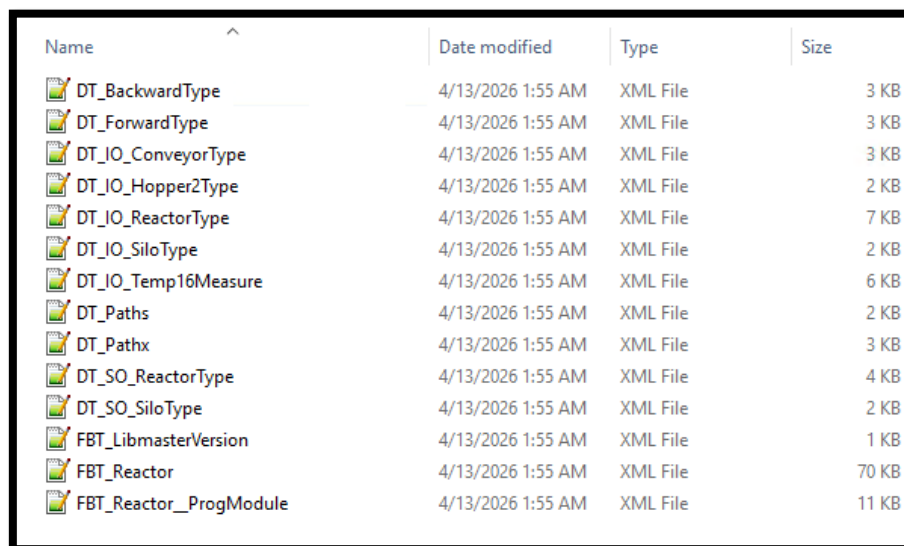
Nästa huvudresultat var att den interna strukturen kunde serialiseras till CB Open Interface XML, det format som 800xA faktiskt kan ta emot. För varje körd SattLine-fil genererades två typer av utdata: en samlad granskningsfil i XML-format samt en uppsättning separata objektfiler i en underkatalog.

Detta var ett medvetet designval istället för att skicka allt direkt till 800xA skapades en tydlig kontrollpunkt där resultatet kunde granskas och verifieras först. Denna mellanliggande granskningsnivå visade sig i praktiken vara värdefull, eftersom den gjorde det möjligt att kontrollera att typnamn, parametrar, komponenter och kodblock hade översatts på ett rimligt sätt innan import genomfördes.

Ett konkret resultat från den generella testkörningen på *ProcessObjectDemoLib.txt* var att flera objekt kunde genereras. Antalet och typen av dessa sammanfattas i tabell 5.2. Figur 5.3 visar output-katalogen med de objektfiler som genererades vid testkörningen.

Tabell 5.2: Antal genererade objekt från testkörning på *ProcessObjectDemoLib*.

| Objekt typ        | Antal | Exempel                                                |
|-------------------|-------|--------------------------------------------------------|
| DataType          | 11    | IO_ReactorType, SO_ReactorType, IO_ConveyorType, Paths |
| ControlModuleType | 3     | LibmasterVersion, Reactor, Reactor_ProgModule          |
| Totalt            | 14    | Kombination av datatyper och modultyper                |



| Name                   | Date modified     | Type     | Size  |
|------------------------|-------------------|----------|-------|
| DT_BackwardType        | 4/13/2026 1:55 AM | XML File | 3 KB  |
| DT_ForwardType         | 4/13/2026 1:55 AM | XML File | 3 KB  |
| DT_IO_ConveyorType     | 4/13/2026 1:55 AM | XML File | 3 KB  |
| DT_IO_Hopper2Type      | 4/13/2026 1:55 AM | XML File | 2 KB  |
| DT_IO_ReactorType      | 4/13/2026 1:55 AM | XML File | 7 KB  |
| DT_IO_SiloType         | 4/13/2026 1:55 AM | XML File | 2 KB  |
| DT_IO_Temp16Measure    | 4/13/2026 1:55 AM | XML File | 6 KB  |
| DT_Paths               | 4/13/2026 1:55 AM | XML File | 2 KB  |
| DT_Pathx               | 4/13/2026 1:55 AM | XML File | 3 KB  |
| DT_SO_ReactorType      | 4/13/2026 1:55 AM | XML File | 4 KB  |
| DT_SO_SiloType         | 4/13/2026 1:55 AM | XML File | 2 KB  |
| FBT_LibmasterVersion   | 4/13/2026 1:55 AM | XML File | 1 KB  |
| FBT_Reactor            | 4/13/2026 1:55 AM | XML File | 70 KB |
| FBT_Reactor_ProgModule | 4/13/2026 1:55 AM | XML File | 11 KB |

Figur 5.3: Skärmdump av output-katalogen för *ProcessObjectDemoLib* med genererade objektfiler.

Ett kort kodexempel på genererad XML visas nedan samt motsvarigheten i SattLine-textfilen. Exemplet illustrerar hur en RECORD-definition i SattLine omformades till den *DataType*-struktur som CB Open Interface förväntar sig:

- Deklarationen av *IO\_ReactorType* i SattLine-textfilen:

```
IO_ReactorType = RECORD DateCode 479481731
    InletWater: IO_Valve2W2PType;
    Level: AnalogSignalIn;
    LevelDesc: string;
    Outlet: IO_Valve2W2PType;
ENDDEF
(*IO_ReactorType*);
```

- Genererad XML:

```
<?xml version="1.0" encoding="UTF-16"?>
<DataType xmlns="CBOpenIFSchema3_0" Name="IO_ReactorType" Protected="0"
Hidden="0" Scope="public">
  <Components>
    <Component Name="InletWater" Type="IO_Valve2W2PType" Attribute=""
InitialValue="" ReadPermission="" WritePermission=""
AuthenticationLevel="None" TypePath="IO_Valve2W2PType"/>
    <Component Name="Level" Type="AnalogSignalIn" Attribute=""
InitialValue="" ReadPermission="" WritePermission=""
AuthenticationLevel="None" TypePath="AnalogSignalIn"/>
    <Component Name="LevelDesc" Type="string" Attribute="" InitialValue=""
ReadPermission="" WritePermission="" AuthenticationLevel="None"
TypePath="System.string"/>
    <Component Name="Outlet" Type="IO_Valve2W2PType" Attribute=""
InitialValue="" ReadPermission="" WritePermission=""
AuthenticationLevel="None" TypePath="IO_Valve2W2PType"/>
  </Components>
  <Description>From SattLine RECORD 'IO_ReactorType' (DateCode 479481731)
[PUBLIC]</Description>
</DataType>
```

Kodexemplet visar att resultatet inte bara var textkopiering, utan en faktisk omformning till rätt XML-struktur med komponentnamn, typnamn och typpreferenser. Det rödmarkerade typnamnet i SattLine-definitionen återkommer i den genererade XML-strukturen och visar därmed hur datatypens namn förs över mellan representationerna. De färgmarkerade fälten i RECORD-definitionen motsvaras samtidigt av de markerade komponenterna i XML-exemplet.

## 5.4 Resultat från import till 800xA

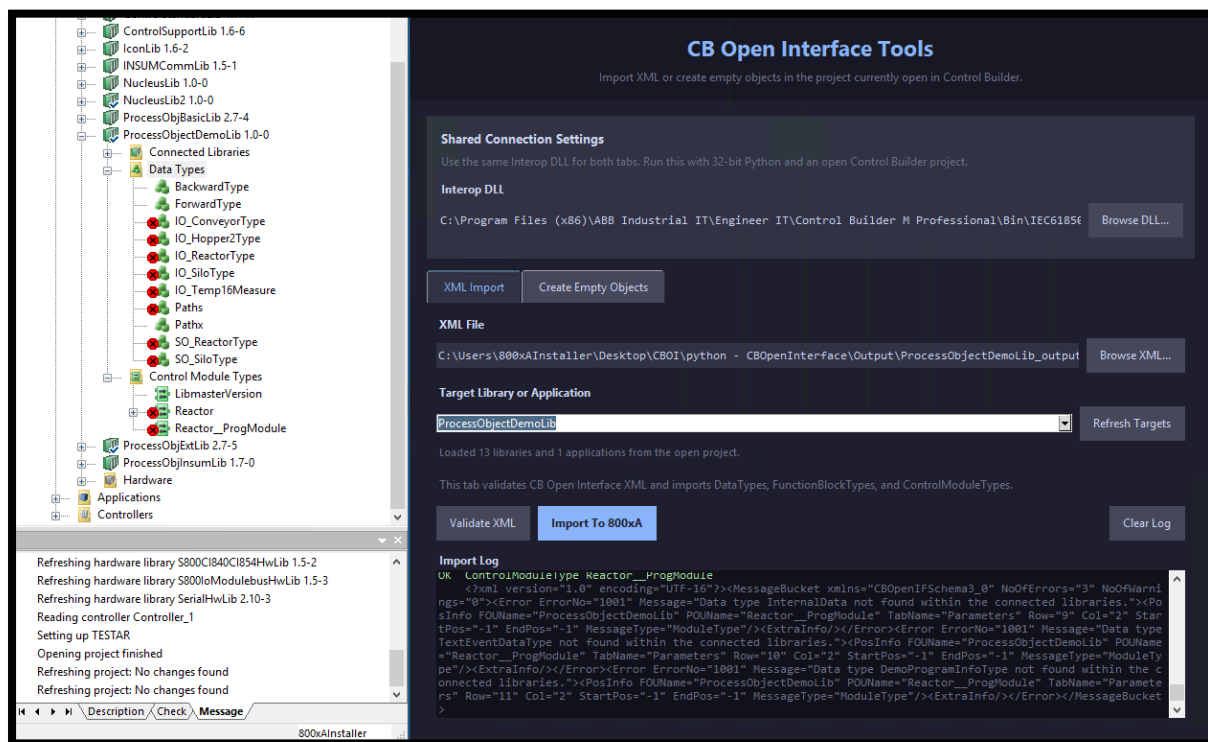
Det viktigaste praktiska resultatet i examensarbetet var att kedjan *SattLine txt* → *XML* → *800xA* fungerade som avsett på den nivå som definierades i arbetet. Resultatet var alltså inte bara att XML kunde genereras, utan att XML också kunde användas i nästa steg för att skapa eller importera objekt i 800xA via CB Open Interface.

*Command Line Interface*-verktyget *cboi\_cli.py* gav stöd för att läsa in XML-filer, validera dem innan import, välja rätt mål i 800xA samt anropa CB Open Interface-metoder som *NewDataType()*, *NewFunctionBlockType()* och *NewControlModuleType()*. GUI-verktyget *cboi\_import\_gui.py* gav samma typ av funktionalitet i en mer tillgänglig och direkt arbetsmiljö.

Det praktiska resultatet kan sammanfattas som:

- konvertering från textfil till XML fungerade
- XML-filerna kunde granskas och valideras
- importerbara objekt kunde skapas i 800xA

Figur 5.4 visar den praktiska importmiljön, där XML-filer kunde valideras och användas för att skapa objekt i 800xA.



Figur 5.4: Skärmdump av cboi\_import\_gui.py med validerings- och importfunktion samt control builder till vänster

## 5.5 Resultat i form av en användbar arbetsmiljö

Examensarbetet resulterade i en sammanhängande arbetsmiljö och inte bara i en enstaka parserfil. Kombinationen av parser, XML-generator, CLI och GUI gjorde att processen kunde användas steg för steg:

- 1) Välj SattLine-fil i GUI.
- 2) Kör konvertering, parsern extraherar struktur och genererar XML.
- 3) Granska XML-filer och loggar i output-katalogen.
- 4) Validera filerna innan import.
- 5) Importera till 800xA via GUI eller CLI.

Följande funktioner bidrog särskilt till användbarheten: filval och output-hantering i GUI, loggning av körningar och fel, tydlig uppdelning mellan generering och import, möjlighet att validera innan något skrivs till projektet, samt möjlighet att skapa objekt av olika typer via samma verktygskedja. Metoden i arbetet innebär inte bara att analysera systemen utan att bygga, prova och iterera fram ett faktiskt användbart arbetssätt.



## 6 Diskussion

I detta kapitel diskuteras de huvudsakliga resultaten från examensarbetet, deras implikationer och de begränsningar som framkommit under arbetet.

### 6.1 Samlad resultatbedömning

Sammantaget visar resultaten att examensarbetet lyckades med det som var mest relevant inom den givna tidsramen och i relation till företagets behov. Det som faktiskt uppnåddes var:

- en fungerande parser för SattLine-textfiler
- en fungerande XML-generator för CB Open Interface-format
- fungerande stöd för validering och import till 800xA
- en användbar arbetsmiljö med både CLI och GUI
- en dokumenterad metod för hur konverteringsarbetet kan genomföras

Resultatet bör ses som en fungerande demonstrationslösning och en dokumenterad arbetsmetod, snarare än ett färdigt produktionssystem för total automation. Samtidigt gav arbetet företaget ett verktyg och ett arbetssätt som fungerar i praktiken och kan ligga till grund för fortsatt utveckling.

### 6.2 Modellskillnader som påverkar migreringen

Det djupaste problemet i konverteringen är inte tekniskt utan konceptuellt. SattLine och 800xA bygger på olika objektmodeller. I SattLine kan ett processobjekt samla I/O, processkod, grafik, paneler och biblioteksberoenden i samma fil. I 800xA delas motsvarande funktion i stället upp på parametrar, variabler, function blocks, control modules och aspekter. Det innebär att en direkt kopiering från SattLine till 800xA kan ge importerbara objekt, men inte nödvändigtvis objekt som fungerar naturligt i 800xA. Resultatet från pipelineimporten visar därför främst att strukturen kan flyttas, medan en fullständig migration kräver aktiva designbeslut om hur objekten ska återskapas i 800xA:s terminologi.

### 6.3 Verktygens styrkor och begränsningar

Verktygskedjan som togs fram har tydliga styrkor. Den är reproducerbar, loggad och uppdelad i tydliga steg med granskningspunkter, vilket gör den kontrollbar för en ingenjör. Verktøget är inte bara ett konceptuellt skelett utan ett praktiskt fungerande verktyg. Begränsningarna är lika tydliga. Parsern hanterar de vanligaste SattLine-konstruktionerna men täcker inte nätverkskonfiguration och grafiska metadata kräver ytterligare arbete. Typmappningen från SattLine-typer till 800xA-typer utgör en approximation. En korrekt mappning är beroende av objektets funktionella roll i den specifika anläggningen, vilket inte kan avgöras automatiskt av verktøget. Dessutom är mappningen i hög grad projektberoende och kan därför förväntas variera mellan olika tillämpningar. Dessa begränsningar är förväntade och rimliga för en proof of concept, men de behöver adresseras i en produktionslösning.



## 7 Slutsats

### 7.1 Sammanfattning av de viktigaste resultaten

Examensarbetet syftade till att undersöka och ta fram en metod för att konvertera befintliga SattLine-projekt till ABB:s moderna DCS-system 800xA på ett effektivt och tillförlitligt sätt, samt att ta fram en fungerande demonstration av konverteringsprocessen. Resultatet är en verktygskedja som fungerar och är dokumenterad.

Det mest centrala resultatet är att objektens struktur, exempelvis datatyper, modulnamn, parametrar, variabler och delar av kodrepresentationen, till stor del kan extraheras och omformas automatiskt till CB Open Interface XML. Däremot kan verktyget inte själv avgöra om det importerade objektet är funktionellt ekvivalent med SattLine-originalet i en faktisk anläggning. Den funktionella migrationen kräver därför fortsatt ingenjörsgranskning, särskilt för grafik, I/O-kopplingar, biblioteksberoenden och projektspecifik styrlogik.

### 7.2 Svar på problemformuleringens frågor

#### *1. Hur är ett SattLine-projekt strukturerat och vilka centrala komponenter måste beaktas vid en konvertering?*

SattLine-projekt lagras som textfiler med en heterogen struktur som blandar RECORD-datatypdefinitioner, MODULEDEFINITION med logikblock, grafisk metadata, submodulreferenser och biblioteksberoenden i samma fil. Centrala komponenter att beakta är RECORD-typerna som närmast motsvarar DataTypes i 800xA, MODULEDEFINITION-strukturerna med inslag av både FunctionBlockType och ControlModuleType, samt signaltyper och biblioteksreferenser som objekt är beroende av. Teckenkodningsvariationer och skillnader i deklarationssätt är praktiska utmaningar som måste hanteras tidigt i parsningsfasen.

#### *2. Vilka väsentliga skillnader finns mellan SattLine och 800xA avseende programmeringsmodell, datatyper och systemarkitektur?*

De väsentliga skillnaderna mellan SattLine och 800xA ligger i systemarkitektur, programmeringsmodell och datatyper. SattLine bygger på en mer sammanvävd struktur där moduler ofta innehåller logik, parametrar, variabler, grafik och I/O-koppling i samma enhet, medan 800xA har en tydligare uppdelning mellan bibliotek, applikationer, styrenheter och hårdvarukonfiguration. Även programmeringsmodellen skiljer sig, eftersom SattLine utgår från moduler och submoduler som bär mycket av systemets funktionalitet. När det gäller datatyper finns flera enkla motsvarigheter mellan systemen, men skillnader uppstår särskilt för strängtyper, systemtyper och I/O-relaterade typer, där SattLine ofta har en närmare koppling mellan datatyp och fysisk signal än vad 800xA har. Detta innebär att en migrering mellan systemen inte bara handlar om att översätta syntax, utan om att omstrukturera information från en systemmodell till en annan.

### ***3. Hur kan ett avgränsat SattLine-projekt konverteras till 800xA på ett effektivt och tillförlitligt sätt med hjälp av tillgängliga verktyg?***

En tillförlitlig konvertering bäst genomförs i faser: en strukturell fas där SattLine-biblioteksfiler parsas och extraherade objekt transformeras till CB Open Interface kompatibla XML, och en semantisk fas där typberoenden och objektbeteenden anpassas till 800xA:s konventioner. Den strukturella fasen kan till stor del automatiseras med de verktyg som togs fram. Hantering av komplexa objekt vars beteende är tätt sammankopplat med grafik, paneler och biblioteksberoenden kräver bedömning och manuell ingenjörsinsats.

### ***4. Vilka verifieringssteg krävs för att säkerställa att funktionalitet, styrlogik och systemstruktur bibehålls efter konvertering till 800xA?***

Verifiering sker i flera steg: XML-validering utan fel är ett nödvändigt men inte tillräckligt villkor. Nästa steg är att importera till en aktiv Control Builder-miljö och bekräfta att objekten skapas utan typfel. Därefter krävs funktionell testning för att verifiera att logikbeteendet stämmer med källobjektets avsikt.

### ***5. Vilka begränsningar och risker kan identifieras vid konvertering av mer omfattande eller komplexa SattLine-system, och hur påverkar dessa möjligheten till skalning av den föreslagna metoden?***

Fyra huvudsakliga begränsningar identifierades:

- 1) **Modellskillnader:** SattLine-objekt innehåller beteendelogik tätt kopplad till systemspecifika konventioner som inte kan översättas direkt.
- 2) **Biblioteksberoenden:** typpreferenser till externa bibliotek måste hanteras manuellt eller omskapas i 800xA.
- 3) **Grafik och operatörspaneler:** aspektbaserade presentationsobjekt i 800xA är fundamentalt annorlunda uppbyggda jämfört med SattLines inbäddade grafiklogik.
- 4) **Skalbarhet:** ju större och mer heterogent källprojektet är, desto fler objekt faller utanför den generella parserns täckning och kräver specialanpassning.

## **7.3 Resultatets bidrag till syftet och hur det används**

Examensarbetet uppfyller syftet om att ta fram en metod för konvertering och en fungerande demonstration av konverteringsprocessen. Den framtagna verktygskedjan är en direkt och användbar leverans som Granitor Systems AB kan använda som bas för framtida konverteringsprojekt.

Resultaten bidrar till det övergripande målet om kortare omställningstid av äldre styrsystem. Genom att automatisera den strukturella konverteringsfasen reduceras manuell insats för enklare objekt väsentligt, och de genererade XML-filerna och loggarna ger ingenjörer ett tydligt och dokumenterat startläge för den manuella anpassning som krävs för mer komplexa objekt.

## 7.4 Etiska aspekter

Sveriges Ingenjörers hederskodex betonar att “Ingenjören bör i sin yrkesutövning känna ett personligt ansvar för att tekniken används på ett sätt som gagnar människa, miljö och samhälle”. I detta examensarbete innebär det att den framtagna verktygskedjan ska ses som ett stöd i ingenjörsarbetet, inte som en ersättning för teknisk granskning och omdöme [22].

I hederskodexen nämns att “Ingenjören bör sträva efter att förbättra tekniken och det tekniska kunnandet i riktning mot ett effektivare resursutnyttjande utan skadeverkningar”. Genom att automatisera delar av konverteringsarbetet kan verktygen minska repetitiv manuell konfiguration och frigöra tid för analys, verifiering och kvalitetsarbete, under förutsättning att resultaten alltid granskas inom ordinarie projektprocesser.

Under arbetet användes generativ AI som stöd vid programmering, felsökning och prototypframtagning. Användningen ligger i linje med hederskodexens krav på personligt ansvar, effektivt resursutnyttjande och saklighet, genom att AI-stödet behandlades som ett hjälpmedel medan det tekniska innehållet och de slutliga formuleringarna alltid granskades och ansvarsmässigt bedömdes av författaren.

Hederskodexen betonar också respekt för konfidentiell information. Examensarbetet har utförts i Granitor Systems interna miljö, och rapporten innehåller endast generiska exempel utan kunds specifika data eller anläggningsuppgifter.

## 7.5 Framtida utvecklingsmöjligheter

Den nuvarande lösningen är en proof of concept och ett naturligt nästa steg är att bredda och fördjupa den på flera sätt.

Parsern täcker de vanligaste SattLine-konstruktionerna men saknar stöd bland annat för komplexa control moduler, kommunikationsblock och grafiska metadata. Utökad parsningsstöd för dessa delar skulle öka täckningen avsevärt. Kopplingen till externa bibliotek som *NucleusLib* och *EventLib* hanteras manuellt i dag. En beroendeanalys som kartlägger och flaggar typberoenden skulle göra konverteringsresultaten mer direkt användbara.

På verktygs- och integrationssidan finns flera möjligheter. GUI-verktygen kan utökas med en projektöversikt som visar beroendegraf, konverteringsstatus per objekt och möjlighet till selektiv körning. En djupare integration mot 800xA Engineering Studio och Asset Library skulle möjliggöra batchimport med versionskontroll.

På längre sikt är den grundläggande arkitekturen generaliserbar: samma princip om att parsra ett textbaserat källformat och serialisera till ett väldefinierat import-API kan tillämpas på migration från andra äldre DCS-plattformar, vilket skulle vidga verktygets användningsområde väsentligt. Ett exempel på detta kan vara migrering av Siemens PCS7 till 800xA.



## 8 Referenser

- [1] “Om Granitor Systems”, *www.granitor.se*, 2026. [Online]. URL: [Om Granitor Systems - Granitor](#) (Hämtad: 26/01/2026).
- [2] “ABB’s Heritage Brand”, *www.abb.com*, 2026. [Online]. URL: [ABB's heritage brand - Alfa Laval Automation](#) (Hämtad: 26/01/2026).
- [3] “Industrial Process Automation Systems: Design and Implementation”, B.R. Mehta och Y. Jaganmohan Reddy, Elsevier, 2014. [Bok]. ISBN: 9780128009390. (Hämtad: 04/06/2026)
- [4] “SattLine - SattLine Language”, ABB, 2021. [PDF]. (Doc. No. 3BSE024811 311) (Hämtad: 26/02/2026).
- [5] “SattLine - General and Graphics”, ABB, 2021. [PDF]. (Doc. No. 3BSE024810 311) (Hämtad: 26/02/2026).
- [6] “SattLine – I/O and Communication”, ABB, 2021. [PDF]. (Doc. No. 3BSE024812 311) (Hämtad: 26/02/2026).
- [7] “SattLine - Start Package”, ABB, 2021. [PDF]. (Doc. No. 3BSE024819 311) (Hämtad: 26/02/2026).
- [8] “Livscykelanalys”, *new.abb.com*, 2026. [Online]. URL: [Livscykelanalys](#) (Hämtad: 26/02/2026).
- [9] “Literature Review and Model Proposal on the Machine Life Cycle in Industrial Automation from Different Perspectives”, V. Stegmaier, T. Eberhardt, W. Schaaf, N. Jazdi, M. Weyrich och A. Verl, *Procedia CIRP*, vol. 120, pp. 690-695, 2023. [Artikel]. DOI: 10.1016/j.procir.2023.09.060. (Hämtad: 04/06/2026).
- [10] “System 800xA for Satt Products and Systems”, *library.e.abb.com*, 2006. [Online]. URL: [Industrial System 800xA, Satt Products and Systems](#) (Hämtad: 11/02/2026).
- [11] “ABB:s automationssystem 800xA”, *new.abb.com*, 2026. [Online]. URL: [ABB:s automationssystem 800xA](#) (Hämtad: 10/02/2026).
- [12] “System 800xA Control - AC 800M - Getting Started”, *library.e.abb.com*, 2026. [Online]. [System 800xA Control](#) (Hämtad: 11/02/2026).
- [13] “ABB Ability™ System 800xA - System Guide”, ABB, 2022. [PDF]. (Doc. No. 3BSE041434 611) (Hämtad: 26/02/2026).
- [14] “Overview of the IEC 61131 Standard”, *library.e.abb.com*, 2026. [Online]. URL: [Overview of the IEC 61131 Standard](#) (Hämtad: 11/02/2026).
- [15] “Control Builder Open Interface”, ABB, 2023. [PDF]. (Doc. No. 2PAA123770) (Hämtad: 26/02/2026).
- [16] “Northern European sugar producer refines process with ABB's Ability System 800xA Upgrade”, *new.abb.com*, 2026. [Online]. URL: [Northern European sugar producer refines processes with ABB's Ability™ System 800xA Upgrade](#) (Hämtad: 11/02/2026).
- [17] “The “whey” to efficiency”, *new.abb.com*, 2026. [Online]. URL: [The “whey” to efficiency - Seamless technology switchover to System 800xA at Bregott factory](#) (Hämtad: 11/02/2026).
- [18] “Upgrade from ABB SattLine to ABB System 800xA”, *batenburg-industrialautomation.com/en*, 2026. [Online]. URL: [Upgrade from ABB SattLine to ABB System 800xA | Olam Cocoa](#) (Hämtad: 11/02/2026).
- [19] “Styrssystemutbyte från Uniview till ABB:s 800xA”, R. Abboud och R. Muhammad, TEIE-3139, Lunds universitet, LTH, 2025. [Online]. URL: <https://lup.lub.lu.se/student-papers/record/9206029> (Hämtad: 03/06/2026).

- [20] “Practice Standard for Work Breakdown Structures”, Project Management Institute, 3rd ed., 2019. [Bok]. ISBN: 9781628256192. (Hämtad: 04/06/2026).
- [21] “What is a P&ID diagram?”, *creately.com*, 2025. [Online]. URL: [What is P&ID Diagram](#) (Hämtad: 11/02/2026).
- [22] “Hederskodex”, [www.sverigesingenjorer.se](http://www.sverigesingenjorer.se), 2026, [Online]. URL: [Vår hederskodex - Sveriges Ingenjörer](#) (Hämtad: 04/05/2025)

# Appendix A – Viktiga kod bitar

## Typmappning mellan SattLine och 800xA

```
TYPE_MAP = {
    "boolean":      ("bool",    "System.bool"),
    "integer":      ("dint",    "System.dint"),
    "real":         ("real",    "System.real"),
    "string":       ("string",  "System.string"),
    "tagstring":    ("string",  "System.string"),
    "identstring":  ("string",  "System.string"),
    "linestring":   ("string",  "System.string"),
    "maxstring":    ("string",  "System.string"),
    "time":         ("Time",    "System.Time"),
    "timer":        ("Timer",   "System.Timer"),
}

def map_type(t: str):
    k = t.strip().lower()
    return TYPE_MAP.get(k, (t.strip(), t.strip()))

_STARTATTR_MAP = {
    "const": "constant",
    "state": "retain",
    "cold":  "coldretain",
}
```

Kärna i datatypsmappningen. SattLine-typer och startattribut översätts till namn och *TypePath* som senare används i CB Open Interface XML.

## Parsning av fält och *RECORD*-definitioner

```
def parse_fields(text: str) -> List[FieldDecl]:
    flat = re.sub(r'\\(\\*\\.\\*?\\*\\)', ' ', text, flags=re.DOTALL)
    flat = re.sub(r'\\s+', ' ', flat.strip())
    results = []
    for raw in (f.strip() for f in flat.split(';') if f.strip()):
        m = re.match(
            r'([\\w][\\w\\s,]*?)'
            r'(?:\\s+(Const|State|OpSave|Cold|Secure))?'
            r'(?:\\s+"([^"]*)"?)?'
            r'(?:\\s+(Const|State|OpSave|Cold|Secure))?'
            r'\\s*:\\s*(\\w+)'
            r'(?:\\s+(Const|State|OpSave|Cold|Secure))?'
            r'(?:\\s*:=\\s*(.+?))?$' ,
            raw.strip()
        )
        if not m:
            continue
        names = [n.strip() for n in m.group(1).split(',') if n.strip()]
        stAttr = m.group(2) or m.group(4) or m.group(6) or ""
        comment = m.group(3) or ""
        direction, desc = _dir_desc(comment)
        default = (m.group(7) or "").strip().strip('\"')
        results.append(FieldDecl(
            names=names, type_name=m.group(5),
            direction=direction, description=desc,
            default_value=default, StartAttr=stAttr,
        ))
    return results

def parse_records(text: str) -> List[RecordDef]:
    pat = re.compile(
        r'^[ \\t]*(\\w+)'
        r'(?:'
        r'[ \\t]+(PRIVATE|INTERNAL)[ \\t]*'
        r'|(?:[ \\t]+[\\^\\n]*?)?[ \\t]*()'
        r')'
        r'RECORD\\s+DateCode_?\\s*(\\d+)'
        r'[ \\t]*\\n(.*?)ENDDEF',
        re.MULTILINE | re.DOTALL)
    records = []
    for m in pat.finditer(text):
        if m.group(1) in _RECORD_SKIP:
            continue
        vis = m.group(2) or "PUBLIC"
        dc = m.group(4)
        body = m.group(5)
        records.append(RecordDef(
            name=m.group(1), datecode=dc,
            fields=parse_fields(body), visibility=vis.upper(),
        ))
    return records
```

Parser för SattLine RECORD. Här bryts textformatet ned till interna datastrukturer som senare kan emitteras som *<DataType>* i XML

## Bygga *ControlModuleType* från SattLine-modul

```
def _build_cmt_el(mod: ModuleDef):
    cmt = ET.Element("ControlModuleType")
    cmt.set("xmlns", NS)
    cmt.set("Name", mod.name)
    cmt.set("Protected", "0"); cmt.set("Hidden", "0")
    cmt.set("Scope", "public"); cmt.set("InteractionWindow", "")
    cmt.set("AlarmOwner", "0"); cmt.set("BatchObject", "")
    cmt.set("SILLevel", "NonSIL"); cmt.set("SimulationMark", "0")

    if mod.parameters:
        p = ET.SubElement(cmt, "CMPParameters")
        for fd in mod.parameters:
            _mk_cm_param(p, fd)
    if mod.variables:
        v = ET.SubElement(cmt, "Variables")
        for fd in mod.variables:
            _mk_cm_var(v, fd)

    ET.SubElement(cmt, "ExternalVariables")
    ET.SubElement(cmt, "FunctionBlocks")

    logic_subs = [
        s for s in mod.submodules
        if not (s.module_type in _GRAPHIC_HINTS and not s.parameters
and not s.variables)
    ]
    if logic_subs:
        cm_el = ET.SubElement(cmt, "ControlModules")
        for sub in logic_subs:
            el = ET.SubElement(cm_el, "ControlModule", {
                "Name": sub.name,
                "Type": sub.module_type or sub.name,
                "TaskConnection": "",
                "VisibilityInGraphics": sub.visibility_in_graphics,
            })
            cm_conns = ET.SubElement(el, "CMConnections")
            for conn_name, actual in sub.connections.items():
                ET.SubElement(cm_conns, "CMConnection", {
                    "Name": conn_name,
                    "ActualParameter":
                        _normalize_actual_parameter(actual),
                    "GraphicalConnection": "0",
                })

    if mod.equation_blocks:
        cbs = ET.SubElement(cmt, "CodeBlocks")
        for eb in mod.equation_blocks:
            cb_el = ET.SubElement(cbs, "STCodeBlock")
            cb_el.set("Name", eb.block_name)
            st_el = ET.SubElement(cb_el, "ST_Code")
            _set_code_node_text(st_el,
                _wrap_equation_block_code(eb.code))
```

Här visas hur parametrar, variabler, submoduler och EQUATIONBLOCK byggs om till en *ControlModuleType* med *CMPParameters*, *Variables*, *ControlModules* och *ST\_Code*.

## Sammanställd XML-emitter

```
def emit_individual_xml(records, modules) -> Tuple[list, list]:
    ET.register_namespace("", NS)
    dt_xmls, cmt_xmls = [], []
    for rec in records:
        el = ET.Element("DataType")
        el.set("xmlns", NS)
        el.set("Name", rec.name)
        el.set("Protected", "0")
        el.set("Hidden", "1" if rec.visibility in ("PRIVATE",
"INTERNAL") else "0")
        el.set("Scope", "private" if rec.visibility in ("PRIVATE",
"INTERNAL") else "public")
        comps = ET.SubElement(el, "Components")
        for fd in rec.fields:
            _mk_comp(comps, fd)
        ET.SubElement(el, "Description").text = (
            "From SattLine RECORD '" + rec.name +
            "' (DateCode " + rec.datecode + ") [" + rec.visibility +
            "]"")
        dt_xmls.append((rec.name, _obj_xml(el)))
    for mod in modules:
        cmt, nested = _build_cmt_el(mod)
        cmt_xmls.append((mod.name, _obj_xml(cmt)))
        for nname, nel in nested:
            cmt_xmls.append((nname, _obj_xml(nel)))
    return dt_xmls, cmt_xmls

def emit_xml(records, modules) -> str:
    ET.register_namespace("", NS)
    root = ET.Element("SattLineConversion")
    root.set("xmlns", NS)
    if records:
        dts = ET.SubElement(root, "DataTypes")
        ...
    if modules:
        cmts = ET.SubElement(root, "ControlModuleTypes")
        ...
    rough = ET.tostring(root, encoding="unicode")
    dom = minidom.parseString(rough)
    pretty = dom.toprettyxml(indent="  ")
    lines = pretty.split('\n')
    lines[0] = '<?xml version="1.0" encoding="UTF-16"?>'
    xml = '\n'.join(l for l in lines if l.strip())
    return _format_multiline_code_tags(xml)
```

Här skapas både review-filen SattLineConversion och separata XML-filer per objekt. Det är denna del som kopplar parsern till faktiska filer i XML - Output.

## Ladda CB Open Interface och skapa COM-objekt

```
def ensure_supported_process_bitness() -> None:
    bits = struct.calcsize("P") * 8
    if bits != 32:
        raise RuntimeError(
            "This script must run in 32-bit Python because Control
Builder Open Interface is x86. "
            "Run with a 32-bit interpreter, for example: py -3.12-32
cbopeninterface_cli.py"
        )

def load_interop_module(interop_dll_path: str):
    if not os.path.exists(interop_dll_path):
        raise FileNotFoundError(f"Interop.CONTROLBUILDERLib.dll not
found: {interop_dll_path}")

    clr_module, _, _, _ = get_dotnet_types()
    clr_module.AddReference(interop_dll_path)
    import CONTROLBUILDERLib
    return CONTROLBUILDERLib

def create_cbopenif(controlbuilderlib):
    errors: list[str] = []
    _, Activator, Guid, Type = get_dotnet_types()

    try:
        return
Activator.CreateInstance(controlbuilderlib.CBOpenIFClass)
    except Exception as ex:
        errors.append(f"Activator(CBOpenIFClass): {type(ex).__name__}:
{ex}")

    for prog_id in (
        "CONTROLBUILDERLib.CBOpenIFClass",
        "CONTROLBUILDERLib.CBOpenIF",
        "CBOpenIF.CBOpenIFClass",
    ):
        try:
            com_type = Type.GetTypeFromProgID(prog_id)
            if com_type is None:
                errors.append(f"ProgID {prog_id}: not registered")
                continue
            return Activator.CreateInstance(com_type)
        except Exception as ex:
            errors.append(f"ProgID {prog_id}: {type(ex).__name__}:
{ex}")
```

Detta är bryggan mellan Python och Control Builder-miljön. Den verifierar 32-bitars Python, laddar Interop.CONTROLBUILDERLib.dll och skapar CBOpenIF för senare metदानrop.

## Validering och sortering av importbara CB Open Interface-objekt

```
def parse_and_validate_cbopenif_imports(xml_text: str) ->
tuple[list[tuple[str, str, str]], list[str]]:
    errors: list[str] = []
    xml_text = normalize_xml_quotes(xml_text).strip()
    if not xml_text:
        return [], ["XML file is empty."]

    try:
        root = ET.fromstring(xml_text)
    except ET.ParseError as ex:
        return [], [f"Invalid XML: {ex}"]

    elements = extract_importable_cbopenif_elements(root)
    if not elements:
        errors.append(
            "No importable CB Open Interface objects found. Supported
objects are "
            "DataType, FunctionBlockType, and ControlModuleType."
        )
        return [], errors

    imports: list[tuple[str, str, str]] = []
    for element in elements:
        sanitized_element = sanitize_cbopenif_element(element)
        element_errors = validate_cbopenif_element(sanitized_element)
        errors.extend(element_errors)
        if element_errors:
            continue

        tag_name = local_name(sanitized_element.tag)
        object_name = sanitized_element.attrib["Name"].strip()
        xml_block = serialize_cbopenif_element(sanitized_element)
        imports.append((tag_name, object_name, xml_block))

    imports.sort(key=lambda item: (IMPORT_PRIORITY_BY_TAG[item[0]],
item[1].lower()))
    return imports, errors

def import_cbopenif_object(cb, object_type: str, object_name: str,
target_name: str, xml_text: str) -> str:
    method_name = IMPORT_METHOD_BY_TAG[object_type]
    method = getattr(cb, method_name, None)
    if method is None:
        raise RuntimeError(f"{method_name} is unavailable in this Open
Interface installation.")
    return method(object_name, target_name, xml_text)
```

Här visas sista steget före import: XML rensas, valideras, plockas ut som *DataType/FunctionBlockType/ControlModuleType*, sorteras i rätt ordning och sedan skickas till CB Open Interface-metod.

## GUI-flöde för konvertering av SattLine-fil till XML

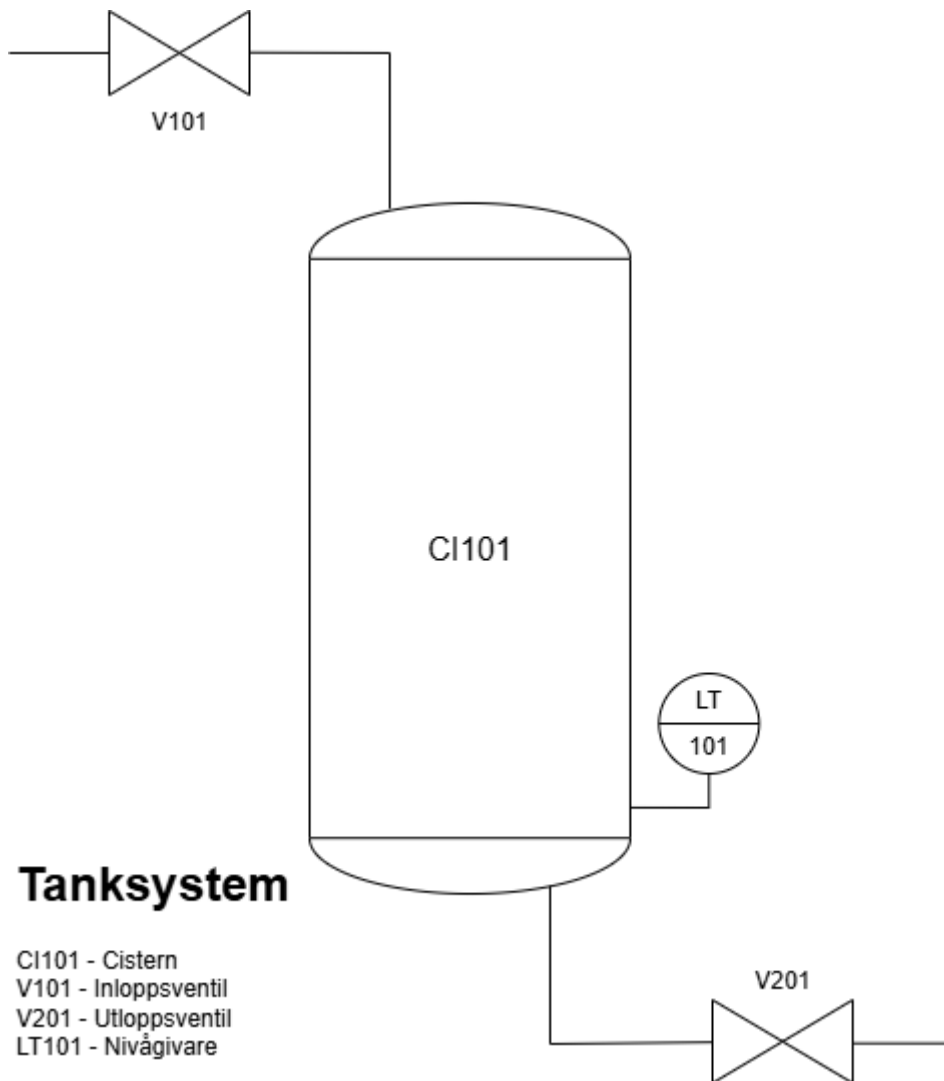
```
def _run_conversion(self):
    if not self.files:
        messagebox.showwarning("No files", "Add at least one file
before running the conversion.")
        return
    if not PARSE_AVAILABLE:
        messagebox.showerror(
            "Parser missing",
            "sattline_parser.py was not found.\n"
            "Place the GUI file in the same folder as the parser.",
        )
        return
    self.run_btn.config(state="disabled", text="Converting...")
    threading.Thread(target=self._do_conversion, daemon=True).start()

def _do_conversion(self):
    ...
    content = sattline_parser.strip_file_header(raw)
    content = sattline_parser.strip_base_picture(content)
    records = sattline_parser.parse_records(content)
    modules = sattline_parser.parse_modules(content)
    eq_blocks = sattline_parser.parse_equation_blocks(content, modules)
    seq_blocks = sattline_parser.parse_sequence_blocks(content,
modules)
    sattline_parser.attach_equation_blocks(modules, eq_blocks)
    sattline_parser.attach_sequence_blocks(modules, seq_blocks)
    ...
    xml_str = sattline_parser.emit_xml(records, modules)
    with open(out_xml, "w", encoding="utf-16") as handle:
        handle.write(xml_str)
```

Detta är arbetsflödet i sattline\_gui.py. Användaren väljer filer, sedan körs hela kedjan: preprocess -> parser -> koppla EQUATIONBLOCK/SEQUENCE -> generera XML -> spara review-fil.



## Appendix B – P&ID



Figur B1: P&ID över tanksystem



# Appendix C – CB Open Interface miljö

## Systemkrav

- Windows-maskin med Control Builder installerad, version 6.2 eller kompatibel
- CB Open Interface registreras automatiskt som en del av Control Builder-installationen
- XML-schemat *CBOpenIFSchema30.xsd* installeras i systemkatalogen (C:\...\32 eller C:\...\64 beroende på Windows-bitversion) av installationsprogrammet
- Python 32-bitarsversion krävs pga. COM-runtime-arkitektur

## Installation och utvecklingsmiljö av Python

### Steg 1 – Installera 32-bitars Python

Ladda ner och installera en 32-bitarsversion av Python 3.x från python.org. Vid installation på Windows kan man använda *py-launchern* för att hantera flera Python-versioner parallellt.

Verifiera att rätt version är installerad:

```
>py -3.12-32 --version
```

### Steg 2 – Installera pythonnet

```
>py -3.12-32 -m pip install pythonnet
```

*pythonnet* exponerar .NET CLR-funktionalitet via modulen *clr*, vilket gör det möjligt att ladda .NET-assemblies och arbeta med COM-objekt direkt från Python.

### Steg 3 – Verifiera Interop-DLL-sökvägen

Skriptet refererar till konstanten `INTEROP_DLL_PATH` som pekar på `Interop.CONTROLBUILDERLib.dll`. Standardsökvägen är:

```
C:\Program Files (x86)\ABB Control Systems\Engineering\Compact Control  
Builder AC 800M\Bin\IEC61850 Import Wizard\Interop.CONTROLBUILDERLib.dll
```

Kontrollera att filsökväg stämmer, kan variera från maskin till maskin. Denna DLL är det .NET-interop-assembly som wrapperar COM-typbiblioteket `CONTROLBUILDERLib`, vilket ger åtkomst till klassen `CBOpenIF`.

### Steg 4 – Öppna ett projekt i Control Builder

Starta Control Builder AC 800M och öppna det projekt som ska importeras till. Skriptet ansluter till det redan öppna projektet alltså det öppnar inte ett projekt automatiskt. Control Builder måste vara i offline-läge; i online- eller testläge är New- och Set-metoderna blockerade och returnerar ett felmeddelande.

### Steg 5 – Kör python program

```
py -3.12-32 example.py
```

eller dubbelklicka på Python programmen.