TEKNILLINEN KORKEAKOULU
Elektroniikan, tietoliikenteen ja automaation tiedekunta

Jarkko Kääriäinen

OHJELMAKIRJASTON HYÖDYNTÄMINEN AUTOMAATIOJÄRJESTELMÄSSÄ

Diplomityö, joka on jätetty opinnäytteenä tarkastettavaksi diplomi-insinöörin tutkintoa varten Espoossa 22.9.2008

Työn valvoja:

Prof. Raimo Sepponen

Työn ohjaaja:

DI Suvi Hyyryläinen

TEKNILLINEN KORKEAKOULU

DIPLOMITYÖN TIIVISTELMÄ

Tekijä: Jarkko Kääriäinen

Työn nimi: Ohjelmakirjaston hyödyntäminen automaatiojärjestelmässä

Päivämäärä: 22.9.2008 Kieli: Suomi Sivumäärä: 6+63

Tiedekunta: Elektroniikan, tietoliikenteen ja automaation tiedekunta

Professuuri: Sovellettu elektroniikka Koodi: S-66

Valvoja: Prof. Raimo Sepponen

Ohjaaja: DI Suvi Hyyryläinen

Tässä diplomityössä työssä tarkastellaan automaatiosovelluksen rakennetta ja elinkaarta ja tutustutaan logiikkaohjelmointiin liittyvään standardiin IEC 61131. Lisäksi tutustutaan sovelluksen laatuun vaikuttaviin seikkoihin. Työssä tutkitaan, kuinka ohjelmointia voidaan tehostaa kirjastoja käyttämällä ja mitä vaatimuksia se asettaa ohjelmoinnille. Työssä kartoitetaan kahden PLC-kehitysympäristön, Step 7:n ja Unity Pro:n, ominaisuuksia ja luodaan näihin ympäristöihin yhtenäiset ohjelmakirjastot sekä nimeämiskäytäntö ohjelmoinnissa käytettäville symboleille.

Avainsanat: automaatiojärjestelmä, ohjelmoitava logiikka, ohjelmakirjasto, IEC 61131, toimilohko, ohjelmiston laatu, Step 7, Unity Pro

Author: Jarkko Kääriäinen

Title: Using software library in an automation system

Date: 22.9.2008 Language: Finnish Number of pages: 6+63

Faculty: Faculty of Electronics, Communications and Automation

Professorship: Applied electronics Code: S-66

Supervisor: Prof. Raimo Sepponen

Instructor: M.Sc.(Tech.) Suvi Hyyryläinen

This thesis examines the structure and the life cycle of an automation application. The IEC 61131 standard conserning programmable logic controllers is also introduced along with the quality factors of the automation application. The thesis examines how software library can help the programming task and what requirements it sets for the code. The thesis surveys the features of two PLC development environments, Step 7 and Unity Pro. Unified software libraries and naming convention for the symbols used in programming are created for these environments.

Keywords: automation system, programmable logic controller, software library, IEC 61131, funtion block, software quality, Step 7, Unity Pro

Esipuhe

Haluan kiittää Professori Raimo Sepposta ja ohjaajaani Suvi Hyyryläistä hyvästä ohjauksesta. Lisäksi haluan kiittää kaikkia, jotka auttoivat minua työssäni.

Mikkeli, 22.9.2008

Jarkko Kääriäinen

Sisältö

Lyhenteet

ANSI	American National Standards Institute
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
CAMX	Computer Aided manufacturing using XML
DB	Data Block
DCOM	Distributed Component Object Model
DFB	Derived Function Block
ERP	Enterprise Resource Planning
FAT	Factory Acceptance Test
FB	Function Block
FBD	Function Block Diagram
FC	Function
HTTP	Hypertext Transfer Protocol
I/O	Input/Output
IEC	International Electrotechnical Committee
IEEE	Institute of Electrical and Electronics Engineers
IL	Instruction List
IP	Internet Protocol
IPC	Association Connecting Electronic Industries
ISA	The Instrumentation, Systems, and Automation Society
ISO	Industry Standards Organization
ISO	Industry Standards Organization
LAD	Ladder Diagram
MES	Manufacturing Executing System
OB	Organization Block
OMAC	Open Modular Architecture Controls
OPC	OLE For Process Control
OPC A&E	OPC Alarms and Events
OPC DA	OPC Data Access
OPC DX	OPC Data eXchange
OPC HDA	OPC Historical Data Access
OPC UA	OPC Unified Architecture
PI	Prosessi-instrumentointi
PID	Proportional/Integral/Derivative
PLC	Programmable Logic Controller
POU	Program Organisation Unit
SA/SD	Structured Analysis / Structured Design
SCADA	Supervisory Control And Data Acquisition
SFC	Sequential Function Chart
SOAP	Simple Object Access Protocol
ST	Stuctured Text
TCP	Transmission Control Protocol
UML	Unified Modeling Language
XML	Extensible Markup Language

1 Johdanto

Tämä diplomityö sai alkunsa, kun mikkeliläinen automaatioalan yritys Mipro Oy katsoi tarpeelliseksi kehittää mm. vesi- ja lämpölaitosprojekteissa käyttämäänsä logiikkaohjelmointiprosessiaan. Tavoitteena olisi yhtenäistää ohjelmoinnissa käytettäviä menetelmiä sekä tätä kautta parantaa tuotettavien sovellusten laatua. Aiemmin Mipron automaatioprojekteissa tehdyt sovellukset ovat olleet pienempiä ja monesti yksi ohjelmoija on projektikohtaisesti vastannut koko sovelluksen luomisesta. Ajan myötä projektien vaatimat sovellukset ovat kasvaneet, minkä lisäksi vuosittain toteutettavien projektien määrä on kasvanut. Samalla projektit olisi kuitenkin pystyttävä toteuttamaan nopeassa aikataulussa. Resurssipulaa on paikattu palkkaamalla lisää työntekijöitä. Lisäresursseista ei kuitenkaan välttämättä aina ole saatu maksimaalista hyötyä, koska yhtenäinen ohjelmointitapa on puuttunut ja jo olemassa olevaa, aiemmissa projekteissa luotua ohjelmakoodia on hyödynnetty vain jossain määrin.

Jotta valmiista ohjelmakoodista saataisiin maksimaalinen hyöty ja jottei jokaisessa uudessa projektissa tarvitsisi luoda uudestaan toiminnallisuuksia, jotka ovat jo olemassa, havaittiin Mipro:lla tarpeelliseksi luoda ohjelmakirjasto, johon jo aiemmin toteutetut ja hyödylliset ohjelmamoduulit koottaisiin. Koska Mipro käyttää projekteissaan useamman valmistajan ohjelmoitavia logiikoita ja kehitysympäristöjä, tulisi kirjaston olla sellainen, että se soveltuisi mahdollisuuksien rajoissa kaikille kehitysympäristöille tai ainakin useimmin käytetyille. Toisin sanoen kirjastosta tulisi löytyä samat toiminnallisuudet sisältävät moduulit eri ympäristöihin.

Tässä työssä pyritään selvittämään, mitä seikkoja täytyy ottaa huomioon, kun ohjelmakoodia varastoidaan kirjastoon. Lisäksi tarkastellaan muita menetelmiä, joilla ohjelmointiprosessia saataisiin edelleen optimoitua. Lopuksi pyritään arvioimaan, onko kehitetyistä menetelmistä konkreettista hyötyä ohjelmoinnissa.

Ensimmäisessä osiossa tarkastellaan automaatiosovelluksen rakennetta ja elinkaarta, mm. eri arkkitehtuureja ja sovelluksen suunnittelua. Samalla tutustutaan logiikkaohjelmointiin liittyvään standardiin IEC-61131 ja mitä se tuo mukanaan sovelluskehitykseen. Osiossa tutustutaan myös laatuun sekä yleisesti että automaatiojärjestelmän kannalta. Tämän lisäksi tarkastellaan, kuinka laatua on mahdollista parantaa testauksella.

Toisessa osiossa kartoitetaan Mipron käyttämien kehitysympäristöjen Step 7:n ja Unity Pro:n ominaisuuksia. Osiossa mm. selvitetään, miten ne tukevat ohjelmakoodin varastointia kirjastoon ja ovatko ympäristöt standardin mukaisia. Samalla pyritään selvittämään, kuinka ohjelmointia voidaan tehostaa kirjastoja käyttämällä ja mitä vaatimuksia se asettaa ohjelmoinnille. Työssä luodaan ohjelmakirjastot Step 7 ja Unity Pro ympäristöihin.

2 Nykyaikainen ohjelmoitava logiikka ja standardi IEC 61131

Mikropiirien nopea kehitys on mahdollistanut yhä monipuolisempien ja edullisempien logiikoiden valmistamisen. Nykyaikaisen logiikan varsin kattavat ominaisuudet mahdollistavat logiikoiden käytön eri sovelluksissa. Aiemmin eri valmistajien kehitysympäristöt saattoivat erota merkittävästi toisistaan. Eri ympäristöt tukivat mahdollisesti eri ohjelmointikieliä ja dataa käsiteltiin eri tavalla eri ympäristöissä. Lisäksi valmistajat lisäsivät jatkuvasti ominaisuuksia kehitysympäristöihin, mikä lisäsi jatkokoulutuksen tarvetta. Ohjelmoijalle usean täysin erilaisen kehitysympäristön hallitseminen oli vaivalloista ja aika vievää. Ohjelman tai edes sen osan siirtäminen ympäristöstä toiseen saattoi olla hankalaa tai kokonaan mahdotonta. Siten myöskin koodin uudelleenkäyttö oli vaivalloista. [John ja Tiegelkamp, 2001]

Toisistaan merkittävästi eroavien kehitysympäristöjen käyttö eri projekteissa vaatii aikaa ja rahaa. Jatkuva tarve kustannusten minimoimiseksi voidaan saavuttaa standardoinnilla ja sen tarjoamilla eduilla. Ohjelmoitavia logiikoita käsittelee kansainvälinen standardi IEC 61131. Standardi pyrkii mm. yhtenäistämään logiikoiden ohjelmoinnissa käytettäviä menetelmiä. Näin siirtyminen kehitysympäristöstä toiseen olisi helpompaa ja ohjelmointi olisi tuottoisampaa.

2.1 Standardi IEC 61131

Kansainvälinen standardi IEC 61131 Programmable controllers kuvaa nykyaikaisen ohjelmoitavan ohjauslaitteen toiminnan. Ohjelmoitaville ohjauslaitteille on useasti pyritty luomaan standardeja, mutta vasta IEC 61131 on saanut tarvittavan kansainvälisen ja teollisuuden hyväksynnän. Ensimmäinen työryhmä perustettiin työstämään standardia vuonna 1977 ja sen ensimmäiset osat standardoitiin vuonna 1990. Standardi jakautuu seuraaviin osiin [IEC, 2003]:

- IEC 61131-1 – Programmable controllers - Part 1: General information (Ohjelmoitavat ohjauslaitteet - Osa 1: Yleisinformaatio)
- IEC 61131-2 – Programmable controllers - Part 2: Equipment requirements and tests (Ohjelmoitavat ohjauslaitteet - Osa 2: Laitevaatimukset)
- IEC 61131-3 – Programmable controllers - Part 3: Programming languages (Ohjelmoitavat ohjauslaitteet - Osa 3: Ohjelmointikielet)
- IEC/TR3 61131-4 – Programmable controllers - Part 4: User guidelines (Ohjelmoitavat ohjauslaitteet - Osa 4: Käyttäjän ohjeita. Standardi sisältää ohjeita ja kuvauksia)
- IEC 61131-5 – Programmable controllers - Part 5: Communications (Ohjelmoitavat ohjauslaitteet - Osa 5: Tiedonsiirtoliikennöinti)

- IEC 61131-7 – Programmable controllers - Part 7: Fuzzy control programming (Ohjelmoitavat ohjauslaitteet - Osa 7: Fuzzy-ohjauksien ohjelmointi)
- IEC/TR 61131-8 – Programmable controllers - Part 8: Guidelines for the application and implementation of programming languages (Ohjelmoitavat ohjauslaitteet- Osa 8: Ohjausjärjestelmien ohjelmointikielien käyttö ja soveltaminen, IEC:n tekninen raportti)

Standardin kolmas osa (61131-3) keskittyy ohjelmointikieliin ja se voidaan nähdä enemmän ohjeena kuin joukkona tiukkoja sääntöjä [John ja Tiegelkamp, 2001].

2.2 Ohjelman rakenne ja Program Organisation Unit

Standardin IEC 61131 mukaan ohjelma koostuu yksittäisistä itsenäisistä osista, joita kutsutaan Program Organisation Unit:eiksi (POU), joita on kolmea tyyppiä: program (ohjelma), function (funktio) ja funtion block (toimilohko) [IEC, 2003].

Ohjelma käsittää varsinaisen pääohjelman, jossa määritellään mm. kuinka PLC:n fyysinen I/O linkitetään muuttujiin ja millä prioriteetilla ohjelmaa ajetaan [John ja Tiegelkamp, 2003]. Funktiolle voidaan antaa parametrejä ja sillä ei ole omaa muistia tilatiedolle eli se ei muista mitään tietoja edelliseltä suorituskerralta. Kutsuttaessa funktiota useasti samoilla parametreillä, palauttaa se joka kerta saman arvon.

Toimilohkolle voidaan antaa parametreja ja sillä on oma sisäinen tila ja muisti. Tämän takia toimilohkon ulostulo riippuu sekä sille annetuista parametreistä että sen sisäisistä muuttujista.

Muuttujat sisältävät POU:ssa käsiteltävän datan. Nämä muuttujat määritellään declaration part:issa, joka sijaitsee POU:n alussa [John ja Tiegelkamp, 2003]. Määritelmä kertoo mm. muuttujan nimen ja tietotyypin.

Muuttujat on julistettava POU:n alussa, jotta niitä voidaan käyttää. Samalla määritellään muuttujan tyyppi, joka kertoo mihin tarkoitukseen muuttujaa voidaan käyttää. Muuttujan tyypit on listattu taulukossa 1.

Taulukko 1: POU:n sallitut muuttujatyyppit.

	PROGRAM	FUNCTION_BLOCK	FUNCTION
VAR	*	*	*
VAR_INPUT	*	*	*
VAR_OUPUT	*	*	
VAR_IN_OUT	*	*	
VAR_EXTERNAL	*	*	
VAR_GLOBAL	*		
VAR_ACCESS	*		

Muuttujatyypeistä VAR_INPUT, VAR_OUTPUT ja VAR_IN_OUT on tarkoitettu käytettäviksi sisään- ja ulostuloparametrien yhteydessä, VAR_GLOBAL, VAR_EXTERNAL ja VAR_ACCESS ovat yleistä dataa ja pelkkä VAR, eli paikallinen muuttuja, on POU:n sisäistä dataa.

POU:ssa suoritettavat ohjelmakäskyt sijaitsevat code part:issa. Käskyjen kirjoittamiseen käytetään IEC 61131-3 standardissa määriteltyjä ohjelmointikieliä. Ohjelmointikieliin palataan tuonempana.

Merkittävä ominaisuus IEC 61131 standardissa on muuttujien käyttö informaation säilytyksessä ja välityksessä [John ja Tiegelkamp, 2003]. Aiemmin logiikkaohjelmoinnissa oli mahdollista viitata dataan ainoastaan käyttämällä suoraa muistiosoitetta. Tämä on kuitenkin hankalaa ja virhealtista varsinkin laajempien sovellusten yhteydessä. Lisäksi oli tiedettävä tarkkaan muistiosoitteen koko, ettei vahingossa kirjoittanut tai lukenut myös viereisen muistiosoitteen sisältöä.

Standardin IEC 61131 mukaan tietoa käsitellään käyttäen muuttujia, joilla on myös tietotyyppi. Muuttujan nimi yksilöi sen ja tietotyyppi kertoo mitä arvoja muuttuja voi saada. Tämä vapauttaa ohjelmoijan hankalasta muistin hallinnasta, jolloin ei tarvitse muistaa hankalia muistioosoitteita ja niiden kokoja. Sen sijaan voidaan käyttää kuvaavampia ja helpommin muistettavia muuttujien nimiä. Muuttujalla on myös tietotyyppi, kuten BOOL tai INT, ja kehitysympäristö huolehtii tarvittavan muistin varaamisesta. Eri kehitysympäristöjen tarjoamat tietotyypit saattavat erota toisistaan, eivätkä siksi ole yhteensopivia. Standardin IEC 61131 myötä käytettävät tietotyypit ovat yhdenmukaisia, jolloin sovellukset tulevat yhdenmukaisemmiksi. Taulukossa 2 on esitetty standardin mukaiset tietotyypit.

Taulukko 2: Tietotyypit standardissa 61131 [IEC, 2003].

Boolean-tyypit	Kokonaisluvut etumerkillä	Kokonaisluvut ilman etumerkkiä	Liukuluvut	Aika, kesto, pvm, merkkijono
BOOL	INT	UINT	REAL	TIME
BYTE	SINT	USINT	LREAL	DATE
WORD	DINT	UDINT		TIME_OF_DAY
DWORD	LINT	ULINT		DATE_AND_TIME
LWORD				STRING

2.3 IEC 61131-3 ohjelmointikielet

Standardi tarjoaa kolme tekstimuotoista ja kolme graafista ohjelmointikieltä, joilla sovelluksia voidaan luoda [John ja Tiegelkamp, 2003].

Tekstimuotoiset kielet ovat:

- Instruction list (IL)

- Structured text (ST)

Graafiset kielet ovat:

- Ladder diagram (LAD)
- Function Block Diagram (FBD)
- Sequential Function Chart (SFC)

Eri taustan omaavat ohjelmoivat voivat valita itselleen sopivimman ohjelmointikielen. Seuraavaksi esitellään eri ohjelmointikielet. Kielistä Instruction List ja Structured Text käydään läpi pääpiirteittäin, ja Ladder diagram, Function Block Diagram ja Sequential Function Chart syvällisemmin.

Instruction list on assembleriä muistuttava ohjelmointikieli. Se on hyvin yleiskäyttöinen ja monesti muut kielet on mahdollista kääntää Instruction list muotoon. IL on rivipohjainen kieli, eli yksi käsky kuvataan tasan yhdellä rivillä. Myös tyhjät rivit ovat sallittuja. [John ja Tiegelkamp, 2003] Liitteessä A on esitetty esimerkki IL-ohjelmalistauksesta.

Structured text on korkean tason tekstimuotoinen ohjelmointikieli, joka syntaksiltaan muistuttaa Pascal:ia. Stuctured text mahdollistaa monimutkaisen toiminnallisuuden kuvaamisen abstrakteilla lausekkeilla.

Structured text:in edut Instruction list:iin nähden ovat:

- ohjelman rakenteiden ryhmittely
- tehokkaat keinot ohjelmavuon ohjaukseen
- tiiviimpi esitystapa

ST algoritmi koostuu lausekkeista, jotka on erotettu toisistaan puolipisteellä. Toisin kuin IL:ssä, lauseke voi jatkua riviltä toiselle. Yhdellä rivillä on mahdollista olla myös useampia lausekkeita. [John ja Tiegelkamp, 2003] Liitteessä B on esitetty esimerkki ST-ohjelmalistauksesta.

Ladder diagram kieli perustuu virran kulun kuvaamiseen relelogiikkajärjestelmässä. Vaikka ladder diagram on pääasiassa tarkoitettu loogisten signaalien käsittelyyn, voidaan sillä hoitaa myös mm. aritmeettisia operaatioita. Nimensä se saanut siitä että ohjelma muistuttaa ulkoasultaan tikapuita. Ladder diagram on laajasti käytössä eri logiikoissa. Ladder Diagram sopii parhaiten melko yksinkertaisiin ohjauksiin, mutta sillä mahdollista toteuttaa hyvin monimutkaisiakin järjestelmiä.

Ladder diagrammi koostuu oikeassa ja vasemmassa reunassa olevista ns. virtakiskoista ja niitä yhdistävistä askelmista (rung). Vasemman puoleinen kisko on loogiselta tilaltaan 1 ja oikea 0. Virta kulkee vasemmalta oikealle askelmia pitkin riippuen siitä ovatko askelmille asetetut elementit johtavassa tilassa. Ohjelman suoritus etenee vasemmalta oikealle ja ylhäältä alas, myös hypyt toisaalle koodiin ovat sallittuja. [John ja Tiegelkamp, 2003] Liitteessä C on esitetty esimerkki LAD-ohjelmalistauksesta.

Function Block Diagram (FBD) on graafinen ohjelmointikieli, joka muistuttaa erilaisista integroiduista piireistä koostuvaa piirikaaviota. FBD:n lohko vastaa integroitua piiriä, jolla on tietty toiminnallisuus. Lohkoja voidaan myös yhdistää toisiinsa kuten integroituja piirejä. Yksittäinen toimilohko tai funktio koostuu sisääntuloista, itse lohkosta, joka kuvaa sen toiminnon, ja ulostuloista. Graafisessa esityksessä funktion tai toimilohkon kutsu on laatikko, jonka vasemmalle sivulle merkitään sisääntulot ja oikealle ulostulot vaakaviivoin. Laatikon sisälle on merkitty toimilohkon nimi. Funktiolla on vain yksi ulostulo (funktion palauttama arvo), mutta toimilohkolla voi olla useampi ulostulo. [John ja Tiegelkamp, 2003]

Ohjelman suoritus etenee kuten Ladder diagram:issa eli vasemmalta oikelle ja ylhäältä alas. Monissa ohjelmointiympäristöissä on mahdollista yhdistää Ladder-diagrammiin toimilohkoja.

Function Block Diagram soveltuu erittäin hyvin prosessin ohjaukseen, jossa on käsiteltävä paljon dataa ja ohjauskäskyjä. Yhdistämällä graafisesti toimilohkoja toisiinsa voidaan selkeästi esittää loogisia ja matemaattisia laskutoimituksia. Function Block Diagram mahdollistaa erittäin monimutkaisten ohjauksien toteuttamisen ja sopii siten käytettäväksi niin yksinkertaisissa kuin myös erittäin monimutkaisissa järjestelmissä. Function Block Diagram koostuu seuraavista graafisista elementeistä:

1. graafiset elementit, joilla kutsutaan funktiota tai toimilohkoa
2. liitokset
3. ohjelman suoritusta ohjaavat graafiset elementit
4. yhdyspiste (connector)

Toimilohkoa tai funktiota kutsutaan sijoittamalla sitä kuvaava laatikko halutulle paikalle ohjelmakoodin graafisessa esityksessä.

Toimilohkon formaalit parametrit, eli sisään- ja ulostulojen nimet, merkitään laatikon sisäpuolelle. Sisään- ja ulostuloihin liitettävien muuttujien ja vakioiden nimet merkitään laatikon ulkopuolelle vastaava formaalin parametrin kohdalle. Lohkon ulostulo voidaan liittää suoraan toisen lohkon sisääntuloon. Jos toimilohkoa käytetään ladder-diagrammin yhteydessä, lohkossa täytyy olla vähintään yksi kappale binäärisiä sisään- ja ulostuloja.

Ohjelmoinnissa on mahdollista käyttää lohkon suorituksen mahdollistavaa EN-parametria sisääntulona ja lohkon onnistuneesta suorituksesta indikoivaa ENO-parametria ulostulona, mutta näiden käyttö ei kuitenkaan ole pakollista.

Graafiset elementit yhdistetään Function Block Diagram:missa toisiinsa liitoksilla, jotka voivat kulke sekä vaaka- että pystysuoraan. Liitos voidaan jakaa useammaksi ja näin kopioida sama tieto useampaan paikkaan. Useampaa haaraa ei kuitenkaan voida yhdistää uudelleen, eli esimerkiksi yhteen sisääntuloon ei voida yhdistää useaa

eri tietoa. Osassa ohjelmointityökaluissa voidaan liitoksiin tehdä mutkia siten että rivin lopussa on mahdollista tehdä lenkki seuraavan rivin alkuun.

Ohjelman suoritusta voidaan ohjata paluu- ja hyppykäskyillä, jotka voivat olla joko ehdottomia tai ehdollisia. Ehdoton hyppy tai paluu suoritetaan aina, kun taas ehdollinen ainoastaan jos ehto on tosi. Paluu-käskyllä lopetetaan POU:n suoritus ja palataan kutsuneeseen POU:iin ja hyppy- käskyllä hypätään tiettyyn Network-osioon.

Yhdyspisteen avulla voidaan liian pitkäksi muodostuva verkko jakaa useammalle riville. Yhdyspiste sijoitetaan verkon oikeaan reunaan ulostuloon. Nyt ulostulo on linkitettävissä sisääntuloon uudella rivillä. Yhdyspisteen avulla voidaan rakentaa pitkiä verkkoja myös sellaisissa ohjelmointiympäristöissä, joiden rivin pituus on rajoitettu ja jotka eivät tue vieritystä vaakasuunnassa.

Ohjelma suoritetaan verkko kerrallaan, ylhäältä alaspäin. Hyppy- ja paluu- käskyillä voidaan vaikuttaa ohjelmavuon etenemiseen. Yksittäinen verkko suoritetaan seuraavien sääntöjen mukaisesti [John ja Tiegelkamp, 2001]:

1. elementin sisääntulot on suoritettava ennen itse elementtiä
2. elementin suoritus ei ole valmis ennen kuin kaikki sen ulostulot ovat valmiit
3. verkon suoritus ei ole valmis ennen kuin kaikki siinä olevien elementtien kaikki ulostulot ovat valmiit

Function Block Diagram:issa on mahdollista luoda takaisinkytkentä saman verkon sisällä. Tällöin ulostuloparametri liitetään aikaisemman elementin sisääntuloparametriksi. Tällaista parametria kutsutaan takaisinkytkentämuuttujaksi. Takaisinkytkentämuuttujalla on ensimmäisellä suorituskerralla ennalta määrätty alkuarvo, jatkossa sen muuttujan arvo riippuu edellisestä ohjelmakierroksesta. Liitteessä D on esitetty esimerkki FBD-ohjelmalistauksesta.

Sequential Function Chart (SFC) on sekä graafinen että tekstimuotoinen ohjelmointikieli. Sen avulla voidaan jakaa monimutkainen ohjelma pienempiin ja helpommin hallittaviin osiin. SFC kuvaa kuinka ohjelmavuo etenee näitten osien välillä. SFC koostuu askelmista ja siirtymistä sekä linkeistä niiden välillä. Graafinen esitystapa on selkeämpi ja siksi suositumpi kuin tekstimuotoinen. [John ja Tiegelkamp, 2003]

Siirtymät ovat ehtoja, jotka määräävät mitkä askelmat milloinkin suoritetaan. Yksittäinen askelma sisältää kuvauksen siitä mitä ohjelma tekee kyseisen prosessin vaiheen aikana. Askelman sisältämä ohjelma kirjoitetaan käyttäen neljää muuta standardin IEC 61131-3 kielistä. Askelma voi myös sisältää uuden SFC-rakenteen.

Prosessit jotka sisältävät selkeästi toisistaan poikkeavia vaiheita, soveltuvat hyvin ohjelmoitaviksi SFC:n avulla. Hyvä esimerkki tällaisesta prosessista on pesukoneen toiminta, jossa vaiheina ovat pesu, huuhtelu ja linkous.

Jos POU:n ohjelmoinnissa käyttää SFC:tä, on koko POU:n rakenteen oltava SFC:n mukainen. Funktiota ei voi kirjoittaa käyttäen SFC:tä, koska funktioon ei voi liittää

SFC:n vaatimaa tilatietoa. Tekstimuotoiseen SFC:tiin voidaan kirjoittaa kommentteja, samoin kuin muissa IEC 61131-3 kielissä, kaikkiin paikkoihin missä välilyönti on sallittu. Sen sijaan graafisessa esityksessä sallittu kommentin paikka riippuu ohjelmointiympäristöstä.

SFC:ssä on kahdenlaisia peruselementtejä, askelmia ja siirtymiä. Askelma kuvataan graafisesti suorakaiteen muotoisella laatikolla, jonka sisällä on askelman nimi. Askelmalla on kaksi sisäistä muuttujaa, jotka ovat käyttäjältä kirjoitussuojattuja systeemimuuttujia. BOOLEAN tyyppinen tilatieto ”step flag”kertoo onko askelma aktiivisessa tilassa (TRUE = aktiivinen, FALSE = ei aktiivinen). Aika joka on kulunut siitä kun askelma on aktivoitunut, saadaan muuttujasta ”elapsed time”. Jos askelma ei vielä ole ollut aktiivinen, on muuttujan arvo nolla. Jokainen SFC:llä ohjelmoitu POU sisältää askelman joka suoritetaan ensimmäiseksi kun POU:ta kutsutaan. Tämä ensimmäinen askelma merkitään graafisesti laatikon kaksoisreunuksella tai tekstimuodossa ”INITIAL_”etuliitteellä.

Askelmat yhdistetään toisiinsa siirtymillä. Siirtymä sisältää ehtolausekkeen, joka voi saada loogisen arvon TRUE tai FALSE.

Graafisessa esityksessä ehto voidaan kirjoittaa joka suoraan siirtymän viereen, yhdistää ehto siirtymään yhdyspisteen kautta tai antamalla ehdolle nimi ja kutsumalla ehtoa nimellä siirtymässä.

SFC-ohjelma koostuu yhdestä tai useammasta verkosta, jotka sisältävät askelmia ja siirtymiä. Askelma on joukko käskyjä, joita suoritetaan niin kauan kun askelma on aktiivinen. Kun askelmaa seuraavan siirtymän looginen lause muuttuu todeksi, askelma deaktivoidaan ja siirtymän jälkeinen askelma aktivoidaan. Perättäisten askelmien välillä on aina oltava siirtymä ja useampi askelma voi olla samaan aikaan aktiivinen. Liitteessä E on esitetty esimerkki SFC-ohjelmalistauksesta.

Käytännössä suosituin kieli on luultavasti Ladder. Sillä voidaan toteuttaa monimutkaisiakin sovelluksia, erityisesti jos sen yhteydessä voidaan käyttää uudelleenkäytettäviä lohkoja. Samalla se on kuitenkin suhteellisen helppotajuinen ja ohjelman toiminnan ymmärtääkseen ei välttämättä tarvitse olla ohjelmoija. Kun tarvitaan monimutkaisempia toiminnallisuuksia valitaan usein kieleksi FBD, koska se tarjoaa kattavat ominaisuudet ja sillä on selkeä luoda ja lukea koodia. Varsinkin toimilohkojen uudelleenkäyttö on sujuvaa FBD:ssä. Tässä työssä kaikki kirjastoon sijoitetut ohjelmamoduulit oli luotu käyttäen joko Ladder- tai FBD-kieltä.

Tekstimuotoisten kielien IL:n ja ST:n käyttö on jäänyt monesti vähemmälle. Niiden syntaksi on ehkä vieraampi automaatiotaustan omaavalle ohjelmoijalle, kuten sähkö- tai koneinsinöörille. Näitä tekstipohjaisia kieliä käyttää ennemmin ohjelmoija joka jo ennestään hallitsee esimerkiksi C-kielen. Koska SFC-kielen toiminnallisuus voidaan toteuttaa ainakin osittain muilla kielillä, on sen käyttö todennäköisesti harvinaisempaa.

Aiemmat kehitysympäristöt olivat täysin valmistajakohtaisia, ja niiden toteutus saattoi erota merkittävästi toisistaan. Eri ympäristöt tukivat eri ohjelmointikieliä, muuttujien käyttöä ei välttämättä tuettu, tietotyypit olivat erilaisia jne.

Standardi 61131 pyrkii yhtenäistämään eri ohjelmointiympäristöjä. mm. vakioimalla ohjelmointikielet ja käytettävät tietotyypit. Kun eri ympäristöt tukevat samoja ohjelmointikieliä ja dataa kuvataan samoilla tietotyypeillä, on ohjelmoijan huomattavasti helpompi siirtyä ympäristöstä toiseen. Samalla lohkot ja POU mahdollistaa koodin tehokkaan uudelleen käytön. Nämä seikat mahdollistavat nopeamman ja tehokkaamman sovelluskehityksen ja sovellusten helpon siirrettävyyden ympäristöstä toiseen. Edellytyksenä kuitenkin on, että eri ympäristöt ovat standardin mukaisia. Käytännössä kuitenkin eri valmistajien tuotteet noudattavat standardia eri tavalla, koska standardissa 61131 ei ole sanottu minimivaatimuksia. Luvussa 3 palataan tarkemmin Step 7 ja Unity Pro kehitysympäristöjen ominaisuuksiin.

3 Automaatiojärjestelmä

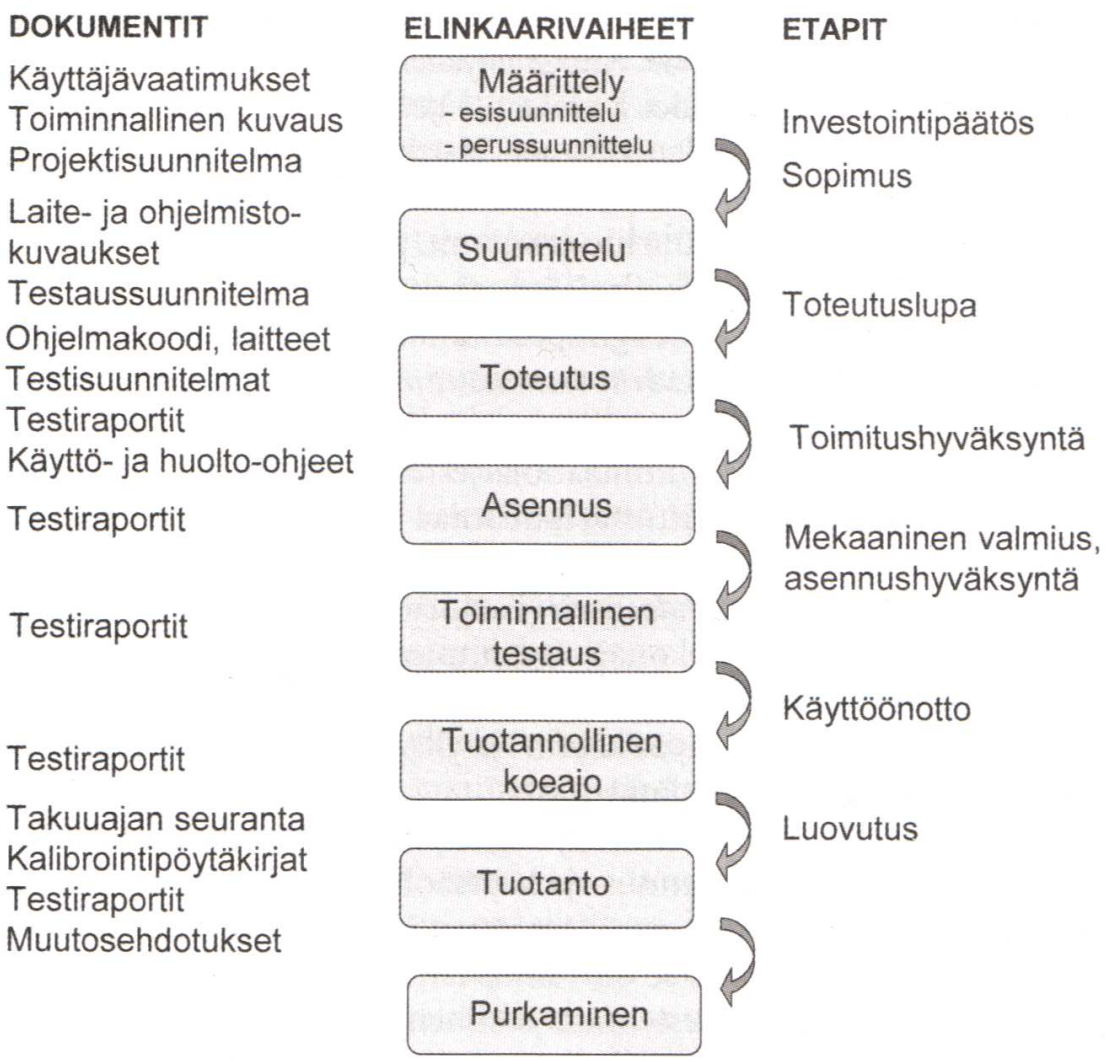
Automaatio on teknologia, jolla tarkoitetaan jonkin tehtävän tai toiminnan suorittamista itsenäisesti. Toisin sanoen jos jokin laite tai järjestelmä toimii automaattisesti, se ei tarvitse jatkuvaa ja/tai välitöntä huomiota käyttäjältä. Automaatio teknologiana käsittää mm. teoriat, menetelmät ja toteutustekniikat [SAS, 2001]. Automaatiolaitteista, kuten antureista, toimilaitteista, ohjelmoitavasta logiikasta ja ohjelmistosta koostuvaa järjestelmää kutsutaan automaatiojärjestelmäksi [SAS, 2005]. Automaatiojärjestelmä on koottu juuri jotain tiettyä sovellusta, kuten esimerkiksi tuotantolinjaa tai kemian prosessin, ohjausta varten. Järjestelmätoimittaja kokoaa järjestelmän yhdistelemällä valmiita kaupallisia tuotteita ja komponentteja. Usein pelkkä tuotteiden yhdistely ei kuitenkaan riitä, vaan tuotteita joudutaan räätälöimään sovellukseen sopiviksi tai joudutaan jopa tekemään kokonaan uusi tuote (esimerkiksi ohjelmoidaan tietoliikenneajuri, jolla eri komponentit saadaan kommunikoimaan keskenään).

3.1 Automaatiojärjestelmä ja sen elinkaari

Automaatiojärjestelmän elinkaari voidaan jakaa vaiheisiin, joita erottavat tärkeät tilanteet ja päätökset eli etapit. Automaatiojärjestelmän elinkaarimallin periaate on esitetty oheisessa kuvassa (kuva 1). Malli pohjautuu yleiseen ohjelmistotekniikan elinkaaren vaihejakoon ja se on muokattu automaatioalalle sopivaksi. [SAS, 2005]

Määrittelyvaiheen aikana kootaan järjestelmän lähtötiedot, joita tarvitaan järjestelmän suunnittelussa ja toteutuksessa. Automaatiojärjestelmä pyritään kuvaamaan käyttäjän kannalta ja siten että kuvaus on riippumaton mistään tietystä toteutustavasta.

Määrittelyvaiheen aikana on tärkeää että asiakas (eli järjestelmän tuleva käyttäjä) ja toteuttaja tekevät tiivistä yhteistyötä. Useasti määrittelyvaihe on investoinnin onnistumisen kannalta kriittinen vaihe ja monien myöhempien ongelmien syy voidaan jäljittää puutteisiin määrittelyvaiheessa [HSE, 1995]. Määrittelyvaihe jaetaan kahteen peräkkäiseen vaiheeseen: esisuunnitteluun ja perussuunnitteluun.



Kuva 1: Elinkaaren vaiheet [SAS, 2005]

Esisuunnittelun tarkoituksena on selvittää järjestelmän vaatimukset ja lisäksi pyrkiä arvioimaan investoinnista saatavia hyötyjä ja siitä aiheutuvia kustannuksia. Vaatimusten määrittely toimii koko suunnittelun perustana ja määrittelyn tuloksista syntyy käyttäjävaatimukset dokumentti (User requirement specification). Tulevan järjestelmän pääpiirteet ja kustannusarvio ovat selvillä esisuunnitteluvaiheen päättyessä. Kerätyn tiedon perusteella asiakas tekee investointipäätöksen. [SAS, 2005]

Perussuunnitteluvaiheen alussa asiakas valmistelee esisuunnittelussa syntyneen materiaalin perusteella tarjouspyynnöt haluamastaan järjestelmästä ja lähettää nämä tarjouspyynnöt potentiaalisille toimittajille.

Toimittajat puolestaan vastaavat tarjouspyyntöön tarjouksella. Tarjouksessa toimittaja esittelee ratkaisun, jolla asiakkaan vaatimukset saadaan täytettyä. Tätä kutsutaan järjestelmän toiminnalliseksi kuvaukseksi (Functional specification). Tarjous sisältää myös toimittajan tekemän projektisuunnitelman ja laatusuunnitelman.

Tutustuttuaan tarjouksiin ja vertailtuaan niitä asiakas valitsee yhden (tai useamman) toimittajan, jonka kanssa vielä tarkennetaan vaatimuksia, projektisuunnitelmia ja toteutustapaa. Lopulta saadaan aikaan toimitussopimus. [SAS, 2005]

Suunnitteluvaiheen tavoitteena on jalostaa määrittelyvaiheen tuottamaa tietoa tarpeeksi pitkälle, jotta järjestelmän toteuttaminen voidaan aloittaa. Suunnittelun perustana käytetään aiemmissa vaiheissa tuotettuja dokumentteja, toiminnallista kuvausta ja käyttäjävaatimuksia. Niiden avulla tarkennetaan teknologiavalinnat ja valitaan sopivat tuotteet käytettäväksi toimituksessa. Suunnitteluvaihe on jaettavissa järjestelmäsuunnitteluun ja toteutussuunnitteluun. [SAS, 2005]

Järjestelmäsuunnittelun aikana tarkennetaan käytettävää arkkitehtuuria, valitaan käytettävät tekniikat sekä tehdään suunnitteluvaiheen ja toteutuksen tarkempi aikataulu. Järjestelmäsuunnittelu koostuu laitteistosuunnittelusta ja ohjelmistosuunnittelusta.

Laitteistosuunnittelussa kartoitetaan tarvittavat laitteistot ja niiden sijoittelu. Laitteistoon kuuluu esim. instrumentit, PLC:t, kaapelointi ja valvomon tietokoneet. Lopputulokseksi saadaan laitteistokuvaus (Hardware design specification) ja verkkokuvaus (Network design specification).

Ohjelmistosuunnittelu määrittelee mm. ohjelmiston arkkitehtuurin, käyttöliittymät ja rajapinnat. Ohjelmistosuunnittelussa kannattaa käyttää hyväksi aiemmin luotuja moduuleja, mikäli niitä on saatavilla [SAS 2005]. Näin voidaan merkittävästi vähentää ohjelmointia ja testausta sekä niihin kuluvaa aikaa. Usein toistuvien toiminnallisuuksien toteutus kannattaa määritellä jo suunnitteluvaiheessa siten, että komponenttia olisi mahdollista käyttää hyväksi myös tulevissa projekteissa.

Ohjelmistosuunnittelun tuloksena on ohjelmistokuvaus (Software design specification). Ohjelmistokuvaus voi olla laitteistokuvausta yleisempi, eikä se välttämättä sisällä tarkkaa kuvausta ohjelmistokomponenttien toteutuksesta. [SAS, 2005]

Toteutussuunnittelun aikana määrittelyvaiheen ja järjestelmäsuunnittelun tietoja tarkennetaan edelleen. Ohjelmistomoduulien toteutus suunnitellaan yksityiskohtaisesti ja tulokset kirjataan moduulikuvaukseen (Software module design specification). Moduulikuvaukseen liittyen laaditaan testisuunnitelma. Kun kytkentäkaaviot ovat valmiit ja laitteistokuvaus on tarkentunut, tehdään myös laitteistolle testisuunnitelma. Nämä testisuunnitelmat tarkentavat testaussuunnitelmaa, joka puolestaan käsittelee testausta yleisemmin koko projektin tasolla. [SAS, 2005]

Toteutusvaiheessa toimittaja rakentaa automaatiojärjestelmän hankkimalla ja valmistamalla tarvittavat laitteistot ja ohjelmistot. Valmis kokonaisuus testataan toimittajan tiloissa tehtävissä tehdastesteissä (Factory Acceptance Testing, FAT). Tehdastesteissä testataan I/Orajapinnat sekä käyttöliittymät soveltuvin osin. Toteutus vaiheen aikana laaditaan järjestelmän käyttö- ja asennusohjeet. Käyttäjien koulutus voidaan aloittaa tehdastestien yhteydessä. Kun sekä toimittaja että asiakas ovat yksimielisiä siitä että järjestelmä on valmis toimitettavaksi, toteutusvaihe päättyy. [SAS, 2005]

Asennusvaiheessa järjestelmä kuljetetaan kohteeseensa, laitteet asennetaan paikoil-

leen ja kytketään toisiinsa. Kun järjestelmä on saatu asennettua suoritetaan laitteistotestit, joilla varmistetaan järjestelmän mekaaninen ja sähköinen toimivuus. [SAS, 2005]

Toiminnallisen testauksen (Commissioning) tarkoituksena on varmistaa, että järjestelmän toiminta vastaa toiminnallisissa kuvauksissa sovittua. Näin ollen järjestelmä on toimittajan kannalta valmis otettavaksi tuotantokäyttöön.

Toiminnallisen testauksen ensimmäisessä vaiheessa, kylmätestauksessa, testataan ensin järjestelmän turvallisuuteen liittyvät toiminnot. Mahdollisiin turvallisuuspuutteisiin, kuten hätäpysäytyksen toimimattomuuteen, on reagoitava heti ja puutteet on korjattava välittömästi. Jos turvallisuuspuutteita ei havaita, jatketaan testausta käymällä läpi yksinkertaiset toiminnot. Kylmätestaus suoritetaan nimensä mukaisesti mahdollisimman vaarattomilla aineilla ja materiaaleilla. Tämän jälkeen kuumatestauksessa käydään läpi laajemmat toiminnallisuudet käyttäen mahdollisimman oikeita aineita ja materiaaleja sekä olosuhteita.

Toiminnallisen testauksen päätteeksi asiakas hyväksyy tehdyt testit. Automaatiojärjestelmä vastaa nyt toiminnallista kuvausta ja on valmis tuotantokäyttöön. [SAS, 2005]

Tuotannollinen koeajo varmistaa että koko järjestelmä (laitos tai tehdas) toimii halutulla tavalla. Koeajon aikana kiinnitetään huomiota mm. tuotantokapasiteettiin ja laatuun sekä suorituskykyyn.

Toimittaja ja asiakas sopivat keskenään vaadituista kriteereistä ja niiden täytyttyä asiakas hyväksyy järjestelmän. Toimittaja ja asiakas allekirjoittavat luovutustodistuksen ja tuotanto voidaan aloittaa. Järjestelmän takuuaika alkaa tästä hetkestä. [SAS, 2005]

Tuotantovaiheessa automaatiojärjestelmän toimintaa seurataan ja mahdolliset puutteet korjataan. Järjestelmän optimaalisen toiminnan varmistamiseksi tehdään tarvittavat säännölliset huoltotoimenpiteet. Asiakas voi hoitaa järjestelmän ylläpidon itse tai ostaa sen palveluna järjestelmän toimittajalta. Järjestelmien muuttuessa jatkuvasti monimutkaisemmiksi ylläpitosopimukset asiakkaan ja toimittajan välillä ovat yleistyneet. [SAS, 2005] Useasti jo olemassa oleviin automaatiojärjestelmiin halutaan tehdä muutoksia tai lisäyksiä. Muutokset on suunniteltava huolellisesti ja ajoitettava sopivasti, jotta niistä ei aiheutuisi liikaa haittaa tuotannolle. Suuremmat muutokset tehdään tavallisesti uusina ja omina projekteinaan.

3.2 Elinkaaren palvelut

Nykyisin on tavallista että asiakas haluaa ostaa palveluja pelkän laitteen tai järjestelmän sijaan, mikä tarkoittaa sitä että laitteille ja järjestelmille on tarjottava mm. kunnossapitoa ja käyttötukea. Tämän seurauksena järjestelmätoimittaja on mahdollisesti mukana kaikissa elinkaaren vaiheissa aina järjestelmän purkamiseen saakka. Tämän seurauksena olisi erittäin tärkeää ottaa kaikki elinkaaren vaiheet huomioon jo järjestelmää suunniteltaessa, jotta myöhemmät vaiheet sujuisivat mahdol-

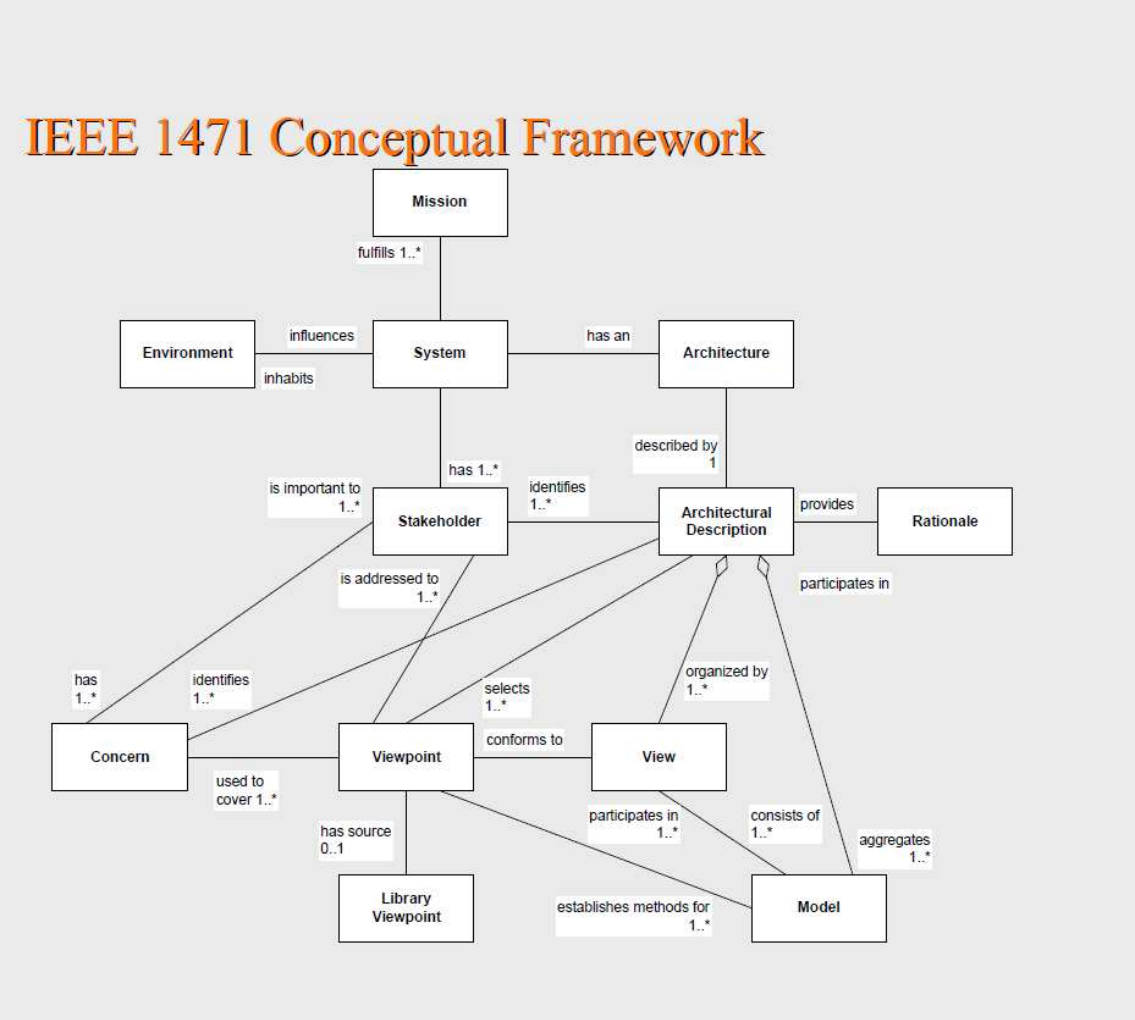
lisimman kitkatta. Automaatiojärjestelmien odotettu toiminta-aika on usein varsin pitkä, minkä vuoksi mm. varaosien ja ohjelmistopäivitysten saatavuus on hyvä varmistaa myös järjestelmän elinkaaren loppuvaiheessa. Järjestelmän ylläpidettävyyttä voidaan parantaa tekemällä ohjelmista selkeitä ja modulaarisia, jolloin ohjelmia ja niiden osia on helpompi päivittää ja muokata jatkossa. [SAS, 2005] Mipro Oy on tavallisesti mukana automaatiojärjestelmän toimituksessa sen suunnitteluvaiheesta eteenpäin. Mipro suunnittelee ja toteuttaa järjestelmän asiakkaan määrittelyjen pohjalta. Tyypillinen Mipron toimittama järjestelmä on puhdas- tai jätevesilaitoksen automaatiojärjestelmä. Automaatiojärjestelmä pitää sisällään yhden tai useamman logiikan, riippuen laitoksen koosta, ja valvomoon sijoitetun scada-järjestelmän. Monesti asiakas haluaa, että esimerkiksi pumppaamot ja paineenkorottamot liitetään osaksi automaatiojärjestelmää kaukovalvottuina ala-asemina. Nämä ala-asemat sisältävät oman logiikan, joka on yhteydessä pääjärjestelmään esimerkiksi radiomodeemilla. Isoissa kaupungeissa tällainen vesihuoltojärjestelmä on helposti laaja ja monimutkainen ja se usein sisältää monta hyvin samanlaista ala-asemaa. Asiakkaalle toimitettava järjestelmä voi olla joko kokonaan uusi, tai vaihtoehtoisesti nykyiseen järjestelmään tehdään lisäyksiä tai muutoksia. Mipro testaa järjestelmän itse, minkä jälkeen se otetaan käyttöön yhdessä asiakkaan kanssa. Järjestelmän fyysinen asennus on ulkoistettu ja suoritetaan alihankintana. Lisäksi Mipro tarjoaa kunnossapitopalvelua toimittamilleen järjestelmille. Seuraavissa luvuissa tarkastellaan menetelmiä, joilla voidaan parantaa järjestelmän laatua mm. kehittämällä suunnittelua, sovelluksen varsinaista ohjelmointia sekä järjestelmän testausta, käyttöönottoa ja ylläpitoa.

4 Järjestelmäarkkitehtuuri

Automaatiojärjestelmää suunniteltaessa on hyödyllistä tunnistaa ja valita oikea arkkitehtuuri, jolla järjestelmä toteutetaan. Tässä luvussa esitellään mitä arkkitehtuuri tarkoittaa ja käydään läpi tyypillisiä arkkitehtuureja. Arkkitehtuuri kuvaa abstraktilla tasolla ja eri näkökulmista järjestelmän rakenteen. Tekemällä arkkitehtuuri selkeäksi saadaan monimutkainenkin järjestelmä helpommin ymmärrettäväksi ja siten voidaan helpottaa eri osapuolien yhteistyötä ja muutosten tekoa [SAS 2005]. Sopivaa arkkitehtuuria käyttämällä järjestelmä saadaan usein toteutettua paremmin, nopeammin ja edullisemmin [Hilliard, 2000]. Jotta koko järjestelmä voitaisiin kuvata tietyllä arkkitehtuurilla, on kuvaus tehtävä riittävän yleisellä tasolla. Arkkitehtuuri kuvaa järjestelmän toiminnalliset ominaisuudet ja ei-toiminnalliset ominaisuudet. (suorituskyky, luotettavuus ja turvallisuus).

4.1 Arkkitehtuurin kuvauksen näkymät

Eri osapuolilla on erilaiset näkökulmat järjestelmään, minkä vuoksi kuvauksesta tarvitaan useita eri näkymiä. Tämä IEEE-järjestön laatima suositus arkkitehtuurin kuvauksesta on esitetty kuvassa 4-1.



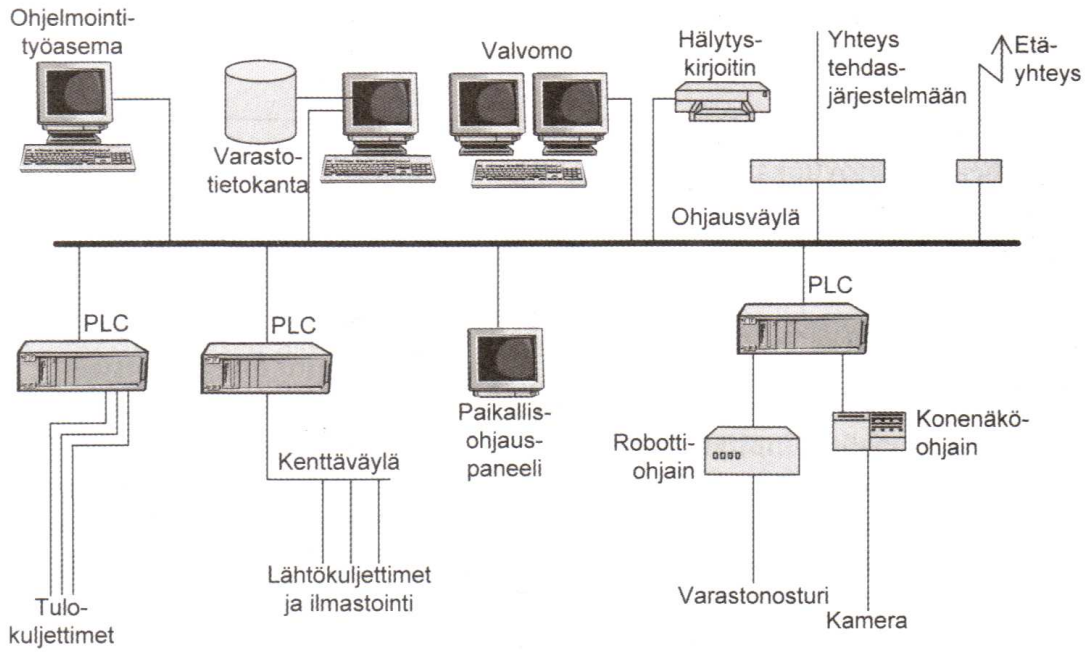
Kuva 2: Arkkitehtuurin malli [Hilliard, 2000]

Eri näkymistä valitaan yksi tai useampi tarpeista riippuen. 4+1-mallissa näkymiksi on valittu skenaariot, looginen näkymä, prosessinäkymä, kehitysnäkymä ja fyysinen näkymä [Kruchten, 1995]. Näkymät eivät yksin ole kaiken kattavia, vaan ne täydentävät toisiaan. Skenaariot (Scenarios) ovat esimerkkejä järjestelmän käyttötapauksista. Skenaariota voidaan pitää järjestelmän vaatimusten kuvauksina ja niitä voidaan käyttää hyväksi järjestelmän määrittelyvaiheessa kun kartoitetaan käyttäjävaatimuksia. Skenaarioiden esitykseen käytetään esimerkiksi vuorovaikutuskaavioita ja skenaarioiden kautta on mahdollista selkeyttää prosessinäkymän ja loogisen näkymän informaatiota. Looginen näkymä (Logical View) sisältää toiminnallisten vaatimusten kuvauksen, joka on selkeä ja riippumaton toteutuksesta. Selkeys saadaan aikaan kätkemällä alemman tason toteutus abstraktimpiin kapseleihin. Looginen näkymä saa alkunsa projektin määrittelyvaiheen aikana, minkä jälkeen sitä tarkennetaan suunnitteluvaiheessa. Looginen näkymä esitetään esimerkiksi SA/SD:n tietovuokaavion tai UML:n aktiviteettikaavion avulla.

Kehitysnäkymä (Development View) kertoo miten ohjelmisto jaetaan osiin ja miten

osat rakentuvat (esimerkiksi PLC:ssä käytettävät toimilohkot kuvataan hierarkkisesti). Kehitysnäkymässä ei käsitellä enää toimintoja abstraktilla tasolla kuten loogisessa näkymässä, vaan siinä kuvataan ohjelmiston jako varsinaisiin toteutettaviin moduuleihin. Kehitysnäkymää käytetään projektin suunnitteluvaiheen aikana ja sen avulla voidaan tunnistaa uudelleen käytettäviä moduuleja.

Fyysinen näkymä (Physical View) kuvaa laitteiston eri osat (mm. PLC:t, tietokoneet...) ja niiden välisen tiedonsiirron (esimerkiksi väylät, verkot ja protokollat). Samalla kuvataan muita, ei-toiminnallisia vaatimuksia, kuten suorituskykyä, luotettavuutta ja laajennettavuutta. Automaation yhteydessä fyysinen näkymä kuvataan järjestelmäkaavion avulla (Kuva 3). Fyysistä näkymää käytetään projektin suunnitteluvaiheessa.



Kuva 3: Järjestelmäkaavio [SAS, 2005]

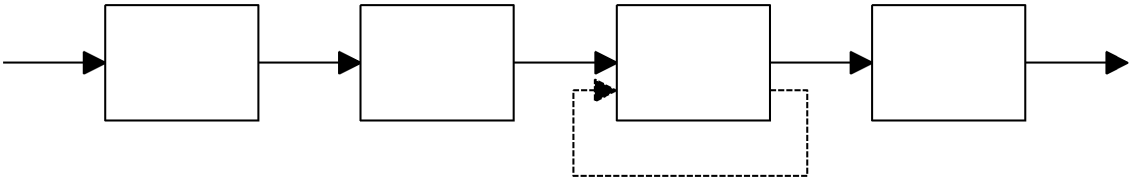
Prosessinäkymällä (Process View) kuvataan automaatiossa käytettävien ohjelmistokomponenttien suoritusta. Tässä yhteydessä prosessilla tarkoitetaan käyttöjärjestelmän päällä ajettavan ohjelman suoritusta. Prosessinäkymän avulla kehitysnäkymän moduulit yhdistetään fyysisen näkymän prosessoreihin.

4.2 Yleiset arkkitehtuurimallit

Samanlaiset rakenteet esiintyvät eri ohjelmistoissa ja niitä voidaan kuvata yleisillä arkkitehtuurimalleilla.

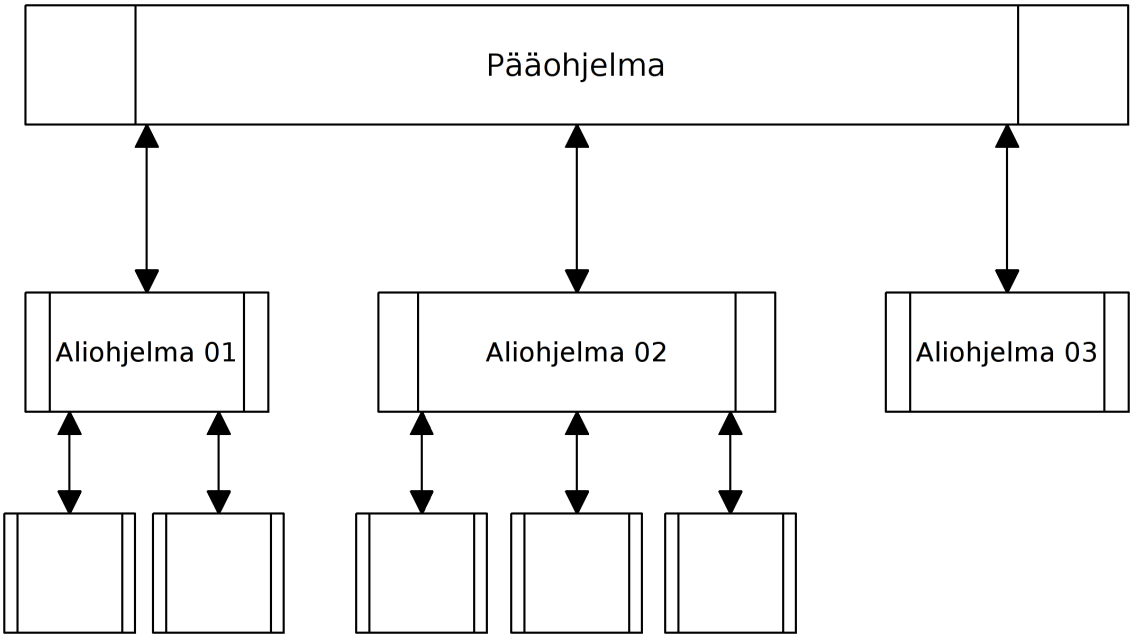
Tietovuoarkkitehtuurissa (Data Flow Systems) tieto virtaa järjestelmän läpi ja matkan varrella sitä prosessoidaan ja siihen tehdään muutoksia. Tietovuoarkkitehtuurit voidaan jakaa kahteen päätyyppiin: ”Batch Sequential”ja ”Pipes and Filters”. Batch

Sequential mallissa osaprosessi suoritetaan kokonaan ennen kuin seuraavan suoritus aloitetaan, kun taas Pipes and Filters mallissa tiedon kulku on jatkuvaa. [SAS, 2005]



Kuva 4: Esimerkki tietovuoarkkitehtuurista [SAS, 2005]

Kutsumekanismiarkkitehtuurimalli keskittyy kuvaamaan ohjelman eri osien suorituksen ohjausta. Pääohjelma-/aliohjelmamallissa ohjelma jaetaan useaan eri osaan, jolloin ohjelmointiongelma on helpommin ratkaistavissa. Tässä mallissa pääohjelma kutsuu aliohjelmaa, joka voi edelleen kutsua toista aliohjelmaa, jne. Kutsuva ohjelma odottaa että kutsuttu ohjelma on saatu suoritettua. Osa ohjelmoitavista logiikoista sisältää organisointilohkon, joka kutsuu suoritettavia lohkoja toimien siten pääohjelman tapaan. [SAS, 2005]

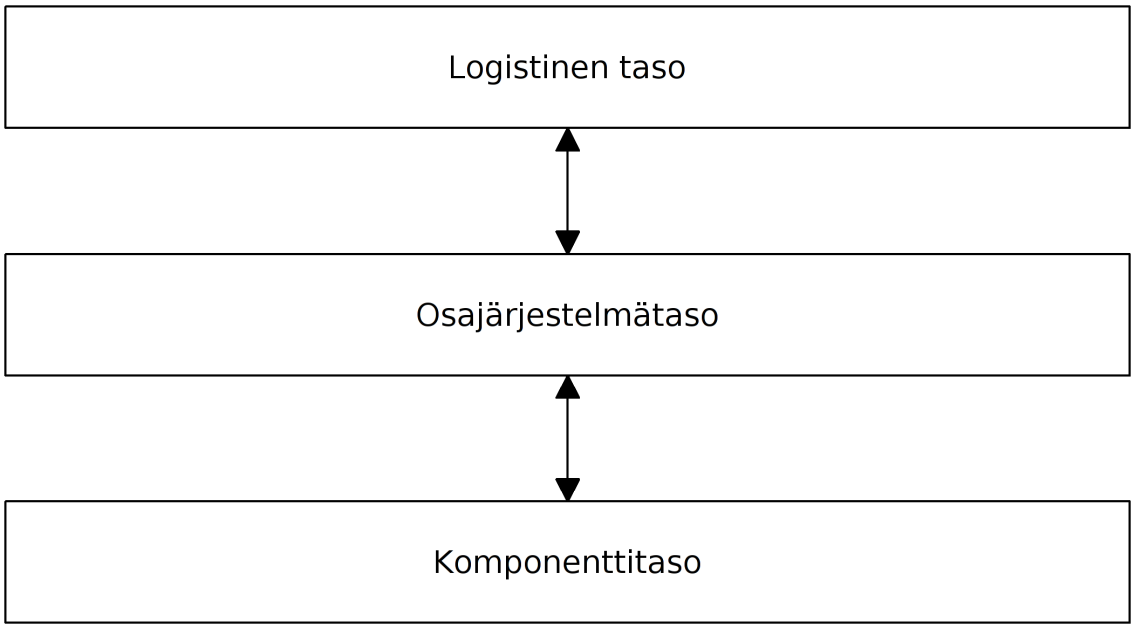


Kuva 5: Pää- aliohjelma -arkkitehtuuri [SAS, 2005]

Kerrosarkkitehtuurimallissa alempi kerros tarjoaa palveluja, joita ylempi kerros käyttää. Alemman tason tarjoamista yleisistä palveluista kootaan ylemmillä tasoilla sovelluksessa tarvittava kokonaisuus.

Hierarkkisessa kerrosarkkitehtuurimallissa on palvelukutsut kohdistettava aina joko samalle tai alemmalle kerrokselle. Ohituksettomassa kerrosarkkitehtuurimallissa ei ole sallittua ohittaa välissä olevia kerroksia palvelukutsuja tehtäessä. [SAS, 2005]

Koska automaatiossa eri järjestelmien ja prosessilaitteiden fyysinen rakenne ja toiminnallisuus ovat kerroksellisia, on kerrosarkkitehtuurin käyttö luonnollista.



Kuva 6: Kerrosarkkitehtuuri [SAS, 2005]

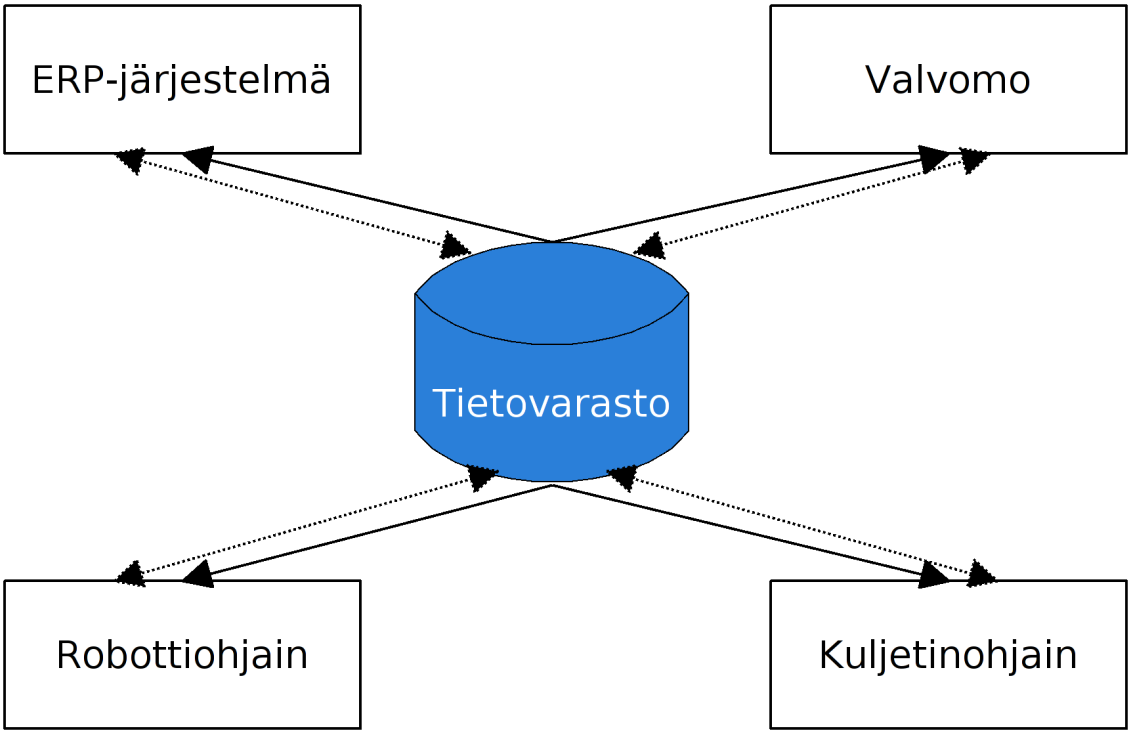
Oliopohjaisessa ohjelmoinnissa tieto ja sen käsittely kootaan ikään kuin kapseleiksi, joihin pääsee käsiksi vain ennalta määrättyjen metodien kautta. Tällaisen kapseloinnin tarkoituksena on kätkeä joskus monimutkainenkin toteutustapa kapselin käyttäjältä, jolloin voidaan edesauttaa ohjelmamoduulien muunneltavuutta ja uudelleenkäyttöä. [SAS, 2005]

Riippumattomien komponenttien arkkitehtuurimallissa komponentit kommunikoivat lähettämällä toisilleen viestejä. Eri komponentit ovat itsenäisiä eivätkä ne ole kovin vahvasti sidoksissa toisiinsa. Esimerkkeinä tästä arkkitehtuurimallista ovat client-server ja valtuuden välitys.

Tietovarastoarkkitehtuuri on tietokeskeinen arkkitehtuurimalli ja sen ytimenä toimii tietovarasto. Ohjelmistokomponenttien avulla tietovarastoon lisätään, poistetaan ja päivitetään tietoja sekä haetaan sieltä tietoja. Kaikki kommunikointi komponenttien välillä tapahtuu tietovaraston kautta. Aktiivinen tietovarasto ilmoittaa muille komponenteille sisältämänsä tiedon muutoksista, kun taas passiivisessa tietovarastossa siihen liittyneet komponentit huolehtivat tietomuutoksista. Järjestelmät jotka sisältävät paljon tietoa ja joissa siksi tarvitaan tiedonhallintaa soveltuvat hyvin kuvattavaksi tietokeskeisellä arkkitehtuurimallilla.

4.3 Ohjelmistoarkkitehtuurit automaatiossa

Alun perin automaatiojärjestelmät ovat olleet melko yksinkertaisia: järjestelmän osat ovat toimineet itsenäisesti ja kommunikoineet vähän tai eivät lainkaan kes-



Kuva 7: Tietovarastoarkkitehtuuri [SAS, 2005]

kenään. Tekniikan kehitys on mahdollistanut osajärjestelmien liittämisen toisiinsa: logiikat on liitetty valvomon SCADA-järjestelmiin ja tämä edelleen esimerkiksi raportointijärjestelmään. Tässä käy ilmi automaatiolle tyypillinen kerroksellinen rakenne, jossa eri tasot eroavat merkittävästi toisistaan niin ominaisuuksiltaan kuin vaatimuksiltaan.

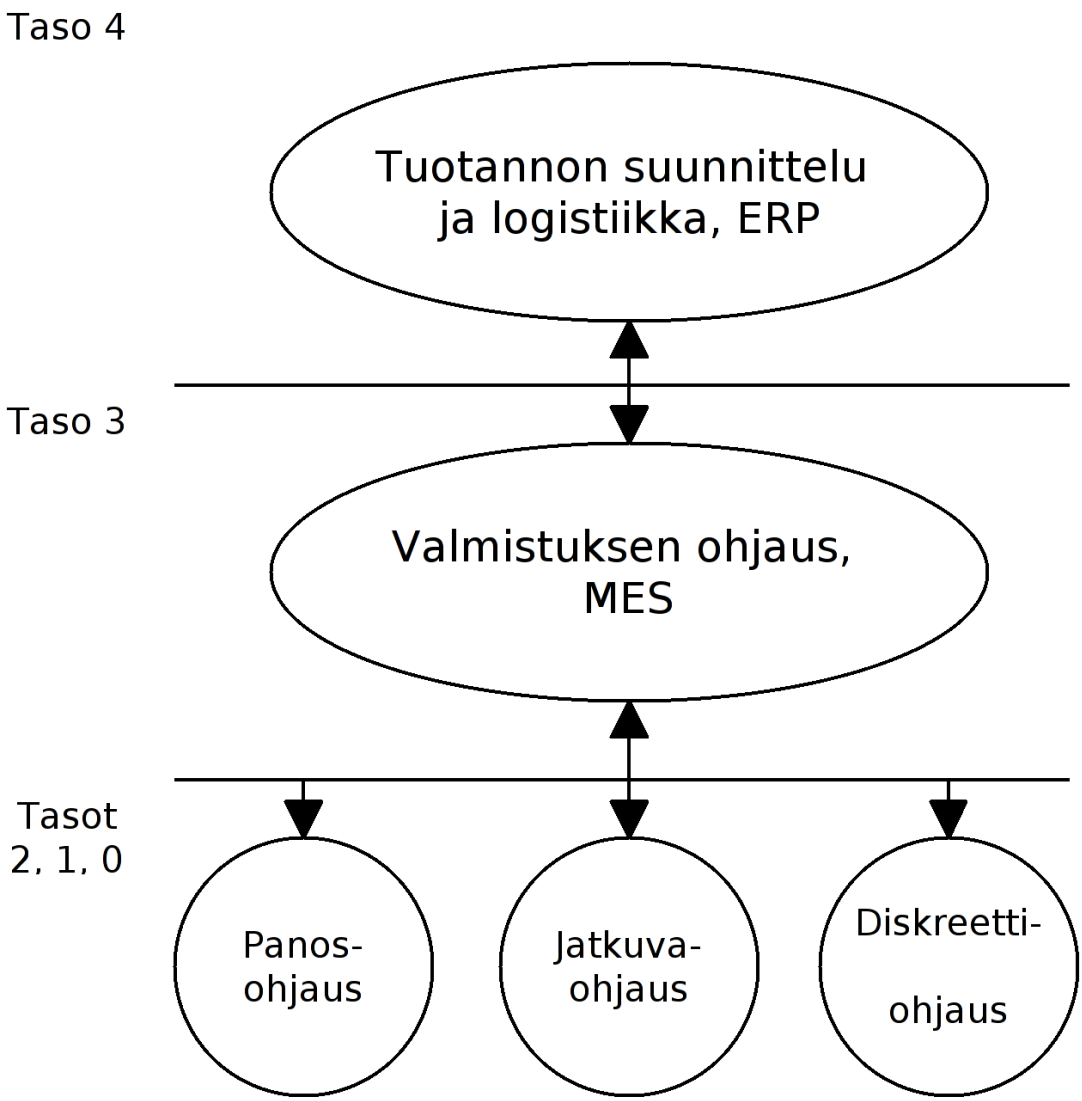
Vaatimukset tuotannon ja toiminnan tehostamiseksi ovat saaneet aikaan tarpeen liittää automaatio osaksi yritystason järjestelmiä. Jotta tällainen integraatio olisi mahdollista, tarvitaan yhtenäinen malli jossa määritellään käytettävät käsitteet ja toiminnot.

Standardissa ANSI/ISA-95 on määritelty malleja automaatiojärjestelmien integroimiseksi yritystason järjestelmiin [ANSI/ISA-95.00.01 2000]. Mallissa automaatiojärjestelmä liitetään valmistuksenohjausjärjestelmään (Manufacturing Executing System, MES) joka puolestaan liitetään toiminnanohjausjärjestelmään (Enterprise Resource Planning, ERP).

MES sisältää automaatioon liittyviä toiminnallisuuksia sekä muita toimintoja, kuten laadunvarmistus ja kunnossapito.

Standardin mukaan laitoksen toiminnan kuvaamiseen käytetään tietovuokaaviota. Käsitteiden väliset suhteet kuvataan olioluokkakaavion avulla (Kuva 9).

ANSI/ISA-95 standardin kehityksessä on ollut mukana useita alan merkittäviä yrityksiä, joten sillä on mahdollisuus saavuttaa keskeinen asema automaation sovellusten ja käsitteiden yhdistämisessä.

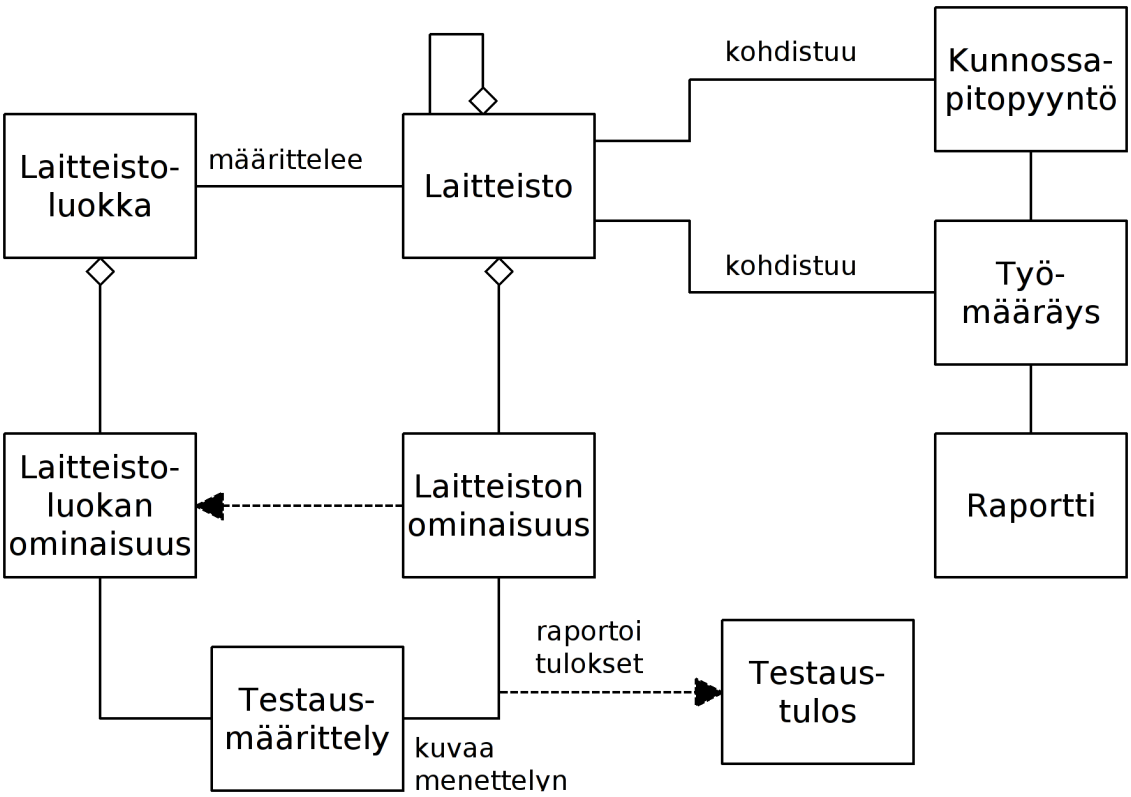


Kuva 8: Toiminnallinen hierarkia [SAS, 2005]

Perinteisesti automaatiolaitteistojen väliset liitynnät ovat lähes poikkeuksetta olleet valmistajakohtaisia. Laitteistojen laajentaminen ja toisiinsa liittäminen on ollut hankalaa yhteisen liityntärajapinnan puuttuessa.

Tätä puutetta paikkaamaan perustettiin vuonna 1996 OPC Foundation. Yhteisön tehtävänä on edesauttaa automaatiosovellusten integrointia ja sovellusten välisen prosessidatan välitystä laatimalla avoimia määrittelyjä näitä varten. OPC Foundation:iin kuuluu yli 450 yritystä eri puolilta maailmaa. Yhteisö määritteli OPC-rajapinnan (OLE for Process Control) joka mahdollisti eri valmistajien automaatiolaitteiden käyttämisen Microsoftin Windows-ympäristössä.

OPC:n avulla on siis mahdollista liittää esimerkiksi eri valmistajien ohjelmoitavia logiikoita samaan valvomoon. Tällöin valvomotietokoneessa pyörivä OPC-palvelin pystyy kommunikoimaan eri protokollien kautta usean eri PLC:n kanssa. OPC-



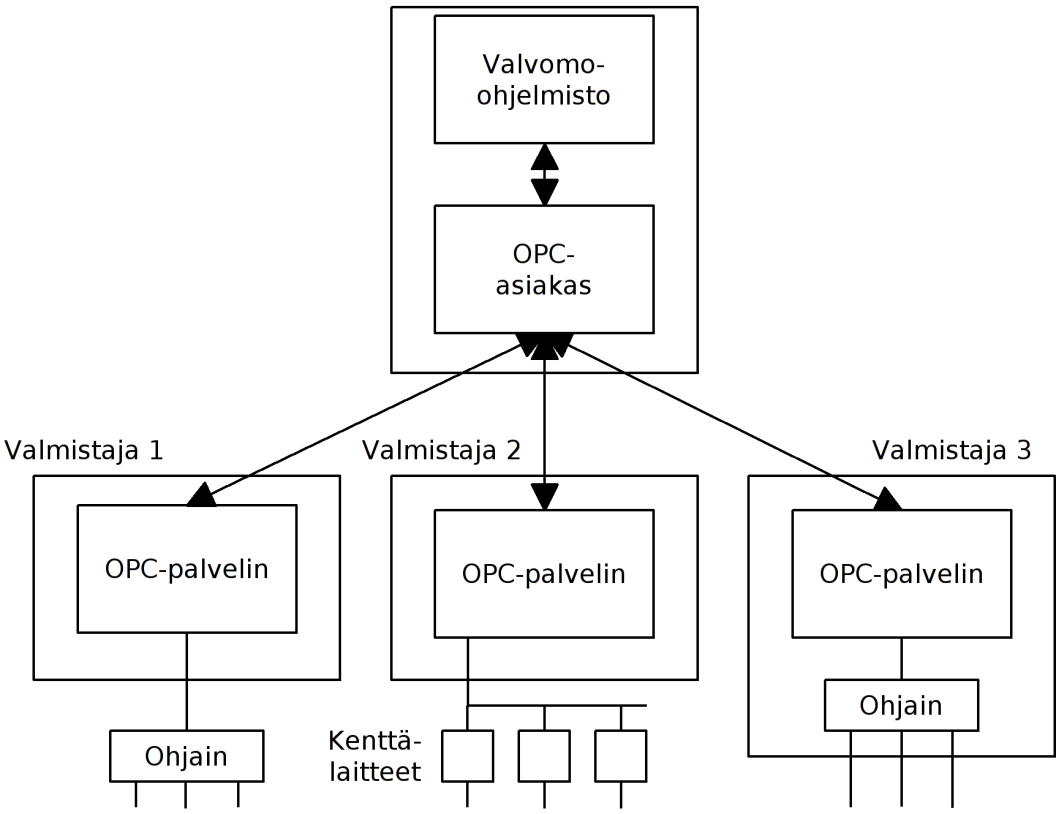
Kuva 9: Laitteistomalli [SAS, 2005]

asiakasohjelma kerää haluamansa tiedot palvelimelta ja välittää ne valvomo-ohjelmistolle, joka puolestaan visualisoi tiedot näytöllä.

Kuten edellä kävi ilmi, OPC noudattaa palvelin-asiakas -arkkitehtuuria. Eri tilanteita ja käyttötarkoituksia varten on tarjolla useita eri määrityksiä. Ohessa esimerkkejä OPC-määrittelyistä:

- Data Access (DA): muuttujatietojen luku/kirjoitus
- Alarms and Events (A&E): hälytysten ja tapahtumien välitys
- Historical Data Access (HDA): historiatietojen välitys
- Data eXchange (DX): palvelimien välinen tiedon välitys
- XML-DA: XML-pohjainen tietojen välitys

OPC-määritykset käyttävät XML-DA:ta lukuun ottamatta Microsoftin OLE/DCOM-tekniikkaa. Ensimmäiseksi määriteltiin DA, jonka avulla OPC-client pääsee käsiksi OPC-serverin muuttujiin ja voi lukea ja kirjoittaa niitä. DA on yleisesti käytössä ja siitä on tullut automaatiojärjestelmien de-facto-standardi.



Kuva 10: OPC:n arkkitehtuuri [SAS, 2005]

A&E-määrittelyä käytetään hälytys- ja tapahtumaviestien välityksessä. Määrittelyn mukaan palvelin havaitsee automaatiolaitteen muodostamat tapahtumat ja hälytykset ja välittää ne asiakkailleen. Palvelin voi myös itse muodostaa hälytyksiä ja tapahtumia.

OPC HDA määrittelee rajapinnan historiatietojen välitykseen. Se soveltuu esimerkiksi automaation ja tuotannonohjausjärjestelmien yhdistämiseen.

Jos sovelluksessa halutaan välittää tietoja suoraan OPC-palvelimesta toiseen, voidaan siihen käyttää Data eXchange määrittelyä. Tiedonsiirto on tässä tapauksessa DA:sta poiketen vaakasuuntaista. Käytännössä monet DA-palvelinsovellukset pystyvät itsenäisesti kommunikoimaan eri valmistajien automaatiolaitteiden kanssa, jolloin tarve DX-rajapinnalle on jäänyt vähäiseksi.

XML-DA määrittelyn tavoitteena on laajentaa DA:n tarjoamia mahdollisuuksia. XML-DA mahdollistaa kommunikoinnin palvelimeen jopa internetin välityksellä. Tämä on saatu aikaan korvaamalla Microsoftin DCOM-tekniikka SOAP-protokollalla (Simple Object Access Protocol) ja WSDL:llä (Web Services Description Language), jotka ovat alustasta riippumattomia. Koska tietoliikenne on XML-pohjaista, jää varsinaisen hyötykuorman kapasiteetti pieneksi ja siten tiedonsiirron suorituskyky heikkenee merkittävästi.

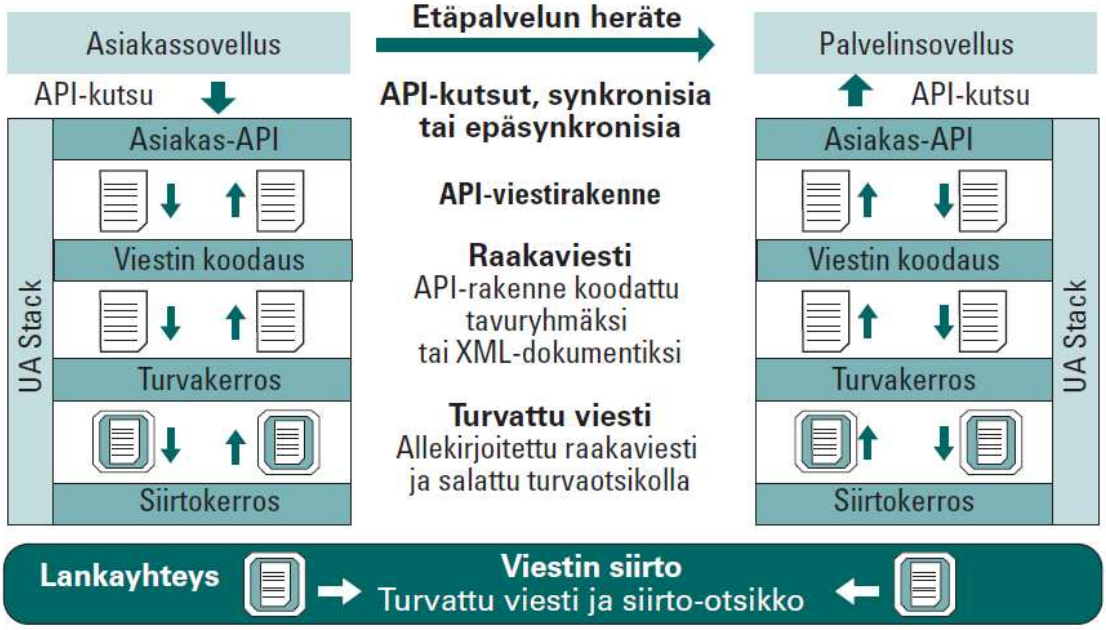
Koska OPC perustuu Microsoftin OLE/DCOM-tekniikkaan, rajaa se sovellusym-

päristön käyttöjärjestelmän Microsoftin Windows:in eri versioihin. Ei myöskään ollut varmaa, tulisiko Microsoft säilyttämään DCOM-komponentit tulevissa Windows versioissa. Toimivien DCOM-asetuksien löytäminen saattoi olla työlästä, eikä DA-määrittely soveltunut hyvin käytettäväksi usean tietokoneen välillä monimutkaisissa verkoissa [www.opcconnect.com].

Aikaisempien useiden eri määrittelyiden tilalle haluttiin luoda yksi yhtenäinen alustasta riippumaton määrittely, joka olisi myös mahdollisimman pitkäikäinen. Tämän uuden, OPC Unified Architecture nimisen arkkitehtuurin kehittäminen aloitettiin vuonna 2004. Se pyritään saamaan IEC:n viralliseksi standardiksi [Prosessori, 12/2007].

OPC UA ei käytä edeltäjistään poiketen Microsoftin DCOM-tekniikkaa, mikä mahdollistaa alustan vapaamman valinnan. Lisäksi UA yhdistää aikaisempien määrityksien (mm. DA, A&E ja HDA) toiminnallisuudet ja korvaa niiden käyttämät eri tietorakenteet uudella oliomallilla. Tämä helpottaa osaltaan automaation tietojärjestelmien yhdistämistä muihin ylemmän tason järjestelmiin.

UA:n kommunikointipino (UA Stack) huolehtii sovellusten välisestä tiedonsiirrosta ja pino koostuu neljästä kerroksesta: siirtokerros, turvakerros, viestin koodaus ja asiakas-API.



Kuva 11: OPC UA kommunikaatiopino [Prosessori, 12/2007]

Alimpana oleva siirtokerros huolehtii viestien välityksestä verkkotasolla käyttäen TCP/IP- ja HTTP/SOAP-protokollia. Käytettäviä siirtoratkaisuja on mahdollista tarvittaessa lisätä tulevaisuudessa.

Toisena kerroksena on turvakerros joka salaa välitettävät viestit käyttäen yksinkertaista salausta. TCP-yhteydet salataan käyttäen UA:n omaa binääristä UA Secure

Conversion:ia ja SOAP-yhteydet salataan WS-security-protokollalla.

Välitettävät viestit, eli palvelupyynnöt ja -vastaukset, koodataan siirrettävään muotoon viestin koodauskerroksessa. Koodaamiseen voidaan käyttää joko tavu- tai XML-muotoa. Molempia koodaustapoja voidaan käyttää niin TCP/IP-yhteyden kuin SOAP-yhteyden kanssa.

Ylimpänä pinossa on asiakasohjelmointirajapinta (Asiakas-API) hoitamassa viestien välitystä. Käytettävät siirto-, salaus- ja koodaustavat määritellään profiileissa, joita voi olla yksi tai useampi.

Aikaisemmat OPC-määritelmät eivät sisältäneet lainkaan tietoturvaan liittyviä kohtia, mikä oli ongelmallista erityisesti kun sovelluksia haluttiin ajaa erilaisissa verkkoympäristöissä. Viime kädessä käyttäjän täytyi itse huolehtia riittävän tietoturvan saavuttamisesta järjestelmässä.

OPC UA muuttaa tilanteen paremmaksi määrittelemällä yksityiskohtaisesti menetelmät järjestelmän tietoturvan ratkaisemiseksi. Menetelmät perustuvat käytännössä toimiviksi ja hyviksi osoittautuneisiin tekniikoihin.

UA:n toinen osa, Security Model, sisältää tietoturvaa koskevat vaatimukset, jotka on otettava huomioon sovelluskehityksessä heti alusta alkaen. Turvakerroksen kaksi protokollaa noudattavat samaa arkkitehtuuria ja sovellus voi joustavasti käyttää kumpaa tahansa protokollaa, koska ne näkyvät sovellukselle samanlaisina. Läpinäkyvyys protokollille on saatu aikaan kopioimalla Web Services:in tietoturva-arkkitehtuuri myös UA Secure Conversion:iin.

Aikaisemmat OPC-määritelmät ovat mahdollistaneet automaatiosovellusten toteuttamisen Microsoft Windows-ympäristöissä. Samalla kuitenkin sidonnaisuus suljettuun Microsoftin DCOM-tekniikkaan on rajoittanut perinteisten OPC-toteutusten alustavaihtoehtoja. Käytännössä OPCtoteutus on vaatinut että järjestelmä sisältää Windows-työaseman tai - palvelimen, jossa sovellusta ajetaan. OPC:n integroiminen esimerkiksi suoraan PLC:hen ollut erittäin hankalaa.

OPC UA:ta on puolestaan mahdollista käyttää eri alustoilla, koska se ei perustu Microsoftin DCOM-tekniikkaan. Jatkossa OPC UA voidaan toteuttaa myös sulautetuissa järjestelmissä, joiden resurssit voivat olla hyvinkin rajalliset (esimerkiksi PLC:ssä, taajuusmuuttajissa tai jopa kenttälaitteissa). Samalla ethernet-verkko saadaan ulottumaan näille laitteille saakka.

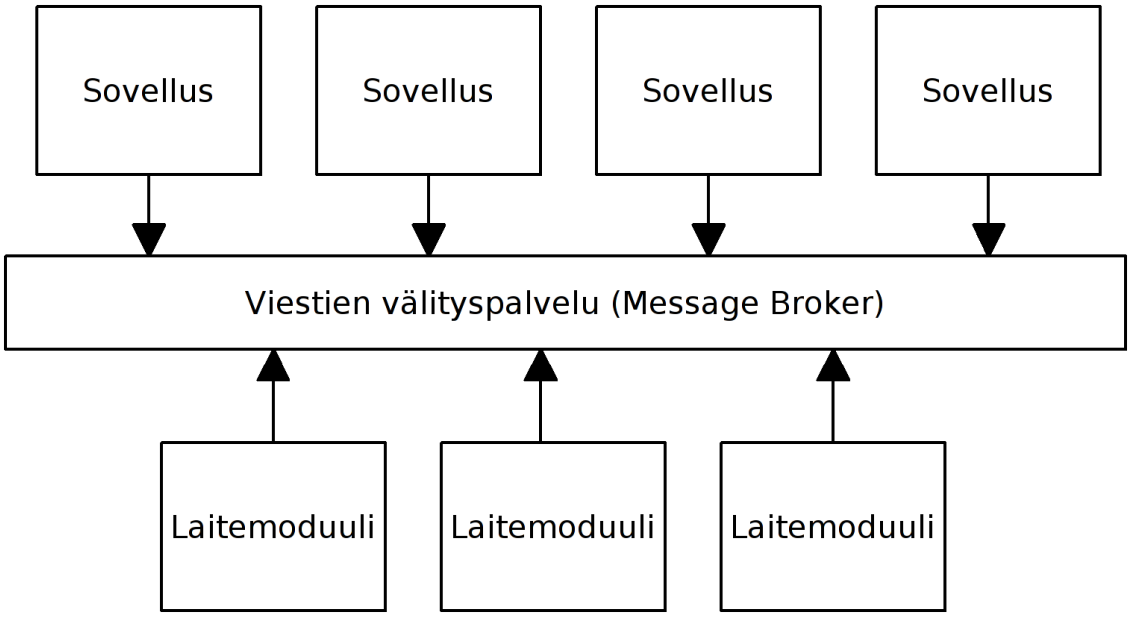
UA:n referenssitoteutuksen suorituskyky vastaa perinteisen DA:n suorituskykyä, jos käytetään TCP-yhteyttä ja tavukoodausta eikä siirrettävien datapisteiden (muuttujien) määrä viestiä kohden ole kovin suuri.

Kappaletavaratuotannon tuotantolinjat koostuvat vakiomoduuleista, jotka voivat olla joko täys- tai puoliautomaattisia. Moduulien toiminnot vaihtelevat käyttötarkoituksen (esimerkiksi kokoonpano, testaus ja pakkaus) mukaan. Lisäksi linjojen on oltava nopeasti uudelleen muokattavissa tuotannon muuttuessa.

Yksittäisessä moduulissa on yleensä PLC tai robotti, joka ohjaa moduulin toimintaa. Moduuleista koostuvaa linjaa ohjaa oma ohjain, johon ylemmän tason järjestelmät

(MES, ERP) ovat liittyneinä. Tyypillinen esimerkki tällaisesta tuotantolinjasta on elektroniikan kokoonpanolinja.

Jotta eri valmistajien moduulit ja sovellukset toimisivat yhdessä ilman ongelmia, on kehitteillä useita standardeja. Yksi näistä on Association Connecting Electronic Industries:in (IPC) laatima standardisarja IPC- 25XXX [IPC-2501, 2003]. Se sisältää CAMX-määrittelyihin (Computer Aided manufacturing using XML) perustuvan viestienvälitysmekanismin (Message Broker).



Kuva 12: CAMX-määrittelyn mukainen viestienvälitysmekanismi [SAS, 2005]

Standardi pitää sisällään käytettävän kommunikointikerroksen ja sen viestityyppien määritelmät. Message Broker vastanottaa siihen liittyneiden komponenttien lähettämät viestit ja välittää kunkin viestin sen tilanneelle komponentille.

Määritelmät perustuvat HTTP:n, SOAP:in ja XML:n käyttämiseen ja Message Broker onkin siis kuin mikä tahansa internetpalvelin. Tämä mahdollistaa määritelmän hyödyntämisen myös muissa sovelluksissa.

OMAC on ei -kaupallinen järjestö, johon eri alojen yritykset, loppukäyttäjät ja yhteisöt ovat liittyneet kehittääkseen avoimia ja keskenään yhteensopivia automaatiojärjestelmiä.

OMAC:in tavoitteena on mm. luoda yleinen arkkitehtuuri ohjausjärjestelmille [OMAC, 2002a]. Määrittelyt pyritään rakentamaan edullisiin ja toimiviksi havaittuihin ratkaisuihin pohjautuen.

Vaikka vahvimmat vaikuttajat OMAC:issa ovat autoteollisuus ja kappaletavaran tuottajat, minkä vuoksi OMAC:in kehittämät menetelmät soveltuvat erityisesti näille aloille, löytyy OMAC:ista menetelmiä myös prosessiautomaatioon liittyen. OMAC:illa on kuusi keskeistä kehitys- kohdetta, joille on omat työryhmänsä:

- arkkitehtuuri
- ohjelmistorajapinnat (OMAC API)
- käyttöliittymät
- Microsoft Manufacturing User Group
- työstökoneiden ohjelmointi
- pakkauskoneiden liikkeenohjaus

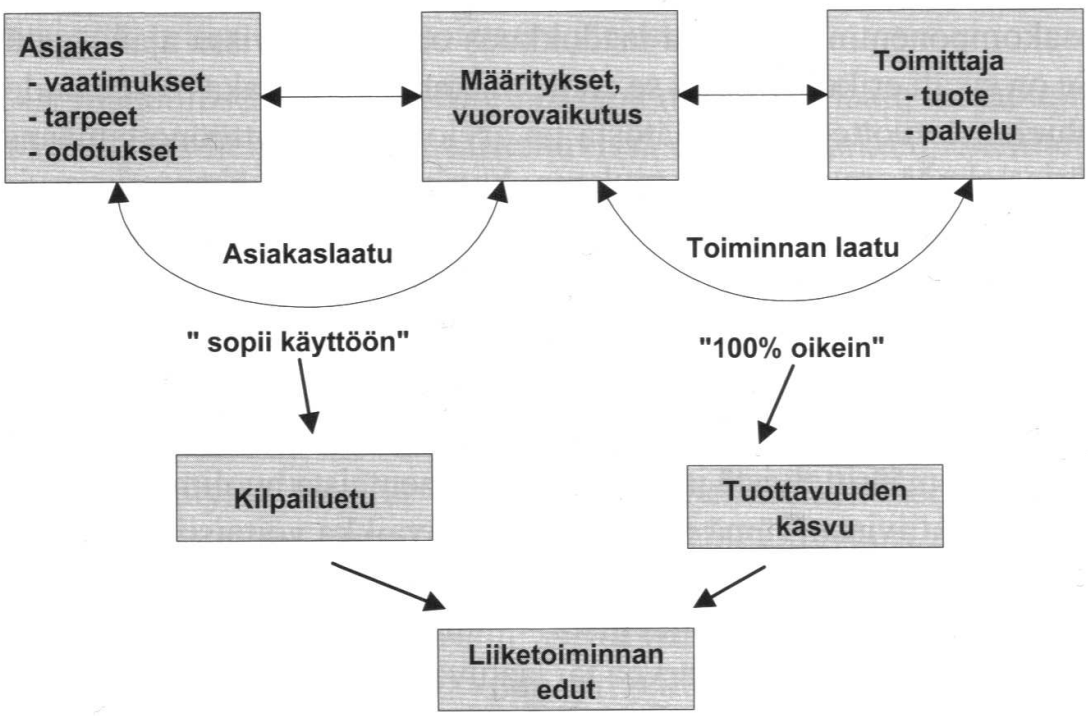
4.4 Arkkitehtuurin vaikutukset järjestelmään

Automaatiojärjestelmät kehittyvät jatkuvasti yhä monimutkaisemmiksi ja suuremmiksi. Järjestelmät saattavat sisältää useita erilaisia väyliä ja tietoliikenneprotokollia, minkä vuoksi automaatiojärjestelmää suunniteltaessa ja toteutettaessa on hyvä tuntea erilaiset arkkitehtuuriratkaisut. Jokin arkkitehtuuri voi toimia hyvin pitkänkin aikaa, mutta kun järjestelmään halutaan tehdä muutoksia aiheuttaa arkkitehtuuri ongelmia. On olemassa kasvava tarve liittää erilaisia automaatiojärjestelmiä toisiinsa, esimerkiksi kaksi kunnallista vesihuoltojärjestelmää halutaan yhdistää kuntaliitoksen yhteydessä, jolloin on tärkeää että järjestelmien arkkitehtuurit mahdollistavat yhteen liittämisen. Jatkuvasti on nähtävissä trendi, jossa kenttälaitteet muuttuvat yhä älykkäämmiksi sisältäen enemmän sulautettua elektroniikkaa (esimerkiksi taajuusmuuttajat). Tällaisten monipuolista toiminnallisuuksia omaavien kenttälaitteiden yhteydessä on hyödyllistä käyttää uudelleenkäytettäviä toimilohkoja. Tällöin toiminnallisuus täytyy ohjelmoida vain kerran ja jatkossa ohjelmointi on enemmän sopivien parametrien antamista lohkoille. On myös täysin mahdollista, että OPC-palvelin sisällytetään kenttälaitteeseen. Tämän seurauksena perinteinen kerrosarkkitehtuuri ikään kuin litistyy ja kenttälaite siirtyy lähemmäksi yritystason järjestelmiä.

5 Laatu

Sanalle laatu löytyy erilaisia määritelmiä riippuen mitä lähdettä käytetään. Teollisuudessa laadulla tarkoitetaan yleisesti tuotteen tai palvelun kykyä täyttää asiakkaan suorat ja epäsuorat tarpeet. Laatu on siis hyvin subjektiivinen ominaisuus, joka riippuu henkilöstä ja ympäristöstä [Haikala & Märijärvi, 2002]. Laatu ei tässä yhteydessä merkitse huippulaatua, vaan tuotteen tai palvelun ominaisuuksia eli laatutekijöitä, jotka voidaan mitata.

Laatua voidaan tarkastella myös taloudelliselta kannalta. Tällöin voidaan käyttää mittarina asiakastyytyväisyyttä. Se kuinka korkea asiakastyytyväisyys saavutetaan, vaihtelee tapauskohtaisesti. Perinteisesti on ajateltu, että asiakkaan tyytyväisyys saavutetaan, kun hän saa haluamansa tuotteen haluamallaan ajankohtana ja haluamallaan hinnalla.



Kuva 13: Liiketoiminta ja laatu [Haikala & Märijärvi, 2004]

Nykyisin tämä ei välttämättä pelkästään riitä, vaan asiakas haluaa usein ratkaisun ongelmaansa tai tarpeeseensa. Tämä tarkoittaa, että tuotteen lisäksi pitää olla tarjolla myös sitä tukevia palveluja, kuten asennus-, käyttö- ja kunnossapitopalveluja. Tuotteen hinnan ja laadun suhde muodostaa asiakkaan saaman lisäarvon. Mitä enemmän asiakas saa lisäarvoa maksamaansa hintaan nähden, sitä tyytyväisempi hän on [Gale, 1994].

5.1 Automaatiojärjestelmän laatu

Alun perin automaatiojärjestelmien toimitukset ovat olleet laitteistopainotteisia ja ohjelmiston osuus on ollut pieni. Järjestelmien elinkaaret ovat olleet pitkiä, eikä uusia teknologioita ja menetelmiä ole otettu käyttöön niin nopeasti ja laajasti kuin ohjelmistoteollisuudessa. Nykyään kuitenkin ohjelmistojen merkitys automaatiojärjestelmien osana kasvaa jatkuvasti [SAS, 2005]. Tämän seurauksena esiin on noussut tarve kehittää automaation ohjelmistotuotantoa ja parantaa sen laatua.

Ohjelmiston laatua voidaan tutkia monella tavalla. Yksi nykyisin erityisen suosittu tapa on esitetty standardissa ISO-9126 [Tian, 2005]. Standardissa laadun määrittelyyn on käytetty kuutta tekijää, joilla on vielä ei-päällekkäisiä ominaisuuksia.

- Toiminnallisuus (Functionality): Ohjelmiston toiminnot joilla tarpeet tyydytetään ja toimintojen ominaisuudet, kuten:

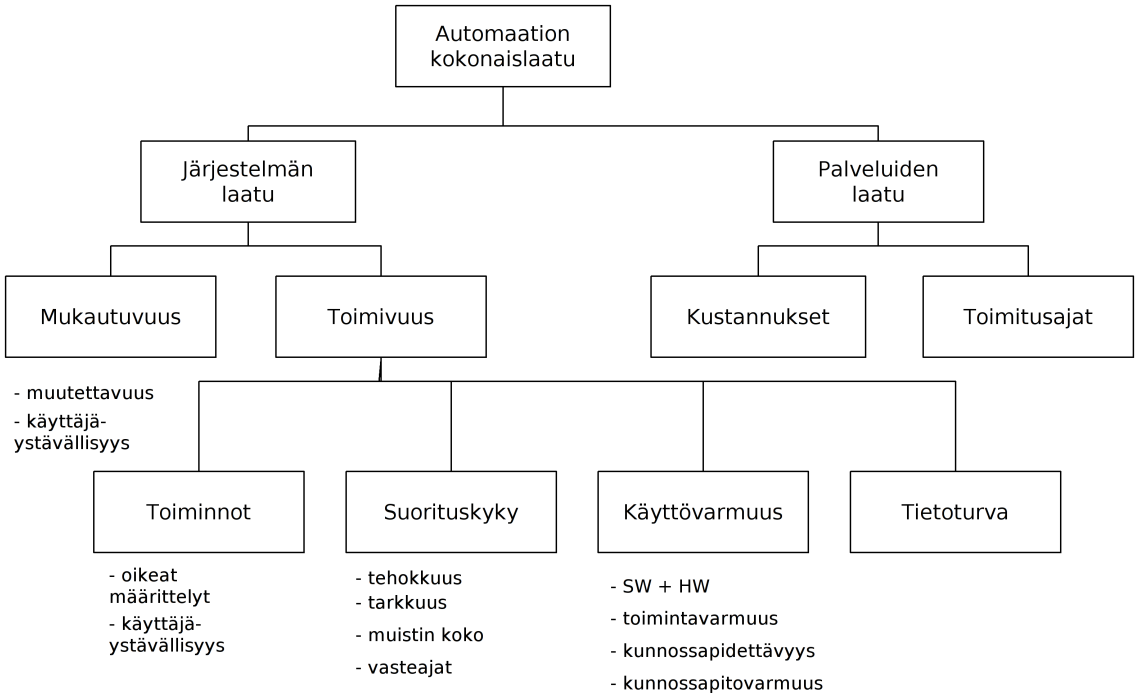
  - soveltuvuus
  - tarkkuus
  - yhteensopivuus
  - turvallisuus

- Luotettavuus (Reliability): Ominaisuudet jotka kuvaavat ohjelmiston kykyä toimia halutulla suorituskyvyllä ja säilyttää se taso. Esimerkiksi:
  - kypsyys
  - vikasietoisuus
  - viasta toipuminen
- Käytettävyys (Usability): Ominaisuudet, jotka kertovat ohjelmiston käytettävyydestä, kuten:
  - ymmärrettävyys
  - opittavuus
  - operoitavuus
- Tehokkuus (Efficiency): Ominaisuudet, jotka kertovat ohjelmiston suorituskyvyn ja käytettyjen resurssien suhteesta, kuten:
  - toiminta ajan kuluessa
  - resurssien kulutus
- Ylläpidettävyys (Maintainability): Ominaisuudet, jotka kertovat mahdollisuuksista tehdä ohjelmistoon muutoksia, kuten:
  - analysoitavuus
  - muutettavuus
  - vakaus
  - testattavuus
- Siirrettävyys (Portability): Ominaisuudet, jotka kertovat mahdollisuuksista siirtää ohjelmisto ympäristöstä toiseen, kuten:
  - sovellettavuus
  - asennettavuus
  - yhdenmukaisuus
  - korvattavuus

Tällainen tiukka hierarkia ei välttämättä sovellu kaikkiin tilanteisiin, koska tuotteen tietyt ominaisuudet voivat olla liitoksissa useampaan laatutekijään, esimerkiksi redundanttisuus vaikuttaa sekä luotettavuuteen että ylläpidettävyyteen.

Automaatiojärjestelmän yhteydessä laatu on jaettavissa sisäiseen ja ulkoiseen laatuun [SAS, 2001]. Sisäisesti laatua tarkasteltaessa keskitytään vain itse järjestelmään, eli sen ohjelmiston toteutukseen, arkkitehtuuriin ja dokumentointiin, eikä kiinnitetä huomiota ympäristöön ja sen vaikutuksiin järjestelmään. Ulkoisessa tarkastelussa laatutekijöitä tutkitaan järjestelmän varsinaisessa käyttöympäristössä. Voidaan siis ajatella, että sisäisesti laatua tarkasteltaessa käytetään kehittäjän näkökulmaa ja ulkoisesti tarkasteltaessa käyttäjän näkökulmaa.

Automaatiojärjestelmän yhteydessä merkittävimmiksi laatutekijöiksi voidaan nostaa toimintavarmuus, suorituskyky ja turvallisuus [SAS, 2005]. Eräs mahdollinen esitystapa automaation laatutekijöistä on esitetty kuvassa 5-2.

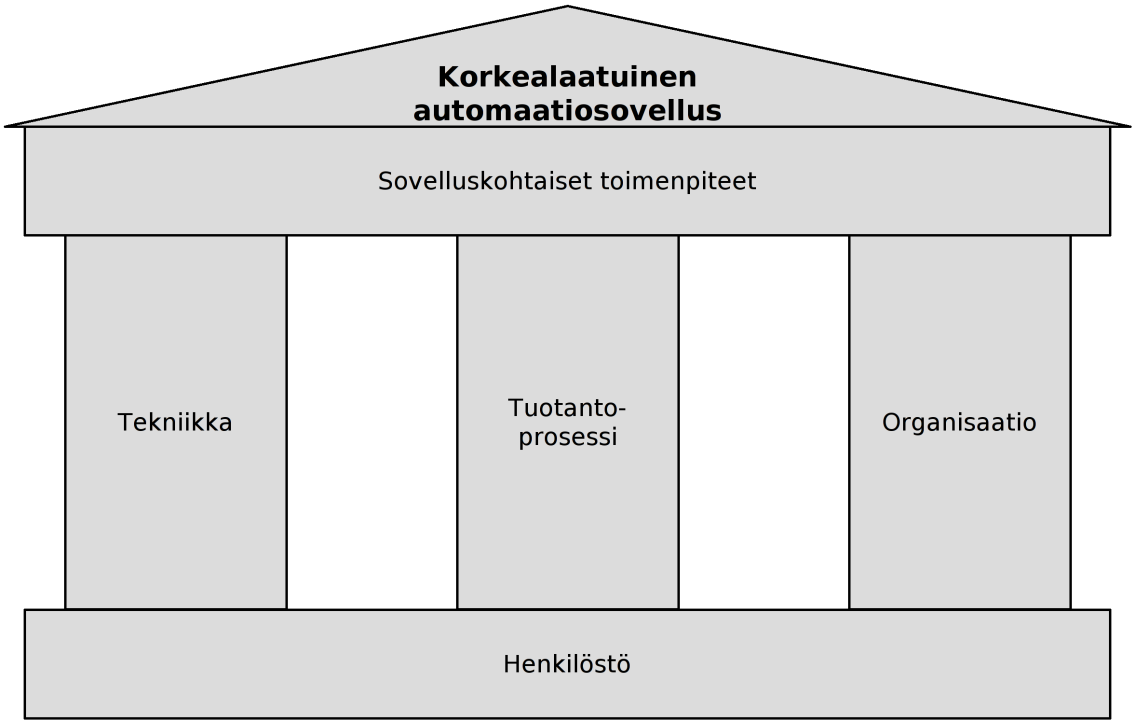


Kuva 14: Laadun osatekijät automaatiossa [SAS, 2001]

5.2 Laadun edellytykset

Pyrittäessä luomaan korkeatasoista automaatiosovellusta, ei ole olemassa yhtä ainoaa oikeaa ratkaisua. Voidaan kuitenkin löytää tietyt laatua varmistavat edellytykset [SAS, 2005].

Käytettävät tekniikat riippuvat toteutettavasta sovelluksesta. On tärkeää pystyä valitsemaan sellaiset tekniikat, joilla voidaan täyttää sovelluskohtaiset tarpeet. Jotta



Kuva 15: Laadun perusedellytykset [SAS, 2005]

tekniikka osaltaan parantaisi toteutuksen laatua ja luotettavuutta, tulisi sen täyttää seuraavat kriteerit:

- Kypsyys: Tekniikan tulee olla jo käytännössä toimivaa. Täysin uutta tekniikkaa käytettäessä tulee ottaa huomioon sen mahdolliset ongelmat.
- Avoimuus ja yhteensopivuus: Standardin mukaisen ja/tai avoimen tekniikan käytöllä parannetaan yhteensopivuutta.
- Sopivuus: Tekniikan on sovelluttava kyseiseen kohteeseen ja täytettävä sen erityisvaatimukset.
- Saatavuus: Tuen saatavuus tekniikalle läpi järjestelmän elinkaaren on tärkeää.

Ohjelmistotuotantoprosessin laatu ja luotettavuus on sidoksissa siihen, miten hyvin sen eri osa-alueet saadaan suoritettua. Näiden toimenpiteiden määrittelyn ja hallinnan tasoa kuvataan tuotantoprosessin kypsyydellä. Standardissa ISO 15504-2 on määritelty yrityksen prosesseille kypsyystasot [ISO 15504-2, 1998]:

1. Alustava: Toiminta on ad-hoc-tyyppistä, jossa mikä tahansa toimiva tapa kelpaa ratkaisuksi. Toimintatapoja ei ole määritelty ja onnistuminen riippuu yksilösuorituksista

2. Toistettava: Projektinhallintaa käytetään aikataulun, kustannusten ja toiminnallisuuden hallitsemisessa. Aiemmat onnistumiset pystytään toistamaan vastaavissa uusissa hankkeissa.

3. Määritelty: Teknisen toiminnan ja projektinhallinnan kuvaukset on dokumentoitu ja kaikki projektit toteutetaan käyttäen dokumenttien mukaista vakioprosessia.

4. Hallittu: Valmista tuotetta ja sen tekoprosessia tarkastellaan mittarien avulla. Molempia voidaan kuvata ja kontrolloida käyttäen kvantitatiivisia menetelmiä.

5. Optimoiva: Prosessia pyritään optimoimaan jatkuvasti kvantitatiivisen tiedon avulla.

Kypsymättömän prosessin yhteydessä toiminta ei ole kunnolla suunniteltua tai määriteltyä. Saavutettu laatu riippuu keskeisten yksilöiden henkilökohtaisesta panoksesta. Pahimmassa tapauksessa toiminta voi olla hyvinkin sekavaa ja kaoottista, eikä sitä pystytä hallitsemaan.

Kypsää prosessia sen sijaan on on helpompi hallita, koska sen yhteydessä käytetään elinkaarimallia, jossa tuotanto on jaettu vaiheisiin ja se on dokumentoitu. Vaiheiden välissä suoritetaan järjestelmällisiä tarkastuksia. Kypsän prosessin kautta on mahdollista parantaa laatua ja varsinkin tasata sen vaihtelua, mm. koska tuotanto ei ole niin riippuvainen tietyistä henkilöistä.

Epäkypsällä tuotantoprosessilla voidaan myös saada aikaan hyvälaatuinen ohjelmisto, mutta tällöin kyseessä saattaa olla ainoastaan onnekas sattuma eikä sen perusteella voida päätellä yrityksen yleisen laatutason olevan hyvä.

Dynaaminen testaaminen on merkittävä tekijä laadunvarmistuksessa. Sitä tukevat staattinen testaaminen ja erilaiset tarkastukset. Kattava testaaminen on tehokas keino havaita mahdolliset virheet tuotteessa ja korjata ne ajoissa. Kokemus on osoittanut kustannusten nousevan moninkertaisiksi jos virhe havaitaan vasta tuotteen toimituksen jälkeen verrattuna siihen että virhe havaitaan ja korjataan ennen toimitusta.

Osaava ja motivoitunut henkilöstö on tärkeä voimavara mille tahansa yritykselle. Ilman sitä on lähes mahdotonta saavuttaa korkealaatuista lopputuotetta.

Henkilöstön osaamista voidaan parantaa koulutuksella ja perehdyttämisellä. Uuden työntekijän osaamisen perustana toimii sopiva pohjakoulutus ja aiempi työkokemus. Tämän lisäksi uudelle työntekijälle on annettava riittävä opastus käytettävistä menetelmistä ja työkaluista, kuten kehitysympäristöistä ja niihin tarjolla olevista ohjelmakirjastoista. Teknologiat kehittyvät jatkuvasti ja yhä nopeampaan tahtiin, mistä syystä on tärkeää ylläpitää osaamista päivittämällä henkilöstön tiedot ja taidot aina tarpeen vaatiessa. Käytettävien työkalujen tulisi olla ajan mukaisia ja niiden tarjoamia mahdollisuuksia pitäisi pystyä käyttämään tehokkaasti hyväksi. [SAS, 2005]

Nykyään työskennellään erilaisissa ryhmissä ja tiimeissä, joissa saadaan yhdistettyä yksilöiden työpanokset ja erilaiset vahvuudet. Yhteistyö on tullut tarpeelliseksi

järjestelmien kasvaessa ja monimutkaistuessa, jolloin yhden henkilön on erittäin vaikeaa ellei jopa mahdotonta hallita koko järjestelmä yksityiskohtaisesti. Lisäksi on tärkeää varmistaa tiedon välittyminen niin tiimin sisällä kuin eri tiimien välillä. Tiedon jakaminen hoidetaan esimerkiksi dokumenttien avulla tai keskustelemalla. Lyhyt ajatusten vaihto kehittäjien kesken voi useasti olla nopeampi ja joustavampi keino tiedon jakamiseen. [SAS, 2005]

Tiedon jakamisella pyritään lisäksi estämään tilanne, jossa kaikki tietämys jostain asiasta on vain yhden henkilön hallussa eikä dokumentteja aiheesta ole olemassa [SAS, 2005]. Jos tällainen henkilö on esimerkiksi lomalla, sairas, kiinni toisessa projektissa tai pahimmassa tapauksessa ei edes työskentele enää yrityksessä, hänen tietonsa eivät ole käytettävissä ja näin ollen projektin onnistunut toteuttaminen on uhattuna.

Aikataulut ovat monesti tiukkoja ja aiheuttavat kiirettä jo valmiiksi paljon kuormitetuille resursseille. Kiireen myötä virhealttius kasvaa ja mahdollisuudet hyvälaatuiseen lopputulokseen pienenevät. Tästä syystä on tärkeää, että resursseja kuormitetaan kohtuudella ja aikatauluissa on liikkumavaraa. [SAS, 2005]

Oleellinen tekijä laadukkaaseen tekemiseen pyrittäessä on henkilöstön motivaatio. Motivoituneiden työntekijöiden avulla on helpompi saavuttaa haluttu lopputulos. Motivaatiota voidaan parantaa panostamalla kannustavaan ilmapiiriin ja hyvään ryhmähenkeen. [SAS, 2005]

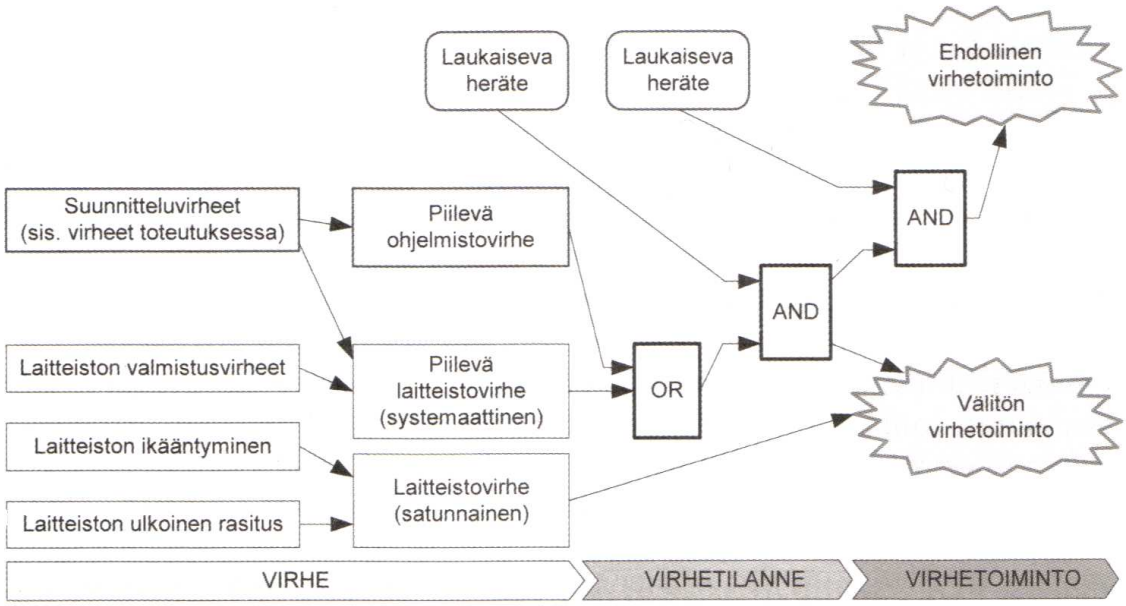
5.3 Sisäisen laatu ohjelmistossa

Ohjelmistovika ymmärretään tilanteena, jossa ohjelma ei toimi halutulla tavalla. Ohjelmiston virhemekanismien kautta voidaan ohjelmiston laatua tarkastella sisäisesti. Ohjelmiston vikoja varten on standardissa IEEE 610.12 [IEEE, 1990] esitelty seuraavat termit: Error, Fault ja Failure.

Ohjelmistovirheet (errors) johtuvat puuttuvista tai vääristä inhimillisistä toiminnoista. Esimerkkejä ohjelmistovirheistä ovat koodausvirheet (esimerkiksi viitataan väärään muuttujaan), suunnitteluvirheet (esimerkiksi ei oteta huomioon kaikkia mahdollisia tilanteita) ja määrittelyvirheet (esimerkiksi aliarvioidaan suorituskyky). Koodausvirheitä voidaan jossain määrin vähentää nykyisten ohjelmointiympäristöjen avulla, koska ne pystyvät monesti havaitsemaan yksinkertaiset virheet itsenäisesti.

Virhetilanne (fault) johtuu ohjelmistossa olevasta virheestä. Tietyssä tilanteessa virhetilanteen saa aikaan sopiva heräte (trigger), jollaisena voi toimia joko väärä syöte tai väärä ajoitus. Heräte itsessään voi olla lähtöisin ympäristöstä (esimerkiksi kaapelirikko), ihmisestä (esimerkiksi väärä komento) tai ohjelman sisältä (puskurin ylivuoto). Virhetoiminta (failure) on ohjelmiston normaalista tai käyttäjän siltä odottamasta käyttäytymisestä poikkeava toiminta, joka johtuu virhetilanteesta. Virhetoiminto näkyy ohjelmiston ulkopuolella, esimerkiksi järjestelmä jumiutuu tai kaatuu kokonaan.

Ohjelmistovirheet, virhetilanteet ja virhetoiminnat muodostavat siis ketjun (kuva 5-4). Ohjelmistovirhe voi aiheuttaa virhetilanteen, joka edelleen voi aiheuttaa virhetoiminnan. Ketju ei välttämättä toteudu ”yksi- yhteen”. Yksi virhe voi aiheuttaa useita virhetilanteita ja yksi virhetilanne voi aiheuttaa puolestaan useita virhetoimintoja. Toisaalta on myös mahdollista että virhetoiminto johtuu useasta virhetilanteesta ja virhetilanne useasta virheestä. Virhemekanismin ketju voi olla myös vajaa, eli virhetilanne ei välttämättä aiheuta virhetoimintaa eikä virhe välttämättä aiheuta virhetilannetta. Ohjelmistossa olevan piilevän virheen eteneminen virhetilanteeksi riippuu siis olosuhteista. Ohjelmistovirhe on aina systemaattinen ja poikkeaa siten laitteistovirheestä, joka voi olla joko systemaattinen (esimerkiksi laitteiston suunnittelu- ja valmistusvirhe) tai satunnainen (esimerkiksi laitteiston kulumisesta tai ikääntymisestä johtuva). Koska virhetoiminto riippuu olosuhteista ja muodostumisessa tarvittava virhemekanismi voi olla hyvin monimutkainen, ei virhetoiminta välttämättä käyttäjän kannalta vaikuta systemaattiselta vaan satunnaiselta. [SAS, 2005]



Kuva 16: Ohjelmiston virhemekanismit [SAS, 2005]

Jotta ohjelmistoviat voitaisiin estää tai niiden vaikutusta pienentää, on vian muodostumiseen liittyvän virhemekanismin ketju saatava poikki. Laatua varmistavat ja virheellistä käyttäytymistä estävät toimet voidaan jakaa kolmeen kategoriaan [Tian, 2005]:

- vian estäminen: virheiden syntymisen estäminen ja virhelähteiden poistaminen
- vian vähentäminen: virhetilanteiden havaitseminen ja poistaminen
- vian rajaaminen: virhetoiminnan estäminen ja sen seurauksen rajaaminen

Vikojen syntymistä voidaan estää työskentelemällä huolellisesti ja käyttämällä hyvää ja kypsää ohjelmistotuotantoprosessia. Virheitä estäviä keinoja ovat mm. kehittäjien kouluttaminen ja harjaannuttaminen. Virhetilanteita voidaan poistaa testaamalla ja tarkastamalla ohjelmistoa ja sen osia jo kehitysvaiheessa.

Virheiden estäminen ja virhetilanteiden poistaminen ohjelmistosta on siinä mielessä tehokas toimenpide, että ilman ainuttakaan virhettä ei synny myöskään virhetilanteita eikä virhetoimintoja. Käytännössä ohjelmistojen kasvaminen ja monimutkaistuminen tekee kuitenkin kaikkien virheiden eliminoinnin mahdottomaksi.

Järjestelmän vikasietoisuudella tarkoitetaan sitä, että virhetilanteet eivät pääse etenemään virhetoiminnoiksi ja että mahdollisten virhetoimintojen vaikutus jää mahdollisimman pieneksi. Vikasietoisuutta voidaan parantaa mm. poikkeustilanteiden hallinnalla ja niistä toipumisella.

5.4 Ulkoinen laatu ohjelmistossa

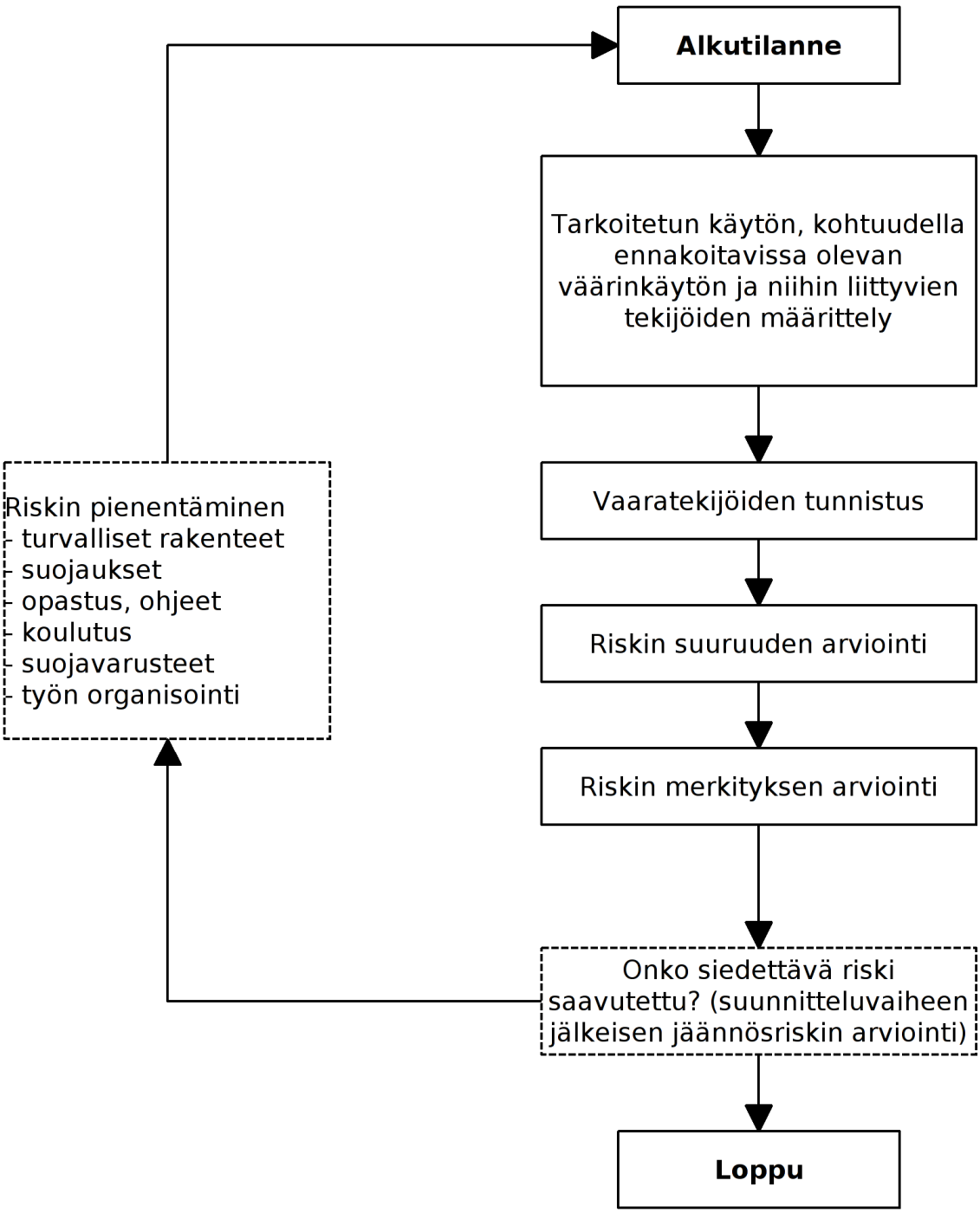
Kun järjestelmää ja sen laatua tarkastellaan järjestelmän toimintaympäristössä, puhutaan ulkoisesta laadusta. Ulkoisia laatutekijöitä ovat järjestelmän luotettavuus ja turvallisuus [SAS, 2005]. Luotettavuus koostuu käyttövarmuuden ja sen osatekijöiden (kuten toimintavarmuus, huollettavuus ja huoltovarmuus) muodostamasta kokonaisuudesta. Kun järjestelmä on vapaa tarpeettomista riskeistä, sen sanotaan olevan turvallinen.

Riskit on pyrittävä tiedostamaan jo ohjelmiston suunnittelussa. Turvallisuutta ja luotettavuutta koskevien vaatimuksien selvittämiseksi tehdään riskianalyysi, johon tekninen toteutus jatkossa perustuu. Tyypillisiä riskianalyysin menetelmiä ohjelmistotekniikassa ovat vika-vaikutusanalyysi ja erilaiset syy ja seuraus -kaaviot. Riskianalyysiä tehtäessä on otettava huomioon järjestelmän laajuus ja siitä johtuvat tarpeet sekä varmistettava analyysin riittävä kattavuus systemaattisella toiminnalla. [SAS, 2005]

Riskianalyysin pohjalta järjestelmä ja sen osat, toiminnot ja laitteet voidaan luokitella erilaisiin riskitasoihin. Eri sovelluksiin on olemassa erilaisia luokitteluja. Luokittelun avulla voidaan laatua varmistavat toimet kohdentaa paremmin, jolloin esimerkiksi järjestelmän kriittisimmät osat ja toiminnot testataan muita kattavammin. Lisäksi luokittelulla voidaan vaikuttaa teknisiin ratkaisuihin ja parantaa niiden laatua, esimerkiksi kriittisiksi osoittautuneet järjestelmän osat, kuten virransyöttö ja tietoliikenneyhteydet, voidaan varmentaa kahdentamalla. [SAS, 2005]

5.5 Laatu käytännössä

Automaatiojärjestelmät ovat aiemmin olleet yksinkertaisempia ja sisältäneet yksinkertaisempi sovelluksia. Mahdolliset virheet on saatu suhteellisen helposti poistettua. Järjestelmien monimutkaistuessa virheitä on helpompi tehdä, mutta vaikeampi löytää ja poistaa. Nykyiset järjestelmät sisältävät logiikkaohjelmien lisäksi yhä



Kuva 17: Riskin arvioinnin vaiheet [SAS, 2005]

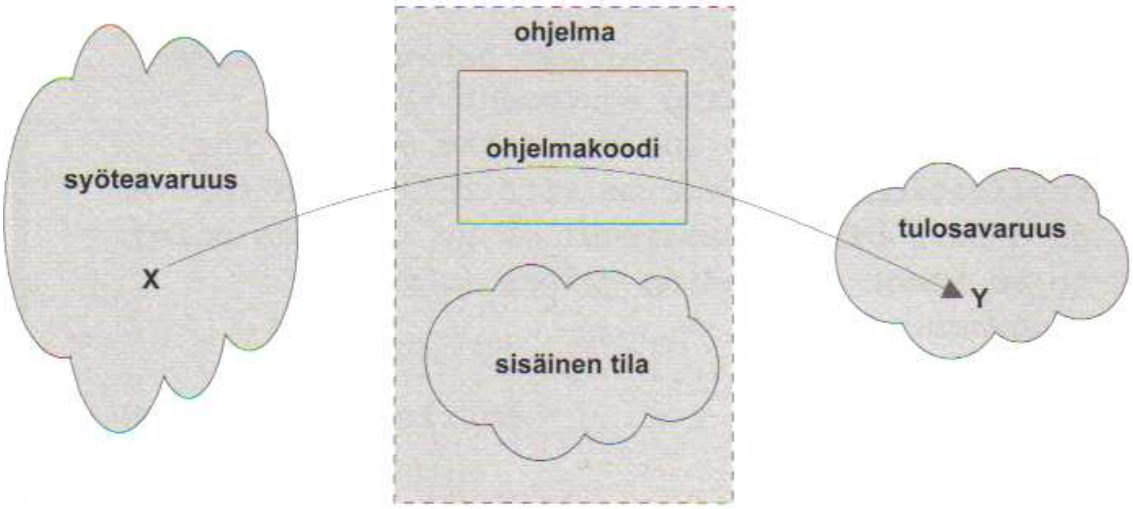
enemmän myös tietokoneella ajettavia ohjelmia, ja samassa projektissa työskentelee niin perinteisiä logiikkaohjelmoijia kuin myös vaikkapa java-sovelluskehittäjiä. Koodin kokonaismäärä on kasvussa, ja siksi tarvitaan menetelmiä joilla joilla sovellusten laatua saadaan parannettua. Logiikkaohjelmointiin voidaan soveltaa samoja menetelmiä mitkä ovat käytössä yleisesti ohjelmistokehityksessä, kuten strukturoitua

ohjelmointia, koodin uudelleenkäyttöä ja yhtenäisiksi sovittuja käytäntöjä. Näillä voidaan saavuttaa huomattavia säästöjä ohjelmointiin ja testaukseen käytettävässä ajassa, mikä on tärkeä etu tavoiteltaessa projektien nopeampaa läpivientiä. Myöhemmissä kappaleissa tullaan esittelemään logiikkaohjelmoinnin menetelmiä, joilla pyritään saamaan sovellusten laatua paremmaksi ja kehitysprosessi kypsemmäksi, kuten ohjelman jakamista osiin (strukturointi), uudelleenkäytettäviä toimilohkoja ja kirjastoja sekä logiikkaohjelman symbolien ja muuttujien yhtenäistä nimeämiskäytäntöä.

5.6 Testaus

Ohjelmiston laadun arvioimiseen voidaan käyttää sekä staattisia menetelmiä että dynaamista testausta. Staattisissa menetelmissä tarkastellaan ja analysoidaan itse ohjelmakoodia, mutta koodia ei suoriteta. Kun ohjelmaa tai sen osaa suoritetaan ja siitä etsitään suunnitelmallisesti virheitä, puhutaan testauksesta [Haikala & Märijärvi, 2002]. Monia yleisen ohjelmistotekniikan testausmenetelmiä voidaan käyttää myös automaatio-ohjelmistojen testaukseen. Testausta pidetään yhtenä tärkeimmistä ohjelmiston laatua varmistavana toimenpiteenä ja sitä myös käytetään hyvin usein [Tian, 2005].

Testaus pitää sisällään useita vaiheita: suunnittelu, sopivan testiympäristön rakentaminen, itse testin suorittaminen sekä tulosten analysointi. Koska testaukseen ja siinä mahdollisesti havaittujen virheiden jäljittämiseen ja korjaamiseen kuluu helposti jopa yli puolet ohjelmistoprojektin resursseista, on tärkeää suorittaa testaus optimaalisesti [Haikala & Märijärvi, 2002]. Testauksen määrä on kompromissi käytettävissä olevien panostusten (aika, raha, välineet, yms.) ja saavutetun luotettavuuden välillä. Testauksen periaate on esitetty kuvassa 6-1.



Kuva 18: Ohjelmiston testaus [Haikala & Märijärvi, 2002]

Testissä ohjelmalle annetaan syöte X ja tulokseksi saadaan Y. Syöte voi olla esi-

merkiksi napin painallus tai lämpötila-arvo mittauksesta. Vastaavasti tulos voi olla esimerkiksi merkkilampun syttyminen tai moottorin käynnistyminen. Saatu tulos ei riipu pelkästään syötteestä, vaan lisäksi siihen vaikuttaa ohjelman sisäinen tila. Sisäinen tila käsittää esimerkiksi ohjelman muuttujien ja rekisterien arvot ja järjestelmän laitteen tilan, kuten venttiilin kiinni-tiedon. Jos sekä tulos Y että ohjelman sisäisen tilan muutos ovat oikein, on testi onnistunut. Testaustilanteessa on välttämätöntä, että syöte- ja tulosavaruus ymmärretään riittävällä tarkkuudella. Lisäksi on tiedettävä mikä on oikea tulos. Tämän vuoksi on oltava olemassa niin määrittely- ja suunnitteludokumentteja, kuin kokemuksen mukanaan tuomaa tietoa, minkä avulla saadaan määritellyksi hyväksyttävissä olevat lopputulokset.

Testaus voidaan kohdistaa järjestelmän rakenteeseen, käyttöön ja toimintaan [SAS, 2005]. Rakenteellinen testaus kattaa mm. komponentit ja niiden väliset rajapinnat. Käyttötestauksella pyritään selvittämään järjestelmän toimintaa käyttäjän kannalta. Toimintaan kohdistuvassa testauksessa testataan järjestelmän toiminnot. Testaamalla voidaan löytää ohjelmistosta virheitä ja ne voidaan jäljittää ja korjata. Jos virheitä ei testaamalla löydy se ei tarkoita etteikö niitä silti voisi ohjelmistossa olla [Haikala & Märijärvi, 2002]. Testaukseen käytettävät resurssit ovat rajalliset ja monimutkaisten virhemekanismien kautta testattavien tilanteiden määrä voi nousta erittäin suureksi.

5.6.1 Testauksen tasot

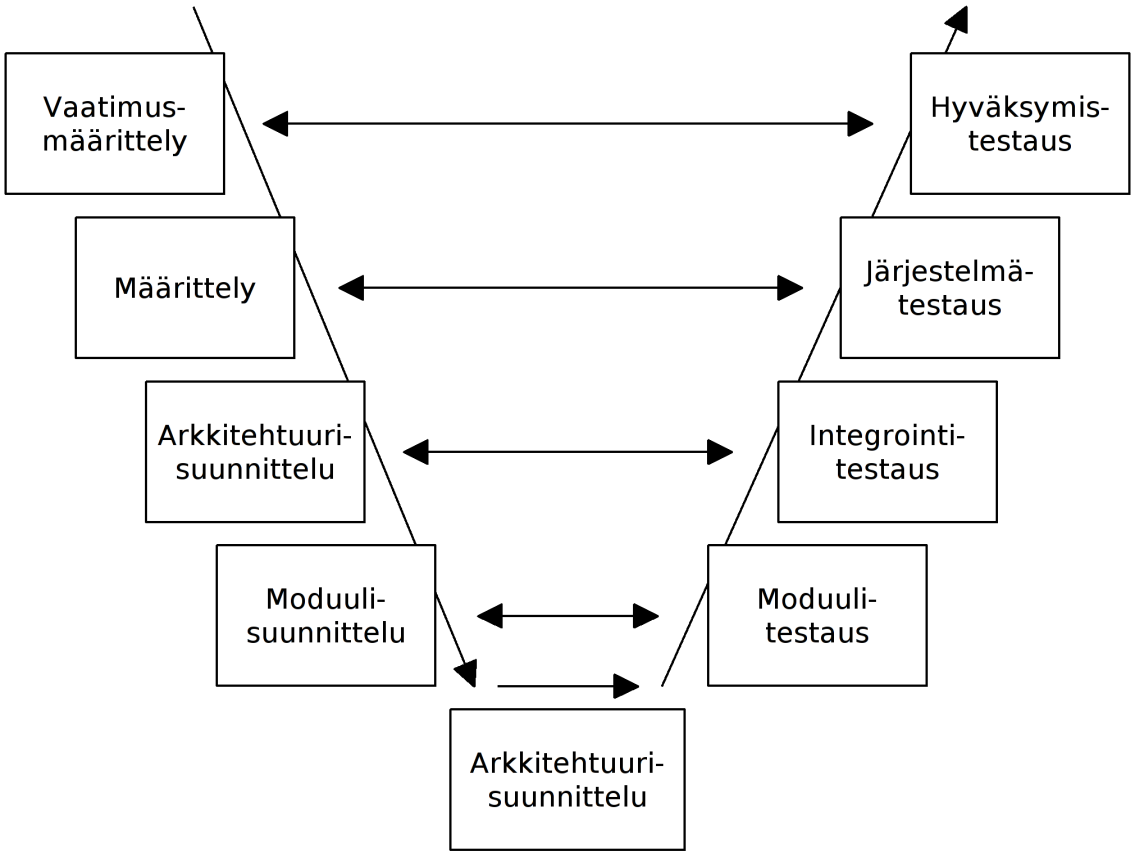
Testaus on tärkeää korkealaatuiseen ohjelmistoon pyrittäessä ja testaus voi kuluttaa ison osan ohjelmistoprojektin resursseista, kuten edellä todettiin. Testauksen tulisikin olla omana prosessinaan ohjelmistokehitysprosessin rinnalla, ellei kyse ole aivan yksinkertaisesta ja pienestä ohjelmasta [SAS, 2005].

Testauksen V-malli pitää sisällään testauksen eri tasot (kuva 6-2), joita ovat moduulitestaus, integrointitestaus, järjestelmätestaus ja hyväksymistestaus.

Testauksen suunnittelu tehdään jokaista testaustasoa vastaavalla suunnittelutasolla: hyväksymistestaus suunnitellaan vaatimusten määrittelyn yhteydessä, järjestelmätestaus määrittelyvaiheessa, integrointitestaus arkkitehtuurisuunnittelun aikana ja moduulitestaus moduulisuunnittelun aikana. Vertaamalla testituloksia suunnitteludokumentteihin voidaan tulokset todeta oikeiksi (Kuva 20) [Stenberg, 2003].

V-mallissa korkeammalle testaustasolle voidaan siirtyä vasta sen jälkeen, kun nykyisen tason testaus on saatu valmiiksi. Jo suoritettuja testejä ei tarvitse enää toistaa korkeammilla tasoilla. Kun testaus suunnitellaan sitä vastaavalla suunnitteluja määrittelytasolla, saadaan osa virheistä eliminoitua ennen kuin suunnittelussa siirrytään alemmalle tasolle.

Suunnittelu- ja määrittelyvaiheessa (V-mallin vasen haara) ohjelmakoodille ja dokumenteille tehtävät tarkastukset ja katselmukset ovat ns. staattista analyysiä. Koska tarkastus kohdistuu V-mallin vasempaan haaraan, virheet löytyvät aikaisemmin ja samalla on mahdollista estää uusien virheiden syntyminen. Tarkastukset on mahdollista kohdistaa seikkoihin, joita ei voida testata tai jotka eivät löydy testaamalla,



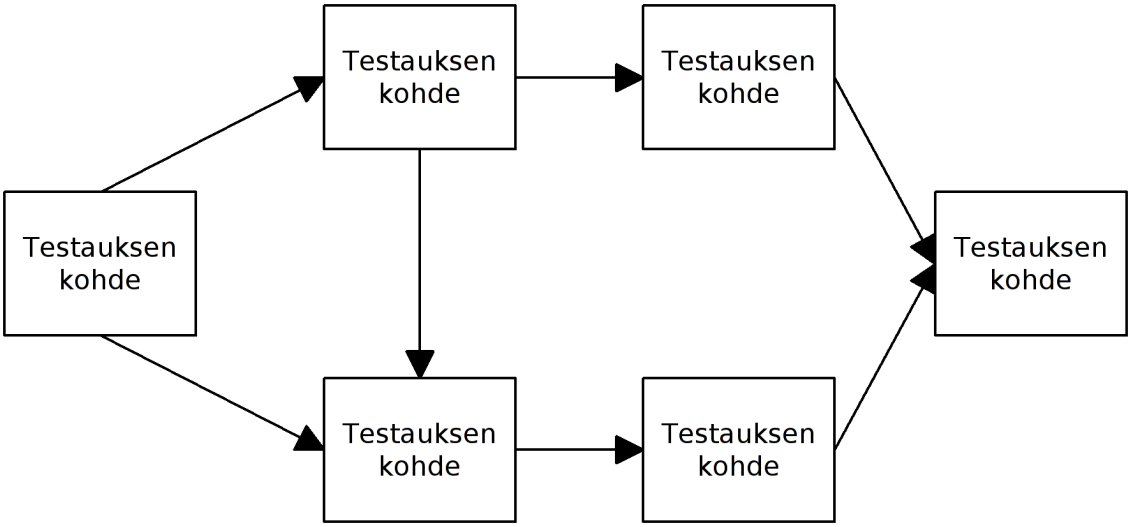
Kuva 19: Testauksen V-malli [Haikala & Märijärvi, 2002]

kuten tyylioppaan vastaiset virheet, erilaiset määrittelyvirheet ja turha ohjelmakoodi [Haikala & Märijärvi, 2002]. Tarkastukset eivät kuitenkaan kokonaan poista testauksen tarvetta, mutta niillä on mahdollista vähentää virheiden määrää jo ennen testausta, jolloin testauksen työmäärä ja siihen kuluva aika vähenee. Tämän seurauksena koko projektin läpivieminen nopeutuu (kuva 6-4).

Tarkastusten käyttäminen projekteissa ei kuitenkaan ole itsestään selvää. Tarkastukset voivat kuormittaa liikaa projektin avainhenkilöitä tai tarkastuksiin suhtaudutaan negatiivisesti, jolloin ne jäävät käyttämättä.

Moduuli- eli yksikkötestauksessa testataan yksittäisiä ohjelmamoduuleja. Moduulitestaus suoritetaan yleensä lasilaatikkoperiaatteella. Tällöin voidaan myös tarkastaa itse ohjelmakoodi [SAS, 2005]. Moduulit testataan suunnitteluspesifikaatioita vasten, ja moduulitestauksella varmistetaan että jokainen moduuli toimii itsenäisesti oikein. Jos testattava moduuli on laiteläheinen (esimerkiksi moottorin tai taajuusmuuttajan ohjaus) on testaajan tunnettava laitteen toiminta. Samalla voidaan käyttää apuna oikeita laitteita ja toiminnallista testausta. Moduulitestauksen suorittaa yleensä moduulin ohjelmoija itse. Pyrittäessä täydelliseen virheettömyyteen käytetään kuitenkin erillistä testaajaa.

Integrointitestauksessa useampia moduuleja on yhdistetty toisiinsa ja niiden muo-



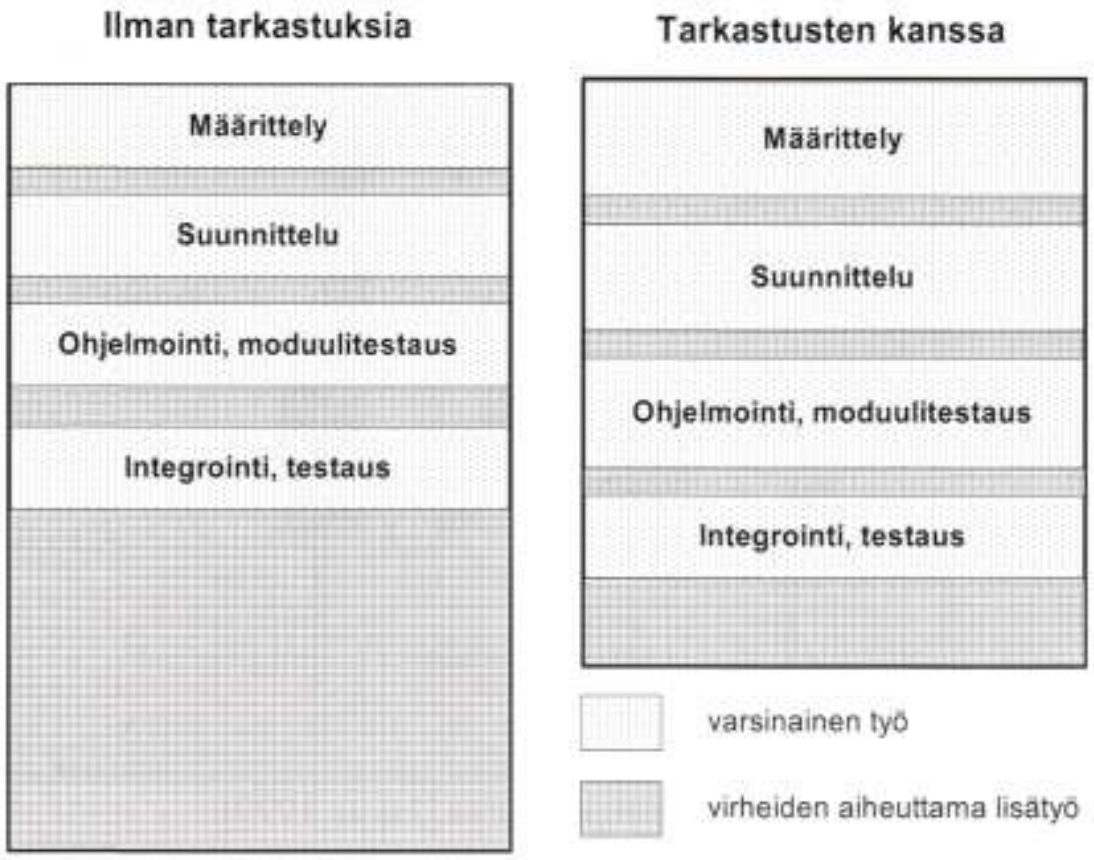
Kuva 20: Testauksen periaate [Stenberg, 2003]

dostamaa kokonaisuutta (esimerkiksi osajärjestelmää) testataan. Näin pyritään varmistamaan, että tiedot välittyvät oikein moduulien välillä ja että moduulit toimivat yhdessä oikein. Mahdollisuuksien mukaan pyritään testaamaan myös yhteensopivuus ulkopuolisten järjestelmien kanssa.

Integrointitestauksessa kaikki moduulit voidaan testata yhtenä isona kokonaisuutena. Tämä vie helposti paljon aikaa ja virheiden paikallistaminen hankaloituu. Toinen vaihtoehto on testata vain osa moduuleista, jolloin tarvitaan apuohjelmia ja ajureita paikkaamaan puuttuvia moduuleja. Tyypillisesti integrointitestaus tehdään kokoavasti (bottom up) tai jäsentävästi (top down). Kokoavasti edetessä liikutaan alemman tason moduuleista ylöspäin ja jäsentävästi edetessä päinvastoin. Vaihtoehtoisesi integrointitestaus voidaan suorittaa toiminnallisin kokonaisuuksin. Tämä sopii hyvin tapauksiin, joissa automaatio- ohjelmiston arkkitehtuuri on suunniteltu noudattaen koneiden ja laitteiden toiminnallista rakennetta. Kaikki osat, jotka liittyvät tiettyyn toimintoon, voidaan tässä tapauksessa testata kerralla.

Järjestelmätestauksessa testataan sen nimen mukaisesti koko järjestelmän toiminta, jossa on mukana automaatio, mekaniikka, ja muut ulkopuoliset järjestelmät. Lisäksi testataan ei-toiminnalliset ominaisuudet: suorituskyky, luotettavuus, käytettävyys jne. Automaatioprojekteissa tärkeä vaihe on tehdastestit (FAT), joissa osa järjestelmätestauksesta tehdään toimittajan tiloissa ennen järjestelmän siirtämistä lopulliseen kohteeseensa. Näin voidaan merkittävästi helpottaa järjestelmän käyttöönottoa kohteessa, koska nyt mahdollisten virheiden määrä on etukäteen pienempi ja ne on helpompi paikallistaa. Käyttöönotot ovat usein hektisiä tilanteita, varsinkin jos ohjattava laitos tai prosessi on koko ajan toiminnassa tai toimintaan ei haluta pitkiä katkoksia.

Lopuksi voidaan suorittaa hyväksymistestaus, jossa asiakas on mukana. Hyväksymistestauksen tarkoituksena on osoittaa mm. asiakasvaatimusten täyttyminen. Hyväksymistestaus tehdään tuotannollisen koeajon aikana ja samalla voidaan antaa



Kuva 21: Virheiden aiheuttama lisätyö [Haikala & Märijärvi, 2004]

käyttäjille koulutusta ja opastusta järjestelmään.

5.6.2 Testauksen menetelmät

Testitapausten valinnassa on kaksi peruslähtökohtaa: lasilaatikkotestaus (glass/white box testing) ja mustalaatikkotestaus (black box testing) [Tian, 2005]. Näistä lasilaatikkotestaus on rakenteellinen ja mustalaatikkotestaus toiminnallinen testausmenetelmä. Lasilaatikkotestauksessa täytyy tuntea ohjelman sisäinen toteutus, jonka pohjalta testitapaukset suunnitellaan. Lasilaatikkotestausta käytetään lähinnä moduulitestauksessa. Mustalaatikkotestauksessa ei tarvitse tuntea ohjelman toteutusta, vaan testitapaukset valitaan ohjelman spesifikaatioiden perusteella. Harmaalaatikkotestauksessa on piirteitä sekä lasi- että mustalaatikkotestauksesta. Siirryttäessä V-mallissa alemmalta tasolta ylemmälle, muuttuu testitapausten luonne lasilaatikkotestauksesta mustalaatikkotestaukseen päin.

Kun sopiva testiympäristö on saatu luotua, suoritetaan halutut testitapaukset manuaalisesti tai automaattisesti. Automaatio-ohjelmiston testaus on yleensä manuaalista, koska testaustyökalut ovat rajoittuneita eikä testauksen automatisointi ole välttämättä helppoa.

Testauksen aikana on tärkeää pitää kirjaa siitä, mitä on testattu ja millaisia tuloksia on saatu. Saatuja tuloksia verrataan odotettuihin tuloksiin, minkä perusteella saadaan selville täyttyvätkö asetetut kriteerit. Testaussuunnitelmassa on yleensä esitetty lopetuskriteerit, jotka on täytettävä ennen kuin testaus voidaan lopettaa. Mahdollisia lopetuskriteerejä ovat mm. rajapinnat, käyttötapaukset, aika ja löydettyjen virheiden määrä [SAS, 2005].

Testaukseen kuluu helposti paljon aikaa eikä kaikkia virheitä silti löydetä. Usein olisi toivottavaa löytää ja poistaa ainakin kaikkein pahimpiin virhetilanteisiin johtavat virheet mahdollisimman tehokkaasti. Testausta voidaan nopeuttaa käyttämällä jo testattua koodia uudestaan ja suunnittelemalla sovellus helpommin testattavaksi. Logiikkaohjelman koodia voidaan käyttää uudestaan luomalla kirjastoja, joista ohjelmoija voi kopioida valmiiksi testattua ja toimivaa koodia sovellukseen. Testaustapahtuma saadaan puolestaan helpommaksi käyttämällä ohjelmassa selkeitä muuttujien nimiä. Näitä menetelmiä tarkastellaan lähemmin seuraavissa kappaleissa.

6 Kehitysympäristöt

Seuraavaksi tarkastellaan kahden Mipro Oy:ssä käytettävän kehitysympäristön ominaisuuksia. Tarkasteltavana ovat Schneider Electric:in Unity Pro ja Siemensin Step 7, jotka ovat varsin suosittuja ympäristöjä.

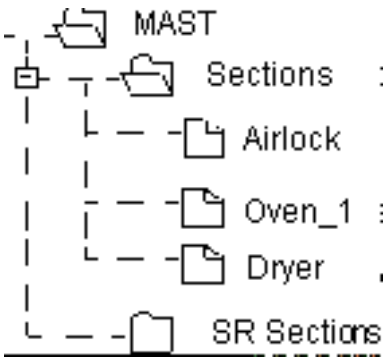
6.1 Unity Pro -kehitysympäristö

Unity Pro on ranskalaisen Schneider Electric:in valmistamille ohjelmoitaville logiikoille tarkoitettu kehitysympäristö. Unity Pro:ta voidaan käyttää seuraavien Schneider Electric:in logiikoiden ohjelmoimiseen: Premium, Atrium ja Quantum. Lisäksi Unity Pro tarjoaa välineet ohjelmien testaukseen, diagnosointiin ja virheiden jäljitykseen.

Unity Pro vaatii toimiakseen järjestelmän, jossa on vähintään Pentium-prosessori 1.2 GHz:in kellotaajuudella (suositus 2.4 GHz) ja 512 megatavua muistia. Vapaata tilaa kiintolevyllä tulisi olla noin 2-4 gigatavua. Tuettuja käyttöjärjestelmiä ovat Microsoft Windows 2000 ja Microsoft Windows XP Professional Edition, minkä lisäksi vaatimuksena on MS Internet Explorer 5.5 tai uudempi.

Ohjelmien tekemiseen voidaan Unity Pro:ssa käyttää kaikkia viittä standardin IEC-61131-3 mukaisia kieliä (FBD, LD, IL, ST ja SFC). Sovellusohjelma koostuu pakollisesta pääelementistä (Master Task) ja siihen mahdollisesti liittyvistä lisäelementeistä (mm. Fast task ja HelpTask). Elementit koostuvat puolestaan osioista (Sections) ja alirutiineista (subroutines). Osioiden avulla ohjelma voidaan jakaa toiminnallisiin osiin, jolloin ohjelmakokonaisuutta on helpompi hallita.

Osiot ovat itsenäisiä kokonaisuuksia, joten hyppykäsky osiosta toiseen on mahdoton. Osio on mahdollista liittää kerralla vain yhteen elementtiin ja osioiden suoritusjärjestys riippuu niiden ryhmittelyjärjestyksestä elementin alle (kuva 7-1).



Kuva 22: Osiot Unity Pro:ssa [Schneider Electric, 2006]

Unity Pro tarjoaa laajan joukon tietotyyppejä käytettäväksi tiedon varastointiin ja käsittelyyn. Tuettuja tietotyyppejä ovat mm. BOOL (TRUE/ FALSE), EBOOL (nousevan tai laskevan reunan tunnistus), 16- ja 32-bittiset bittijonot, 16- ja 32-

bittiset kokonaisluvut etumerkillä tai ilman sekä 32-bittiset liukuluvut. Lisäksi tarjolla on tietotyyppejä merkki- ja bittijonojen, sekä ajan ja päivämäärän esittämiseen. Äskeisten perustietotyyppien lisäksi on mahdollista käyttää johdettuja tietotyyppejä: taulukoita ja struktuureja.

Unity Pro tukee toimilohkokirjastoja. Kehitysympäristön mukana tulee valmiita kirjastoja, minkä lisäksi käyttäjä voi luoda itse omia kirjastojaan. Yksi valmiista kirjastoista on peruskirjasto, joka sisältää ohjelmointityössä yleisesti käytettäviä funktiota, kuten loogiset ja matemaattiset operaatiot sekä funktiot merkkijonojen ja päivämäärän ja kellon ajan käsittelemiseen. Lisäksi löytyy valmiit funktiot joilla voidaan konvertoida dataa tietotyypistä toiseen. Säätökirjasto sisältää funktiota mm. säätöihin, mittausten ja lähtöjen käsittelyyn ja asetusarvojen hallintaan liittyen. Muita valmiita kirjastoja ovat tietoliikennekirjasto, I/O-kirjasto, liikkeenohjauskirjasto, systeemikirjasto, diagnostiikkakirjasto ja TCP Open -kirjasto. TCP Open kirjasto mahdollistaa tcp pohjaisen tiedonsiirron automaatiosovelluksessa. Vaatimuksena on joko ethernet-portilla varustettu keskusyksikkö tai erillinen ethernet-lisäkortti. Valmiiden kirjastojen sisältämien toimilohkojen ja funktioiden lisäksi käyttäjällä on Unity Pro:ssa mahdollisuus luoda omia sovelluskohtaisia toimilohkoja (Derived Function Block, DFB). Käyttämällä omia toimilohkoja voidaan huomattavasti tehostaa ohjelman kirjoitusta sekä saadaan ohjelmakoodista helpompilukuista. Oman toimilohkon käyttäminen pitää sisällään kolme vaihetta:

1. luodaan toimilohkon malli
2. mallista tehdään kopio, eli ilmentymä, aina kun sitä halutaan käyttää sovelluksessa
3. ilmentymää käytetään sovelluksessa

Uutta käyttäjätoimilohkoa luotaessa sille annetaan ensiksi nimi. Nimi voi olla 32 merkkiä pitkä ja sen pitää olla yksiselitteinen (eli samassa projektissa tai kirjastossa ei voi useaa saman nimistä toimilohkoa). Tämän jälkeen määritellään toimilohkon parametrit eli sisään- ja ulostulot, läpiviennit sekä sisäiset ja julkiset muuttujat. Yhdessä toimilohkossa voi olla 32 kappaletta sisääntuloja ja 32 kappaletta ulostuloja. Läpivientien ja sisääntulojen tai läpivientien ja ulostulojen yhteenlaskettu lukumäärä voi olla enimmillään 32. Sisäisiä ja julkisia muuttujia voi olla rajoittamaton määrä. Parametrit on nimettävä enimmillään 32 merkillä. Sallittuja merkkejä ovat kirjaimet, numerot ja alaviivat. Alaviivoja ei sallita peräkkäin. Isojen ja pienien merkkien välillä ei ole eroa. Ensimmäinen merkki voi kirjain tai alaviiva. On mahdollista ottaa käyttöön projektikohtainen optio, jolloin nimi voi sisältää myös ASCII-merkit 192-223 (paitsi 215) ja ASCII-merkit 224-255 (paitsi 247). Optiota käytettäessä ensimmäinen merkki voi olla kirjain, numero tai alaviiva ja alaviivoja voi nimessä olla useampi peräkkäin. Nimeämissäännöt ovat samanlaiset kaikille projektissa käytettäville nimille. Lisäksi parametreille on määritettävä käytettävä tietotyyppi, minkä lisäksi niille voidaan määritellä muita elementtejä, kuten mm. kommentti (1024 merkkiä) ja alkuarvo.

Toimilohkon varsinaisen ohjelmakoodi voidaan kirjoittaa käyttämällä seuraavia kieliä: LD, IL, ST tai FBD. Toimilohko voi sisältää joko usean koodiosion, tai IEC-optio asetettuna ainoastaan yhden osion. Osiolle voidaan antaa nimi (enintään 32 merkkiä) ja kommentti (enintään 256 merkkiä). Lisäksi osion suorittamiselle voidaan asettaa ehto ja osio voidaan luku/kirjoitussuojata.

Kun toimilohkon ohjelmakoodi on saatu valmiiksi, se on valmis käytettäväksi. Tämä tapahtuu luomalla toimilohkosta uusia ilmentymiä, jolloin itse ohjelmakoodi ei kopioidu, vaan ainoastaan toimilohkoon liittyvä data. Uuden ilmentymän luominen käy yksinkertaisesti esimerkiksi raahaamalla toimilohko haluttuun paikkaan ohjelmakoodissa ja yhdistämällä tarvittavat toimilohkon portit. Ilmentymillä on myös oltava yksiselitteiset nimet (enintään 32 merkkiä, ensimmäinen merkki kirjain), joilla ne voidaan erottaa toisistaan. Toimilohkosta voi luoda niin monta ilmentymää kuin haluaa, ainut rajoitus on käytettävissä oleva muistin määrä.

Jos käyttäjä on luonut projektin aikana useita hyödyllisiä toimilohkoja, ne voidaan tallentaa käyttäjäkirjastoon. Sieltä ne ovat käytettävissä uusissa projekteissa aivan kuten valmiit kirjastofunktiot. Käyttäjä voi jakaa omat kirjastonsa haluamiinsa kokonaisuuksiin esimerkiksi toiminnallisuuden perusteella.

Projektin sisältämät toimilohkot on myös mahdollista tuoda ulos export-toiminnolla. Tämän jälkeen toimilohkot voidaan ottaa käyttöön vaikka toisessa työasemassa ja toisessa projektissa import-toiminnolla. Export-toiminto tuottaa tavallisia XML-tiedostoja, joita on siis mahdollista lukea ja käsitellä myös ilman työasemaan asennettua Unity Pro ohjelmistoa. Toimilohkot on myös mahdollista suojata siten, ettei niitä saa tuotua ulos projektista. Tällä tavoin voidaan estää ohjelmakoodin joutuminen vääriin käsiin.

Myös kokonainen kirjasto voidaan siirtää työasemasta toiseen. Jotta siirto olisi mahdollista, on kirjastosta luotava asennettava versio. Tuloksena syntyy tiedosto, joka voidaan puolestaan asentaa ja ottaa käyttöön toisessa työasemassa. Kirjastotyökalu sisältää myös versionhallinnan, jolla voidaan mm. verrata projektin toimilohkoja kirjastossa oleviin.

Projektikohtaisissa asetuksissa voidaan myös määrittää kirjastojen sijainti. Oletuksena käytössä on Unity Pro:n asennushakemisto, mutta se voidaan muuttaa esimerkiksi verkkolevyllä sijaitsevaan hakemistoon. Tällöin on kuitenkin tärkeä muistaa, ettei verkkolevylle ole välttämättä pääsyä, ja sitä kautta kirjastoihinkaan, kun ollaan esimerkiksi kohteessa tekemässä muutosta projektiin.

6.2 Siemens Simatic Step 7 -kehitysympäristö

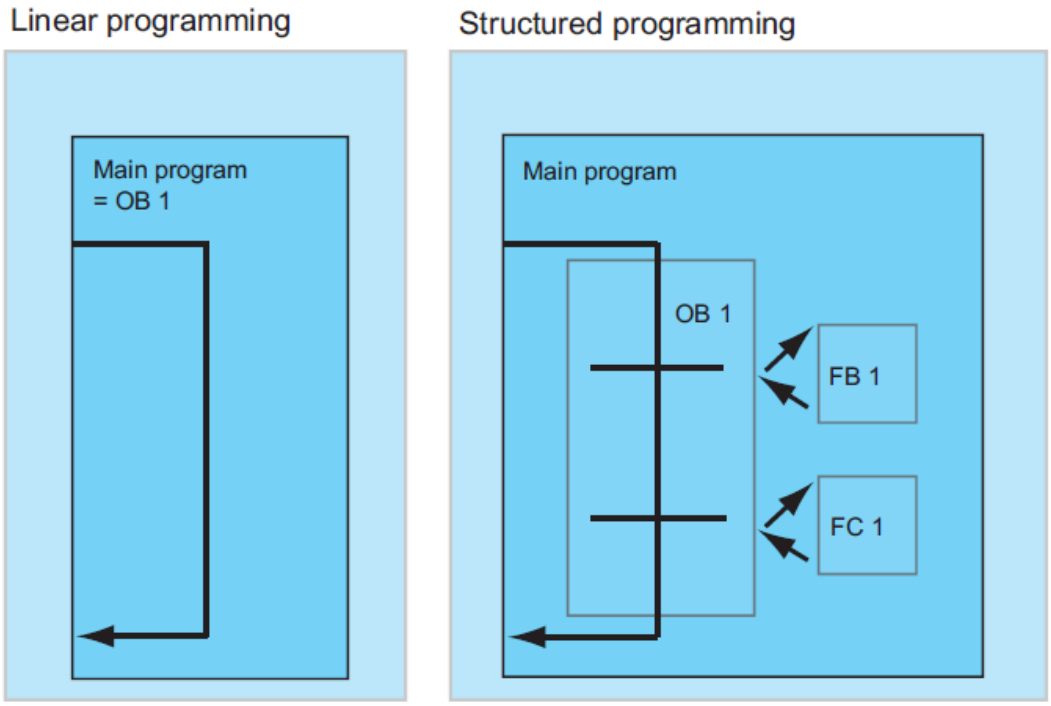
Simatic Step 7 on saksalaisen Siemens yhtiön ohjelmoitaville logiikoille tarkoitettu kehitysympäristö. Step 7 tukee seuraavia Siemensin ohjelmoitavia logiikoita: Simatic S7-200, Simatic S7-300/400 ja Simatic M7-300/400. Step 7 tarjoaa myöskin välineet ohjelmien testaukseen, diagnosointiin ja virheiden jäljitykseen.

Step 7 asennuksen järjestelmävaatimukset ovat seuraavat: vähintään Pentium-prosessori

600 MHz:in kellotaajuudella ja 512 megatavua muistia (suositus 1 gigatavu). Kiintolevyllä tulisi olla noin 650-900 megatavua vapaa tilaa. Tuettuja käyttöjärjestelmiä ovat Microsoft Windows 2000, Microsoft Windows XP Professional Edition sekä Microsoft Windows 2003 Server. Muina vaatimuksina mainittakoon Microsoft Internet Explorer 6.0 tai uudempi.

Step 7:ssä voidaan ohjelmien tekemiseen käyttää standardin IEC-61131-3 mukaisia kieliä (FBD, LD, ST). Muut ohjelmointikielet saa käyttöön hankkimalla lisäosan ja kyseisen lisenssin. Sovellusohjelma koostuu organisointilohkoista (Organization Block, OB). Sovelluksessa on oltava vähintään yksi organisointilohko, joka toimii pääohjelman tyyliin (OB1). Siihen voidaan tarvittaessa liittää muita organisointilohkoja, joilla on alempi suoritusprioriteetti.

Koko sovellusohjelma voidaan kirjoittaa joko lineaarisesti yhteen lohkoon tai strukturoidusti, jolloin sovellus jaetaan useaan lohkoon (Kuva 23). Sovellusohjelman jakaminen lohkoihin ja toiminnallisiin kokonaisuuksiin selkeyttää varsinkin isoja ja monimutkaisia sovelluksia.



Kuva 23: Ohjelman jako lohkoihin [Siemens, 2006a]

Ehkä tavallisin tapa ajaa sovellusta on ns. syklinen suoritus, jossa pelkästään organisointilohko OB1 sisältämää ohjelmaa suoritetaan toistuvasti (Kuva 23). On myös mahdollista suorittaa ohjelmaa tapahtuma pohjaisesti, jolloin pääohjelman suoritus katkeaa keskeytyksen tapahtuessa (Interrupt).

Step 7 tarjoaa lähes tulkoon samat tietotyypit käytettäväksi ohjelmoinnissa kuin Unity Pro. Tuettuja tietotyyppejä ovat mm. BOOL (TRUE/FALSE), 16- ja 32-

Taulukko 3: Lohkot Siemens Step 7:ssä.

Lohko	Kuvaus
Organization block	Määrittelee ohjelman rakenteen
System function blocks and system functions	Integroitu S7-keskusyksikköön, niiden avulla päästään käsiksi järjestelmään
Function blocks	Käyttäjän toteuttamia, oma ”muisti”
Functions	Käyttäjän toteuttamia
Instance data blocks	Järjestelmä luo nämä käännettäessä
Data blocks	Sisältävät toimilohkojen datan

bittiset bittijonot, 16- ja 32-bittiset kokonaisluvut etumerkillä sekä 32-bittiset liukuluvut. Ajan ja päivämäärän käsittelyyn voidaan käyttää mm. IEC:n mukaisia tietotyyppejä (TIME ja DATE) tai Simatic:in omaa S5TIME-tietotyyppiä. Tietotyyppi CHAR mahdollistaa ASCII-merkkien käsittelyn. Lisäksi on mahdollista luoda ja käyttää kompleksisia tietotyyppejä, kuten mm. taulukoita (ARRAY), struktuureja (STRUCT) ja merkkijonoja (STRING).

Step 7:ssä on mukana seuraavat vakiokirjastot, jotka sisältävät usein tarvittavia toimilohkoja ja funktioita:

- systeemikirjasto - systeemifunktiot -toimilohkot
- S5-S7 muunnoslohkot
- IEC toimilohkot
- Organisointilohkot
- PID säätölohkot
- tietoliikennetoimilohkot
- TI-S7 muunnoslohkot
- sekalaiset lohkot

Myös Step 7:ssä on mahdollista luoda omia toimilohkoja ja tallentaa ne omiin kirjastoihinsa. Oman toimilohkon käyttäminen pitää sisällään kolme vaihetta:

1. luodaan toimilohkon malli
2. luodaan toimilohkoon liittyvä datalohko
3. kutsutaan toimilohkoa ja sitä vastaavaa datalohkoa sovelluksessa

Toimilohkon luonti sisältää kolme vaihetta: muuttujien määrittely, ohjelmakoodin luominen ja toimilohkon ominaisuuksien määrittely. Toimilohkon, muuttujan tai symbolin nimi voi olla maksimissaan 24 merkkiä pitkä. Kaikille muuttujille on myös annettava tietotyyppi. Lisäksi nimen on oltava yksiselitteinen. Symbolin nimeäminen riippuu siitä, ovatko ne globaaleja (Shared) vai toimilohkon sisäisiä, paikallisia (Local), symboleja. Globaalin symbolin nimi voi sisältää kirjaimia, numeroita ja erikoismerkkejä (ASCII merkit 0x00 ja 0xFF eivätkä lainausmerkit ole sallittuja). Jos nimi sisältää erikoismerkkejä, nimi pitää laittaa lainausmerkkien sisään. Paikallisen symbolin nimessä sallittuja merkkejä ovat kirjaimet, numerot ja alaviivat. Nimessä ei kuitenkaan voi olla kahta alaviivaa peräkkäin. Versiosta 4.02 lähtien Step 7 ei tee eroa isojen ja pienien kirjainten välillä, joten ”Moottori1”ja ”moottori1”ovat sama symboli. Tämä tärkeää ottaa huomioon työskenneltäessä vanhempien projektien parissa, jolloin symbolinimet eivät välttämättä ole yksiselitteisiä.

Seuraavaksi luodaan toimilohkon ohjelmakoodi. Ohjelmoimiseen voidaan käyttää oletuksena LAD-, FBD- ja STL-kieliä. Siemens suosittelee seuraavaa mallia ohjelman kirjoittamiseen (Kuva 24)

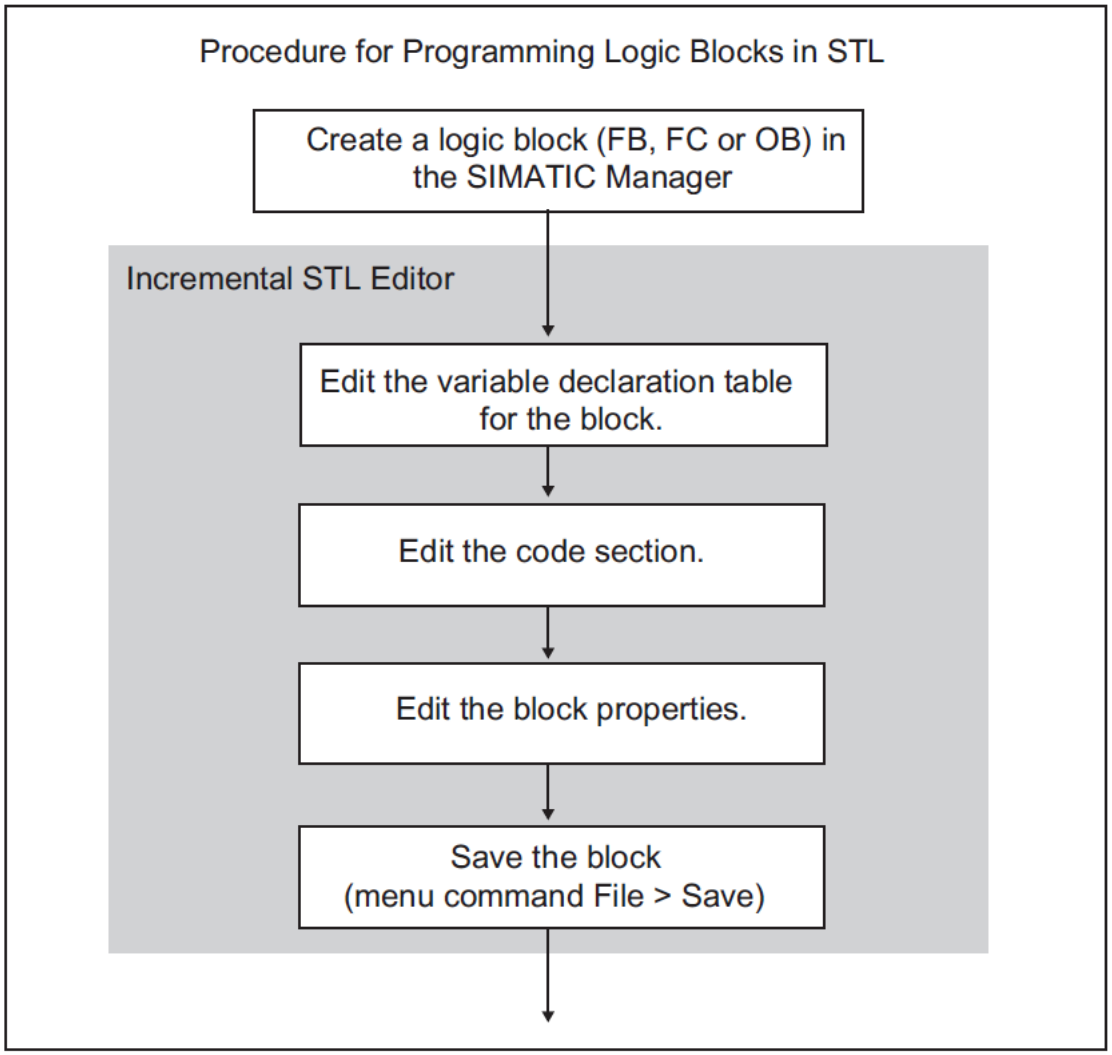
Lohkoon on mahdollista liittää erilaisia kommentteja ja otsikoita. Lohkon tai ohjelmaverkon otsikon maksimi pituus on 64 merkkiä. Lohkolle voidaan kirjoittaa kommentti, jossa voidaan dokumentoida lohkon toimintaa. Samoin yksittäiselle ohjelmaverkolle voidaan kirjoittaa vastaava kommentti. Yhdessä lohkossa on varattuna 64 kilotavua tilaa näille kommenteille. Lisäksi lohkolle voidaan määrittää mm. luku- ja kirjoitussuojaus, versionumero ja tekijä.

Jotta toimilohkoa voitaisiin käyttää sovelluksessa, on sille luotava datalohko. Jokaiselle kutsukerralle on luotava oma datalohko. Datalohkon tehtävänä on linkittää sovelluksen symbolit ja osoitteet toimilohkoon.

Kun tarvittavat datalohkot on luotu, voidaan toimilohkoa kutsua sovelluksessa. Ensiksi asetetaan toimilohko haluttuun kohtaan (Insert -> Function Block). Tämän jälkeen toimilohkoon liitetään haluttu datalohko (Insert -> Data Block). Myös Step 7:ssä toimilohkojen ja datalohkojen lukumäärää rajoittaa ainoastaan käytettävissä olevan muistin määrä. Käyttäjällä on mahdollisuus luoda omia kirjastoja, joihin hän voi tallentaa projekteissa luomiaan toimilohkoja. Uuden kirjaston luominen tapahtuu samaan tyyliin luin uuden projektin (File -> New -> Library). Kirjaston nimi voi olla pidempi kuin kahdeksan merkkiä, mutta jokaisen kirjaston nimi tulee olla ainutlaatuinen noiden kahdeksan ensimmäisen merkin osalta. Kirjastoja voidaan siirtää työasemasta toiseen aivan kuten projektejakin. Toimilohkoja ei kuitenkaan saa tuotua ulos yksittäin, kuten Unity Pro:ssa.

6.3 Kirjastojen ominaisuudet

Kehitysympäristöt asettavat tiettyjä vaatimuksia kirjastolle ja sen sisältämille moduuleille. Kirjaston moduulien ja lohkojen nimien täytyy olla yksiselitteiset. Lohkon nimi ei esimerkiksi voi olla sama kuin jonkun vakiolohkon nimi eikä se voi olla muu ns. varattu sana. Lisäksi kehitysympäristö rajoittaa nimessä käytettäviä merkkejä



Kuva 24: Toimilohkon ohjelmointi Step 7:llä [Siemens, 2006a]

ja nimen pituuden, kuten aiemmin mainittiin. Kirjaston käytettävyyden kannalta olisi toivottavaa, että kirjaston moduulit ja niiden rajapinnat nimettäisiin kuvaavasti ja että nimeämisessä käytettäisiin yhtenäistä käytäntöä. Lisäksi on sovittava ketkä ylläpitävät kirjastoa (moduulien lisäys ja poisto), käytettävä versionhallinta ja kirjaston dokumentointi.

Kehitysympäristöt mahdollistavat kirjaston sijoittamisen lähiverkon palvelimelle, jolloin kaikilla käyttäjillä on käytössään sama versio kirjastosta. Palvelimelta jaettu kirjasto muodostuu ongelmaksi, jos palvelimelle ei jostain syystä ole pääsyä. Tällainen tilanne voi syntyä, kun asiakkaan luona käyttöönotossa ohjelmassa havaitaan virhe, joka vaatii muutosta ohjelmaan.

7 Tulokset

Automaatiojärjestelmä pitää tavallisesti sisällään yhden tai useamman sovelluksen, joka suoritetaan logiikassa (tai useammassa logiikassa). Ensimmäinen tehtävä sovellusta luotaessa on määrittää mitä sen halutaan tekevän. Tämä käy ilmi järjestelmän toiminnallisesta kuvauksesta ja käyttäjävaatimuksista, jotka tehdään joko järjestelmän toimittajan tai erillisen suunnittelutoimiston toimesta. [SAS, 2005]

Seuraavana vuorossa on suunnitteluvaihe, jossa määritellään mm. itse ohjelman rakenne ja käytettävät laitteistot. Ohjelman rakennetta suunniteltaessa on tärkeää tunnistaa usein toistuvat toiminnot, ja suunnitella ne toteutettavaksi parametrisoitavilla moduuleilla. Tällöin moduuleja on helppoa ja tehokasta käyttää toistuvasti, myös tulevissa projekteissa. Jos aikaisemmissa projekteissa on jo saatu luotua moduuleja, niitä kannattaa käyttää hyväksi. [SAS, 2005]

Kun järjestelmä on suunniteltu, on toteutuksen vuoro. Suunnitellut moduulit ohjelmoidaan ja niistä kootaan kokonainen sovellus. Logiikoita ohjelmoitaessa, olennainen vaihe on järjestelmän konfigurointi, jossa kehitysympäristössä määritellään järjestelmän koostumus ja miten eri osat liittyvät toisiinsa. [SAS, 2005]

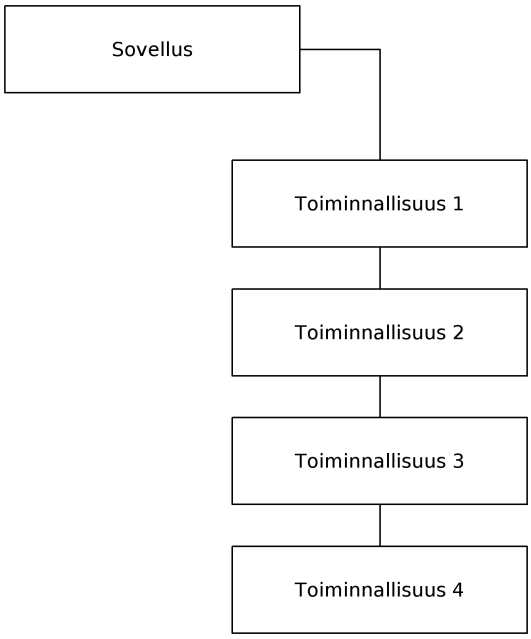
7.1 Sovelluksen rakenne

Ohjelmistosuunnittelun tavoitteena on määritellä mm. sovelluksen rakenne ja rajapinnat. Yksinkertainen automaatiosovellus voidaan toteuttaa yhdellä suhteellisen yksinkertaisella ohjelmalla. Vähänkään isommat ja monimutkaiset sovellukset kannattaa jakaa osiin esimerkiksi toiminnallisuuden perusteella. [Siemens, 2006a] Lisäksi näiden osioiden toteutuksessa on hyödyllistä käyttää hyväksi uudelleen käytettäviä komponentteja [SAS, 2005]. Useimmat nykyaikaiset ohjelmoitavien logiikoiden kehitysympäristöt, kuten esimerkiksi Unity Pro ja Step 7, tukevat vähintään jollain tapaa näitä menetelmiä.

Koko ohjelman toteuttaminen yhtenä suurena kokonaisuutena on vaikeaa. Ohjelma on saatava täysin valmiiksi, ennen kuin sitä voidaan testata. Testattavana ovat kerralla ainakin lähes kaikki ohjelman toiminnot ja virheen paikallistamiseksi on pahimmassa tapauksessa käytävä läpi koko ohjelmakoodi. Dokumentointi voi olla vaikeaa, koska ison ohjelman toiminnan hahmottaminen on hankalaa. Lisäksi kaikkien toimintojen toteutusten yksityiskohtia ei välttämättä muisteta, varsinkaan jos ohjelmoidessa ei ole tehty muistiinpanoja ja dokumentointi aloitetaan vasta lopussa. [Tian, 2005]

Ohjelman suunnittelu ja toteutus helpottuu huomattavasti, jos ohjelma on jaettu pienempiin kokonaisuuksiin [SAS, 2005]. Ohjelman rakenne saadaan selkeämmäksi ja sen toiminnan hahmottuminen helpottuu (Kuva 25).

Pienemmät kokonaisuudet ovat helpompia ja nopeampia tehdä toteutusvaiheessa, varsinkin kun osat voidaan jakaa useamman ohjelmoijan tehtäväksi. Samoin testaus helpottuu ja nopeutuu, koska osiota voidaan testata yksittäin vaikkei muut



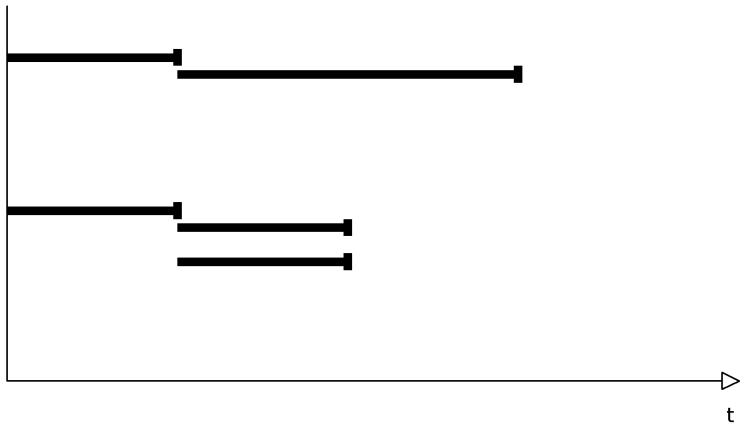
Kuva 25: Strukturoitu ohjelma

osiot olisi vielä valmiina ja käytettävissä. Virheiden löytäminen on helpompaa, koska testattavia toimintoja on vähemmän ja läpikäytäviä koodirivejä on vähemmän. Dokumentointi on myöskin helpompaa, koska osioihin jaettu ohjelma on helpompi hahmottaa. On myös todennäköisempää että dokumentointi tulee hoidettua saman tien, koska kerralla ei tarvitse dokumentoida koko ohjelmaa.

Osioihin jaettuun ohjelmaan on myös huomattavasti helpompi tehdä muutoksia myöhemmin. Jos tietty toiminto vaatii muutosta, löydetään toiminnon sisältämä osio helposti. Jos rajapintoihin ei tule muutoksi, riittää että muutetaan ainoastaan kyseistä osiota eikä muihin osioihin tarvitse koskea.

Kuvassa 26 on kuvattu jaottelun vaikutusta ohjelmoimiseen kuluvaan aikaan. Kuvan esimerkistä nähdään kuinka jakamalla ohjelma kahteen osaan ja käyttämällä kahta ohjelmoijaa rinnakkain voidaan ohjelma saada valmiiksi puolta nopeammin verrattuna siihen jos ohjelma olisi yksi iso kokonaisuus ja käytettävissä olisi yksi ohjelmoija. Jakamalla ohjelma kokonaisuuksiin ja käyttämällä strukturoitua ohjelmointia saadaan kahdesta tai useammasta ohjelmoijasta enemmän hyötyä, koska eri osiot voidaan jakaa ohjelmoijien kesken ja he voivat työstää niitä samaan aikaan rinnakkain. Näin muodostuu ikään kuin liukuhihna, joka tuottaa ohjelmakoodia. Projektin johdon on myös helpompi kohdistaa resursseja ja tasata yksittäiseen ohjelmoijaan kohdistuvaa kuormaa, kun ohjelmoitavat kokonaisuudet ovat pienempiä. Tätä kautta myös projektin aikataulun hallinta helpottuu. [SAS, 2005]

Esimerkiksi vesihuollon automaatiossa laitoskohteet ovat monesti niin isoja ja laajoja ja sisältävät useita erilaisia toiminnallisuuksia, että sovelluksen jakaminen osiin toiminnallisuuden mukaan on perusteltua. Pienemmät kohteet, jotka eivät sisällä monipuolisia toiminnallisuuksia, kuten pumppaamot, on mahdollista toteuttaa il-



Kuva 26: Ohjelman paloittelun vaikutus aikatauluun

man sovelluksen jakamista.

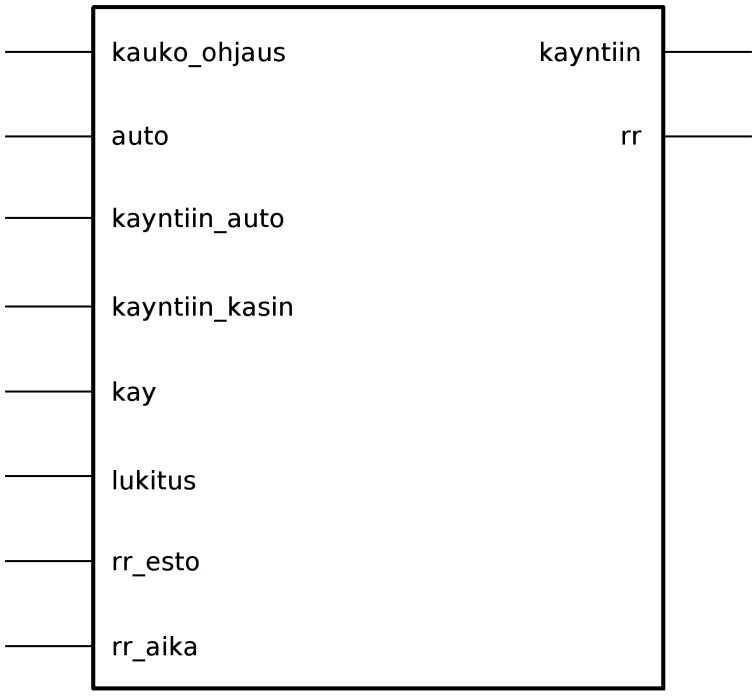
Ohjelmistoa suunniteltaessa on hyödyllistä tunnistaa toiminnallisuudet, jotka toistuvat usein. Nämä voidaan toteuttaa moduuleina eli toimilohkoina [SAS, 2005]. Toimilohko voidaan toteuttaa, eli ohjelmoida ja testata, täysin itsenäisesti riippumatta muusta ohjelmasta. Toimilohko ohjelmoidaan kerran jonka jälkeen sitä voidaan kutsua useita kertoja erilaisilla parametreilla tarpeen mukaan. Näin voidaan merkittävästi vähentää ohjelmoimiseen ja testaukseen kuluvaa aikaa [John ja Tiegelkamp, 2001]. Valmiit ohjelmalohkot kannattaa koota kirjastoihin, joista ne ovat käytettävissä myös tulevissa projekteissa. Ajan kuluessa kirjasto kasvaa ja ohjelmointi tehostuu merkittävästi, koska valmista, testattua ohjelmakoodia on helposti saatavana kirjastosta [John ja Tiegelkamp, 2001]. Arkistoidulla ohjelmakoodilla voidaan lisäksi auttaa uusia ohjelmoijia, joilla ei vielä ole laajaa kokemusta kyseisestä kehitysympäristöstä.

Aiemmin todettiin että toimilohkolla kannattaa toteuttaa usein toistuvia toiminnallisuuksia. Vesihuollossa tällaisia toiminnallisuuksia ovat erilaisten venttiilien, pumppujen, moottorien ja puhaltimien ohjaukset. Nykyisin sovellukset sisältävät myös kasvavissa määrin taajuusmuuttajan ohjauksia. Esimerkkinä pumpun ohjauksen toimilohko, joka sisältää sovelluksen pumppuihin liittyvän toiminnallisuuden, on esitetty kuvassa 27.

7.2 Ohjelman toteutus

Kun ohjelmistosuunnittelu on valmis ja sovelluksen rakenne ja toiminta on selvillä voidaan aloittaa sovelluksen toteuttaminen. Ohjelmoitavan logiikan tapauksessa toteutus koostuu laitteiston konfiguroinnista ja itse ohjelmoinnista. Usein käyttäjä voi valita kumman tekee ensin, konfiguroinnin vai ohjelmoinnin (Kuva 28). Monimutkaisia ohjelmia luotaessa kannattaa kenties suorittaa konfigurointi ennen ohjelmointia, koska näin osoitteet ovat helpommin hallittavissa [Siemens, 2006b].

Laitteiston konfiguroinnissa määritellään laitteiston koostumus, kuten minkä tyyp-

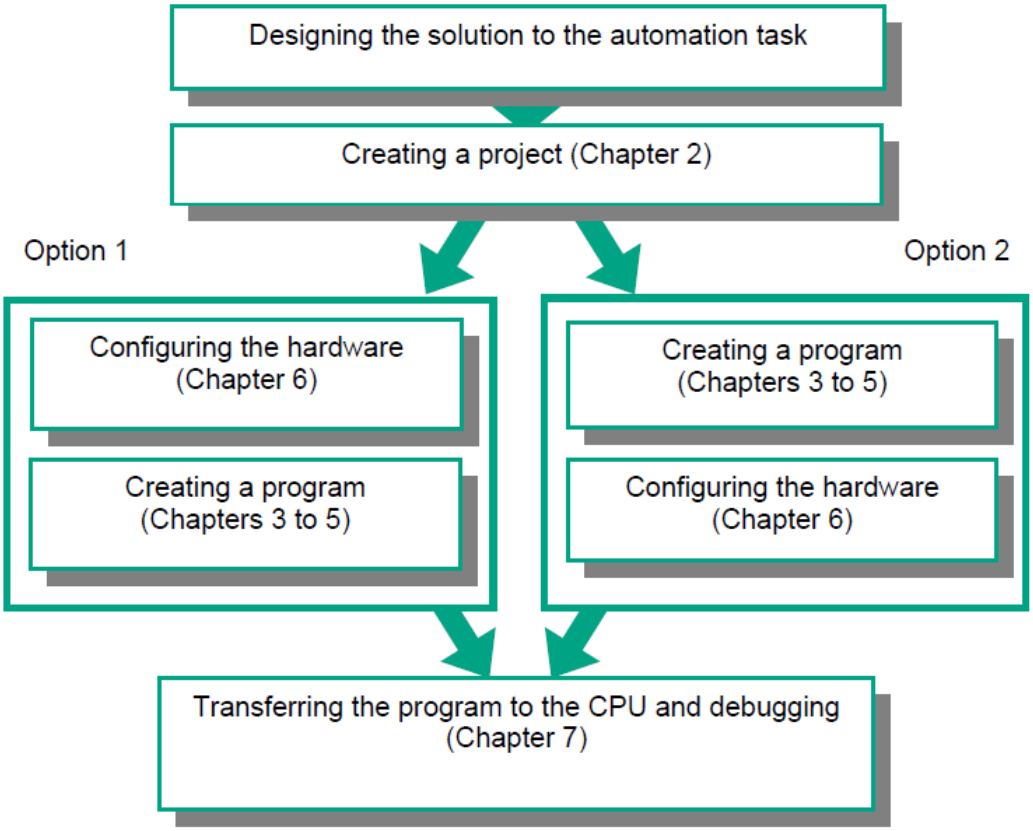


Kuva 27: Pumpun ohjauslohko

pinen on järjestelmän keskusyksikkö ja siihen mahdollisesti liittyvät lisäkortit. Konfiguroinnissa laitteiston fyysiset sisään- ja ulostulot saavat muistiosoitteen ja näin niitä voidaan käyttää ohjelmassa. Muistiosoitteen tyyppi riippuu mm. siitä onko kyseessä sisään- vai ulostulo ja osoitteen leveydestä (esimerkiksi 1-, 16- tai 32-bittiä). Standardin IEC 61131 mukaisissa ympäristöissä muistipaikalle määritetään myös tietotyyppi [IEC, 2001]. Seuraavassa on esimerkkejä muistiosoitteista: Q4.0, I1.1, M2.0 ja FB10.

Jos sovellus pitää sisällään suuren joukon muistiosoitteita, ohjelmoija saattaa mennä sekaisin niiden käytössä. Esimerkiksi voi olla vaikea muistaa minkä venttiilin kiinnitieto löytyy osoitteesta I2.7, jos venttiilejä on kymmeniä tai peräti satoja. Toinen huomioitava seikka on, että muistiosoitteiden kanssa on helppo tehdä virhe, mutta virheen paikallistaminen voi olla erittäin vaikeaa.

Nykyään lähes kaikista uusimmista kehitysympäristöistä löytyy mahdollisuus käyttää ohjelmoinnissa symboleja absoluuttisten muistiosoitteiden sijaan [John ja Tiegelkamp, 2001]. Myös Step 7 ja Unity Pro sisältävät tämän ominaisuuden [Siemens 2006a][Schneider Electric, 2006]. Symboleja käytettäessä muistiosoitteeseen liitetään yksilöllinen, selväkielinen nimi. Esimerkiksi osoitteelle I1.7 annetaan nimi 'moottori1_on', eli kyseessä on ykkösmoottorin käyntitieto. Symboleja käyttämällä ohjelmointi helpottuu huomattavasti, koska ohjelmoidessa ei tarvitse muistaa muistiosoitteita, vaan voidaan käyttää helpommin muistettavia nimiä. Nimien valintaan kannattaa kiinnittää huomiota, aivan kuten nimettäessä muuttujia missä tahansa ohjelmointikielessä, kuten C:ssä ja Java:ssa. Esimerkiksi jos sovelluksessa käsitellään kahta painonappia, ne voidaan hyvin nimetä 'nappi1' ja 'nappi2' ilman että ohjel-



Kuva 28: Sovelluksen konfigurointi [Siemens, 2006b]

moija sekoittaa napit keskenään. Tilanne on toinen jos sovelluksessa on kymmeniä painonappeja ja ne nimetään vain juoksevalla numerolla. Tällöin voi olla hankala muistaa mikä toiminto liittyi mihinkin painonappiin. Ohjelmointiympäristöt asettavat omat rajoituksensa mm. symbolinimien pituudelle ja sallituille merkeille.

Lopullinen sovellus saadaan aikaan luomalla tarvittavat moduulit ja hyödyntämällä jo olemassa olevaa koodia. Jos ohjelmamoduulit on hyvin suunniteltu ja dokumentoitu, on niiden toteuttaminen helpompaa ja suoraviivaisempaa, koska mm. toiminnot ja rajapinnat ovat selvillä [SAS, 2005].

Automaation sovelluskehitys on monesti projekteissa yhden henkilön vastuulla, eli yksi henkilö on hoitanut niin suunnittelun, ohjelmoinnin kuin testauksen. Sovelluksen suunnittelu on voinut olla puutteellista tai se on puuttunut kokonaan. Esimerkiksi projektissa on ryhdytty suoraan ohjelmoimaan ja ohjelmoija on suunnitellut sovelluksen päässään. Lopuksi sovellus on testattu ja dokumentoitu, jos on ehditty. Kokenut ohjelmoija voi tosin näinkin saada aikaan toimivan sovelluksen.

Tällaiseen toteutukseen liittyy kuitenkin ongelmia. Jos sovellusta ja sen moduuleja ei ole suunniteltu kunnolla, on riski että jokin toiminnallisuus jää toteuttamatta tai se on puutteellinen, eli sovellus toimii väärin. Jos sovellusta ei ole jaettu osiin ja dokumentointi on puutteellista, ei sovelluksen toteutusta voida helposti jakaa useal-

le ohjelmoijalle. Samoin toisen ohjelmoijan on vaikea jatkaa sovelluksen toteutusta tarpeen vaatiessa. Puutteellisessa suunnittelussa ei välttämättä osata hyödyntää valmiita ohjelmakirjastoja, vaan jo olemassa olevia toiminnallisuuksia ohjelmoidaan ja testataan uudelleen ja uudelleen projektista toiseen. Jos sovittuja ohjelmointikäytäntöjä ei ole, tulee sovelluksista helposti liikaa tekijänsä näköisiä. Esimerkiksi eri ohjelmoijat saattavat luoda samanlaisen toiminnallisuuden eri projektien yhteydessä, vaikka yksi, kirjastoon arkistoitu toiminnallisuus riittäisi, tai toisen tekemää toiminnallisuutta ei voi hyödyntää, koska rajapinnat ovat erilaiset. [John ja Tiegelkamp, 2001]

Myös sovelluksen testaus hankaloituu ja sitä on vaikea priorisoida, jos testauksen suunnittelu ja dokumentointi on puutteellista. Sovelluksen testaus on hankalaa, koska käytännössä ohjelmoijan on itse testattava sovellus tai oltava läheisesti mukana siinä, eikä ulkopuolinen pysty helposti ja tehokkaasti testaamaan sovellusta. Omaa virhettään ei välttämättä huomaa helposti, kun taas ulkopuolinen testaaja voi löytää saman virheen nopeastikin.

Ohjelmoijan on lähes pakko olla itse mukana järjestelmän käyttöönotossa, koska puutteellisten dokumenttien vuoksi muiden on hankala tuntea järjestelmän toimintaa riittävällä tarkkuudella. Luonnollisesti ohjelmoija tuntee aina sovelluksensa parhaiten, ja sitä kautta hänen mukana olonsa käyttöönoton yhteydessä on perusteltua. Tämä ei kuitenkaan välttämättä aina ole mahdollista, esimerkiksi ohjelmoijan sairastuessa, jolloin muiden voi olla vaikeaa tuntea sovellusta riittävällä tarkkuudella, varsinkin jos sovellus on puutteellisesti dokumentoitu.

Vaikka automaatioprojekteissa käytettäisiinkin hyväksi ohjelmakirjastoja, ei niistä saada irti maksimaalista hyötyä jos kirjastoja ei ole dokumentoitu. Uusi ohjelmoija ei saa toisen tekemän moduulin toiminnasta välttämättä helpolla selkoa, ja koko kirjasto saattaa näin jäädä hyödyntämättä puutteellisesta dokumentoinnista johtuen. Kirjastosta ei ole hyötyä, jos ei tiedetä mitä se pitää sisällään ja jos kirjastoon tutustuminen on vaivalloista. Kun sovellus on suunniteltu hyvin, tiedetään heti mitkä toiminnallisuudet löytyvät valmiina moduuleina kirjastoista ja mitkä uudet moduulit ja toiminnallisuudet joudutaan luomaan. [John ja Tiegelkamp, 2001]

7.3 Ohjelmoinnissa käytettävät nimet

Jotta automaatiosovellus olisi helpommin luettavissa, voidaan käyttää strukturoitua ohjelmointia ja toimilohkoja, sekä symbolinimiä muistiosoitteiden sijaan. Valitsemalla nimet sopivasti voidaan edelleen helpottaa sovelluskehitystä. Käytyäni läpi useita vanhoja automaatioprojekteja ja keskusteltuani niiden toteuttamiseen osallistuneiden henkilöiden kanssa, kävi ilmi että sovelluksessa käytettävät symbolinimet oli valittu hieman sattuman varaisesti, ohjelmoijasta ja projektista riippuen. Esimerkiksi ohjelmoijat saattoivat käyttää aina samantyylisiä nimiä omissa projekteissaan, mutta sama nimi saattoi tarkoittaa eri ohjelmoijien tekemissä ohjelmissa eri asioita. Nimien valinnasta puuttui siis yhtenäinen käytäntö. Tästä puolestaan seurasi, että ohjelmoijien oli hankalaa ja hidasta työskennellä toisen luoman sovelluksen parissa.

Ratkaisuksi sekavaan nimeämiskäytäntöön päätettiin kehittää yhtenäinen käytäntö, jolla sovelluksen symbolinimet valitaan. Nimien tulisi olla riittävän kuvaavia, myös silloin kun käytettävissä on rajoitetusti merkkejä. Lisäksi nimien tulisi olla käytettävissä niin pienissä kohteissa, kuten esimerkiksi paineenkorotusasemalla, kuin myös isoissa kohteissa, kuten esimerkiksi jätevedenpuhdistamoilla.

Jotta nimistä saataisiin sekä mahdollisimman luonnollisia että yksiselitteisiä, valittiin tässä työssä nimen rakenteen perustaksi PI-kaavion tunnukset. PI-kaavion tunnukset ovat valmiina olemassa ja käytettävissä kun automaatiosovellusta ryhdytään luomaan. Lisäksi tunnukset ovat valmiiksi yksiselitteisiä.

Tässä työssä luodussa nimeämiskäytännössä symbolinimen rakenne on kolmiosainen ja alaviiva toimii erottimena:

AAA_BBB_CCC

Ensimmäinen osa (AAA) on laitos-/ala-asematunnus. Se kertoo mihin laitoksen logiikkaan tai ala-asemaan nimi liittyy. Laitoksissa voi olla useita logiikoita, ja näin eri logiikoiden muuttujat pystytään erottamaan toisistaan. Jos esimerkiksi laitoksessa on kolme ohjelmoitavaa logiikkaa, niitä vastaavat laitostunnukset voisivat olla 'PLC1' 'PLC2' ja PLC3'. Samoin näin voidaan erottaa toisistaan eri ala-asemat, kuten esimerkiksi pumppaamot.

Seuraavana nimessä on laitetunnus (BBB). Laitetunnus saadaan PI-kaaviosta ja sen avulla voidaan erottaa mm. eri pumput, moottorit, venttiilit ja säätimet toisistaan. Tunnusta ei tarvitse näin miettiä, vaan se saadaan suoraan kopioimalla PI-kaaviosta. Esimerkiksi pumpun 1 laitetunnus voisi olla 'p1', pumpun 2 'p2' jne. Säätimet puolestaan voidaan erottaa tunnuksella Xic, missä X kertoo säädettävän suureen.

Viimeisenä nimessä on detaljitunnus, joka kertoo yksityiskohtaisesti mistä tiedosta on kyse. Näin saadaan erotettua saman laitteeseen liittyvät tiedot, kuten mittaus ja siihen usein liittyvät erilaiset raja-arvot, kuten hälytysrajat. Detaljitunnus voidaan valita melko vapaasti, tärkeintä on, että samaa tietoa kuvataan aina samalla nimellä ja ettei esimerkiksi laitteen ohjaustieto ja käyntitieto mene keskenään sekaisin. Esimerkiksi valitaan tunnus 'ON' laitteen käyntitiedolle ja 'SETON' käyntiin ohjaukselle. Valvomoon siirrettävien binääristen tietojen kohdalla on myös tärkeää sopia yhtenäinen toimisuunta. Eli esimerkiksi kaikki laitteet ovat päällä, jos niiden tila on 1 ja pois päältä, jos tila on 0. Samoin voidaan päättää että kaikki hälytykset ovat voimassa kun niiden tila on 1. Näin tietojen merkitystä ei tarvitse arvailla valvomon sovelluksia tehtäessä.

Valitsemalla ohjelmoinnissa käytettävät symbolinimet edellä kuvatun mallin perusteella saavutetaan monia etuja. Nimistä saadaan helposti yksilöllisiä, koska nimet perustuvat PI-kaavion tunnuksiin, jotka ovat valmiiksi yksilöllisiä. Samalla nimistä tulee loogisia ja siten helpompia muistaa. PI-kaaviota seuraamalla ohjelmoija tietää heti mitkä laitteet liittyvät mihinkin toiminnallisuuteen ja mitkä nimet liittyvät näihin laitteisiin.

Ohjelmakoodia on helpompi lukea ja ohjelmoijien välisiltä väärinkäsityksiltä välty-

tään. Ohjelma voidaan helposti jakaa usean ohjelmoijan tehtäväksi ja ohjelmoija voi nyt helpommin jatkaa toisen aloittamaa työtä.

Ohjelman testaus helpottuu ja sen voi myös hyvin suorittaa joku muu kuin ohjelmoija itse, koska nyt muuttujien merkitys (nimet ja toimisuunnat) ovat selvät. Virheen havaitseminen ja löytäminen on helpompaa ja nopeampaa käytettäessä järjestelmällisiä nimiä.

Järjestelmällinen nimeämiskäytäntö ei helpota ainoastaan itse logiikkaohjelmointia, vaan koko automaatiojärjestelmän luomista. Myös valvomoohjelmistoissa muuttujille on usein mahdollista antaa nimi. Käyttämällä samaa (tai ainakin saman tyylistä) nimeämiskäytäntöä valvomon muuttujissa kuin logiikan symbolinimissä, voidaan valvomosovelluksen luominen aloittaa logiikkaohjelmoinnin rinnalla. Valvomosovelluksessa logiikan tiedot esitetään kuvissa halutuissa muodoissa, kuten numeroarvoina, kuvaajina tai muuttuvina väreinä. Valvomosovellusta luotaessa logiikalta saatavat tiedot on linkitettävä kuviin. Tietojen linkitys helpottuu huomattavasti kun käytetään samoja nimiä kuin symbolinimissä. Valvomonäyttöjen tekeminen voidaan aloittaa ennen kuin logiikkaohjelma on edes valmis, koska käytettävät nimet ovat selvillä. Jos usein toistuvalle toiminnallisuudelle on luotu ohjelmamoduuli, voidaan sille nyt luoda myös oma valvomonäyttö. Tarvittava tietojen linkitys kuvassa voidaan helposti tehdä projektikohtaisesti esimerkiksi 'etsi ja korvaa' toiminnolla, eikä koko kuvaa tarvitse luoda alusta alkaen. Tämä säästää runsaasti aikaa ja vaivaa esimerkiksi vesilaitoskohteissa, jotka sisältävät suuren määrän (lähes) samankaltaisia pumppaamoja.

Käyttöönoton aikaiset virheet löydetään myös helpommin, koska PI-kaavio sitoo nyt fyysiset laitteet ja automaatiojärjestelmän muuttujat toisiinsa ja toimisuunnat ovat yhtenäiset. Tyypillinen tilanne on esimerkiksi että ohjattaessa venttiili kiinni se aukeaa ja päinvastoin. Nyt on helposti paikallistettavissa onko ongelma esimerkiksi itse kenttälaitteessa, sen kaapeloinnissa, logiikkaohjelmassa vai valvomo-ohjelmassa. Isoissa laitoksissa on erittäin suuri määrä muuttujia (tuhansia), jolloin pelkän oikean muuttujan löytäminen listalta voi olla hankalaa. Järjestelmällinen nimeämiskäytäntö helpottaa oikean muuttujan löytämistä.

7.4 Kirjastot

Kuten jo aiemmin todettiin, olisi hyödyllistä jos kirjastoon saataisiin varastoitua usein käytettäviä ohjelmistomoduuleja ja toimilohkoja. Jotta nämä moduulit olisivat mahdollisimman yleiskäyttöisiä, tulee lisäksi kiinnittää huomiota muutamaan muuhun seikkaan.

Toimilohkojen rajapintojen, eli sisään- ja ulostulojen, tyyppeihin on syytä kiinnittää huomiota varsinkin jos käytössä on useampia eri valmistajien kehitysympäristöjä. Myös käytettävät sovellusarkkitehtuurit vaikuttavat osaltaan moduulien rajapintoihin.

Tässä työssä havaittiin, että eri ympäristöille on mahdollista kehittää lähes yhte-

näiset kirjastot. Tällöin kirjastot sisältävät samat toiminnallisuudet ja toimilohkot, mutta yksittäisen toimilohkon sisäisen toteutus saattaa olla hyvinkin erilainen. Tämä johtuu siitä, että eri ympäristöjen mukana tulee usein ainakin osittain erilaiset vakiokirjastot. Eli esimerkiksi toimilohkoa ei voida välttämättä toteuttaa identtisesti eri ympäristöissä, koska jotain perustoimilohkoa ei ole saatavana kuin ainoastaan tietyssä ympäristössä. Tämä voidaan kiertää kirjoittamalla puuttuva perustoimilohkokin itse. Vaikka vastaava perustoimilohko löytyisikin kaikista ympäristöistä, voivat toimilohkojen parametrien tietotyypit olla erilaiset eri kehitysympäristöissä Tämä voidaan puolestaan kiertää käyttämällä tietotyypin muuntavia funktioita.

Lisäksi tämän työn aikana huomattiin, että sovelluksen arkkitehtuuri ja siinä mahdollisesti käytettävät väylät ja muut tiedonsiirtotavat vaikuttavat myöskin toimilohkojen parametreihin. Sovelluksessa käytettävä kenttäväylä tai protokolla ei välttämättä tue kaikkia tietotyyppejä tai se vaikuttaa muuttujien määrään ja järjestykseen. Esimerkiksi varsin usein käytetty modbus-protokolla ei tue liukulukuja (floating point) ja tiedonsiirron optimoimiseksi liikennöitävien muuttujien tulisi sijaita perättäisissä muistipaikoissa. Tästä seuraa että modbus-protokollaa käytettäessä toimilohkojen parametrien valinnassa kannattaa pyrkiä välttämään liukulukuja. Vaikka käytettävä protokolla ei rajoittaisikaan tiedonsiirtoa, kannattaa silti valita toimilohkon parametrien tietotyypit sopivalla tavalla. Jos esimerkiksi toimilohkon sisääntuloparametri on valvomoon liikennöitävä asetusarvo kuten viive, ja tiedetään että se on aina suuruusluokaltaan joitain sekunteja tai minuutteja, voi olla parempi esittää viive kokonaislukuna IEC:n mukaisen vakioesitystavan sijaan. Tällöin valvomosovellusta luotaessa ei tarvitse käsitellä kompleksisia tietotyyppejä (date#time) ja tiedonsiirrossa voidaan käyttää modbus-protokollaa. Ajan ja päivämäärän esittäminen on muutenkin ongelmallista, koska osassa kehitysympäristöistä käytetään omaa tietotyyppiä ajalle (ja päivämäärälle) joka ei ole IEC-standardin mukainen, esimerkkinä Step 7:ssä ajalle käytettävä tietotyyppi S5TIME.

Kehitysympäristöjen eri versiot saattavat myös aiheuttaa ongelmia. Uusi versio saattaa sisältää niin merkittäviä muutoksia, ettei eri versiolla tehtyjä projekteja saada välttämättä edes avattua. Jotta esimerkiksi Schneider Electricin aikaisemmalla kehitysympäristöllä, PL7:llä, tehtyä projektia voidaan editoida Unity Pro:ssa, on projekti ensin muunnettava Unity Pro ymmärtämään muotoon [Schneider Electric, 2006]. Lisäksi PL7:n ja Unity Pro:n vakiokirjastot eroavat oleellisesti toisistaan. Aina tämänkaltainen muunnos ei ole mahdollinen, vaan eri kehitysympäristöistä joudutaan asentamaan eri versiota. Myös eri käyttöjärjestelmäversiota joudutaan pitämään yllä. Koska automaatiojärjestelmän elinkaari on pitkä verrattuna yleiseen tietotekniikan kehitykseen, ollaan helposti tilanteessa jossa joudutaan pitämään yllä useita eri kehitysympäristöjä eri käyttöjärjestelmien päällä (MS-DOS, MS Windows 2000, MS Windows XP jne.). Yksi kehitysympäristö valmistajaa kohden ei edes riitä välttämättä, vaan eri laitteistoille on eri ympäristöt. Esimerkiksi Schneider Electricin pienimmälle logiikalle Twidolle on oma kehitysympäristönsä Twidosoft, eikä se siis ole yhteensopiva Unity Pro:n kanssa.

8 Johtopäätökset

Tämän työn tavoitteena oli kehittää automaatiojärjestelmän sovelluskehitykseen ohjelmakirjasto, joka olisi käytettävissä eri ohjelmointiympäristöissä. Samalla oli tarkoitus tutkia, mitä vaatimuksia kirjaston käyttö asettaa sovelluksen suunnittelulle ja ohjelmoinnille.

Jo varhaisessa vaiheessa kävi selväksi, ettei moduuleja voitaisi siirtää ympäristöstä toiseen pelkästään kopioimalla moduulin syntaksi, koska kehitysympäristöt eroavat liikaa toisistaan, vaikka ne valmistajan mukaan olisivatkin IEC 61131-3 standardinmukaisia. Standardin tarkoituksena on mm. yhtenäistää logiikkaohjelmoinnissa käytettävät kielet, jolloin sovellusten ja moduulien siirtäminen kehitysympäristöjen välillä olisi helpompaa. Tämä jää kuitenkin toteutumatta, koska IEC 61131 standardi ei rajaa riittävän tarkasti vaatimuksia, jotka tuotteen on täytettävä ollakseen standardin mukainen [SAS, 2005]. Valmistaja voi itse valita, mitkä ominaisuudet se toteuttaa standardin mukaisesti ja mitkä vapaasti haluamallaan tavalla. Tästä huolimatta valmistaja voi mainostaa tuotettaan standardin mukaisena.

Tässä työssä tarkastelluista kehitysympäristöistä molemmat, sekä Unity Pro että Step 7, täyttävät valmistajiensa mukaan standardin. Kuitenkin Step 7:ssä joudutaan toimilohkoa kutsuttaessa käyttämään erillistä datalohkoa, mikä tekee ohjelmoinnista monimutkaisempaa ja lisää virheiden mahdollisuutta. Step 7:ssä ovat nähtävissä jäänteet vanhasta DIN 19239 standardista ja siinä käytetään edelleen monia eri lohkotyyppejä, kuten organisointi-, ohjelma-, sekvenssi- ja datalohkoja. Jälkikäteen mukaan on liitetty pieni osa uudemman IEC 61131 standardin mukaisia toimintoja. Unity Pro:ssa on laajempi joukko IEC 61131 standardin mukaisia toimintoja. Toimilohkon kutsu tapahtuu täysin standardin mukaisesti. Lisäksi siinä on mahdollisuus projektikohtaisesti valita, käytetäänkö laajennettuja, ei standardeja ominaisuuksia.

Ohjelmien siirrettävyyden kannalta olisi hyödyllisempää, että eri valmistajat noudattaisivat tarkemmin standardeja. Tämä tekisi ohjelmoinnista suoraviivaisempaa, varsinkin jos käytössä on useampia kehitysympäristöjä. Uuden IEC 61499 standardin on tarkoitus täydentää IEC 61131 standardia ja korjata sen puutteita [Lewis, 2001]. Tätä kirjoittaessa standardia IEC 61499 tukevia kehitysympäristöjä on niukasti saatavilla. Olisi toivottavaa, että valmistajat ottaisivat tuotteissaan käyttöön uuden standardin, eivätkä ryhtyisi kehittämään omia suljettuja ratkaisujaan.

Työn yhteydessä saatiin luotua eri kehitysympäristöille lähes samanlaiset ohjelmakirjastot. Ongelmia aiheuttivat aiemmin mainitut kehitysympäristöjen eroavuudet. Ottamalla nämä eroavuudet huomioon lohkojen suunnittelussa ja valitsemalla mm. rajapinnat ja muuttujien tietotyypit sopivasti saatiin lähes kaikki toiminnallisuudet toteutettua sekä Unity Pro:ssa että Step 7:ssä. Toimilohkon syntaksia hieman muokkaamalla lohko saatiin usein siirrettyä kehitysympäristöstä toiseen. Kirjastoon sijoitettiin sekä yksinkertaisia ja hyvin usein käytettyjä toimilohkoja että monimutkaisempia ja harvemmin käytettäviä toimilohkoja. Toimilohkot vaihtelivat yksinkertaisesta venttiilin ohjauslohkosta melko monimutkaiseen taajuusmuuttajan ohjauslohkoon. Vaikka esimerkiksi venttiilin ohjauslohko yksinkertaisuutensa vuoksi voi-

taisiin kirjoittaa joka projektissa alusta ilman kirjaston apua, ei se välttämättä ole kannattavaa, koska kirjaston avulla ohjelmista saadaan yhtenäisempiä ja yksinkertaiset kirjoitusvirheet saadaan käytännössä eliminoitua. Monimutkaiset toimilohkot kannattaa puolestaan sijoittaa kirjastoon niiden sisältämän tietotaidon takia, vaikkei toimilohko olisikaan kovin usein käytetty. Joka tapauksessa koodin uudelleen käyttö kirjastojen kautta on hyödyllistä [SAS, 2005] Tulevaisuudessa älykkäiden kenttälaitteiden, kuten taajuusmuuttajien ja muiden sulautettua elektroniikkaa sisältävien laitteiden määrä kasvanee [Lewis, 2001]. Näiden laitteiden sisältämä monipuolinen toiminnallisuus vaatii jatkossa myös yhä monipuolisempia toimilohkoja niitä ohjaamaan.

Kirjaston lisäksi automaatiosovelluksissa otettiin käyttöön tässä työssä luomani yhtenäinen nimeämiskäytäntö, joka pitää sisällään mm. ohjeet muuttujien nimeämiseen. Yhtenäinen nimeämiskäytäntö katsottiin Miprolla erittäin tarpeelliseksi, jotta ohjelmakoodista saataisiin selkeämpää ja virheiden määrä minimoitua. Tutkimuksissa on todettu selkeiden nimien helpottavan ohjelointia ja testausta [Jones, 2008]. Tässä työssä luomani symbolien nimeämiskäytäntö on toteutettu siten, että se on sellaisenaan tai vain pienin muutoksin sovellettavissa erilaisissa kehitysympäristöissä.

Kirjastosta ja yhtenäisistä muuttujien nimistä saatiin Miprolla heti positiivisia kokemuksia, kun mm. uusien ohjelmoijien oli helpompi tulla mukaan projekteihin ja sovelluskehitystä voitiin sujuvasti tehdä yhtä aikaa eri toimipisteissä. Jatko näyttää, kuinka paljon näillä menetelmillä saadaan tehostettua sovelluskehitystä ja saadaanko myös sovelluksista korkealaatuisempia.

Kirjaston käyttöä voitaisiin jatkossa edelleen tehostaa dokumentoimalla kirjaston sisältö dynaamisesti verkkopalvelimelle. Tällöin käyttäjät voisivat tarkastella kirjastoa esimerkiksi internet-selaimella lähiverkon yli, riippumatta siitä, onko työasemaan asennettu kaikkia eri kehitysympäristöjä tai onko kelluvia lisenssejä sillä hetkellä käytettävissä. Näin kirjasto olisi helposti kaikkien saatavilla ja kirjastoon tehdyistä muutoksista ja päivityksistä olisi helppo tiedottaa.

Lisäksi voisi olla hyödyllistä tutustua muihin markkinoilla oleviin logiikoihin ja niiden kehitysympäristöihin. Voitaisiin mm. selvittää, kuinka hyvin ne noudattavat standardeja, miten ne hyödyntävät kirjastoja ja onko niillä työskentely tuottoisampaa kuin nykyisillä työkaluilla. Näin olisi mahdollista löytää uusi logiikkaperhe ja kehitysympäristö, joilla projektit voitaisiin toteuttaa tehokkaammin. Uuden logiikan käyttöönotto on iso askel, ja päätökseen vaikuttavat kehitysympäristön ominaisuuksien lisäksi myös olennaisesti logiikan ominaisuudet, laitteiston ja lisenssien hinnat, toimitusajat, koulutuksen ja tuen saatavuus jne. Tämä voi kuitenkin olla kannattava vaihtoehto, jos nykyiset kehitysympäristöt koetaan puutteellisiksi.

Viitteet

1. ANSI/ISA-95-00-01 2000. Enterprise-Control System Integration Part 1: Models and Terminology. The Instrumentation, Systems and Automation Society, 79s.
2. Bryan L.A, Bryan E.A, 1997, Programmable Controllers - Theory and Implementation, Industrial Text Company, 1047s.
3. Gale B.T., 1994, Managing Customer Value: Creating Quality and Service that Customers Can See. The Free Press, Maxwell Macmillan
4. Haikala Ilkka ja Märijärvi Jussi, 2004, Ohjelmistotuotanto, 10. painos, Talentum, 440s.
5. Hilliard Rich, Recommended Practice for Architectural Description of Software-Intensive Systems, (www.enterprisearchitecture.info/ Images/Documents/IEEE 201471-2000.pdf, viitattu 19.6.2008)
6. HSE 1995. Out of Control - Why control systems go wrong and how to prevent failure. Sudbury (UK), Health and Safety Executive (HSE)/HSE Books, 60 s.
7. IEC 61131-3 Programmable controllers - Part 1: Programming languages, 2003
8. IEC 61131-3 Programmable controllers - Part 2: Programming languages, 2003
9. IEC 61131-3 Programmable controllers - Part 3: Programming languages, 2003
10. IEC 61131-3 Programmable controllers - Part 4: Programming languages, 2003
11. IEC 61131-3 Programmable controllers - Part 5: Programming languages, 2003
12. IEC 61499-1 Function blocks - Part 1: Architecture , 2005
13. IEC 61499-2 Function blocks - Part 2: Software tool requirements , 2005
14. IEC 61499-3 Function blocks for industrial-process measurement and control systems - Part 3: Tutorial information, 2004
15. IEC 61499-4 Function blocks - Part 4: Rules for compliance profiles, 2005
16. IEEE 610.12, 1990, Glossary of Software Engineering Terminology
17. IPC 2501 2003. Definition for web-based exchange of XML data (message broker). Association Connecting Electronics Industries (IPC), July 2003, 31s www.ipc.org [viitattu 8.7.2008]
18. ISO 15504-1 1998 Information technology - Software process assesment -Part 1: Concepts and introductory quide
19. ISO 15504-1 1998 Information technology - Software process assesment -Part 2: A reference model for processes and process capability

20. Jack Hugh, 2007, Automating Manufacturing Systems with PLCs, Version 5.0, 839s.

21. John Karl-Heinz, Tiegelkamp Michael, 2001, IEC 61131-3: Programming Industrial Automation Systems, Springer, 376s.

22. Jones Derek M, 2008 The New C Standard - An Economic and Cultural Commentary, 1615s.

23. Kruchten P., 1995, The 4+1 View Model of Architecture. IEEE Software, s 42-50.

24. Lewis Robert, 2001, Modelling control systems using IEC 61499, The Institution of Electrical Engineers, 192s.

25. OMAC 2002. OMAC baseline architecture -Functional requirements, version 1.0 Open Modular Architecture Control (OMAC), packaging workgroup, April 24, 2002, www.omac.org [viitattu 8.7.2008]

26. Prosessori 12/2007, Marko Mattila, Mikko Salmenperä, Toni Kalajainen ja Matti Paljakka, Yhtenäinen OPC UA -arkkitehtuuri automaatioon, s.50

27. Schneider Electric, Unity Pro User manual, 2006

28. Siemens 2006a, Programming with Step 7, edition 03/2006

29. Siemens 2006b, Working with Step 7, edition 03/2006

30. Stenberg A., Ohjelmiston testaus. SataSPIN koulutusmateriaali, 2003

31. SAS, 2005, Automaatiosovellusten ohjelmistokehitys, Suomen Automaatioseura ry, 151s.

32. SAS, 2001, Laatu automaatiossa - parhaat käytännöt, Suomen Automaatioseura ry, 245s.

33. Tian Jeff, 2005, Software Quality Engineering - Testing, Quality Assurance, and Quantifiable Improvement, Wiley, 441s.

Liite A

Esimerkki standardin IEC 61131 mukaisesta Instruction List (IL) ohjelmalistauksesta:

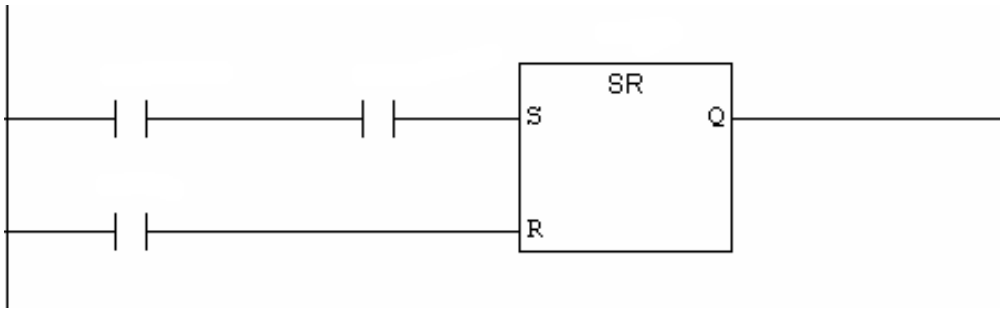
```
          LD     FLOW
          GT     250
          JMPCN  PROS_OK  (* Tämä on kommentti *)
          LD     Volts
PROS_OK   LD     1
          ST     %Q75
```

Esimerkki standardin IEC 61131 mukaisesta Structured Text (ST) ohjelmalistauksesta:

```
(* Ehtolause *)
IF MAX(Temperature) <= MaxTemp
THEN
    Heating := TRUE;
ELSE
    Heating := FALSE;
END_IF;
```

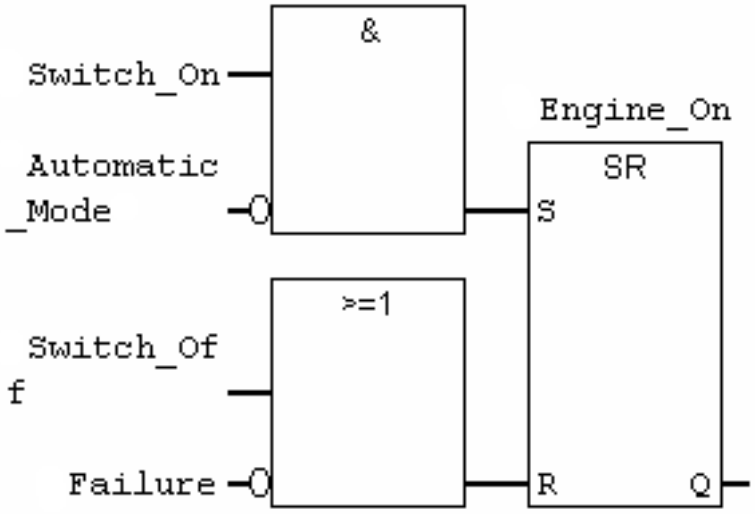
Liite B

Esimerkki standardin IEC 61131 mukaisesta Ladder Diagram (LAD) ohjelmalistauksesta:



Kuva B1: Ladder Diagram ohjelmalistaus

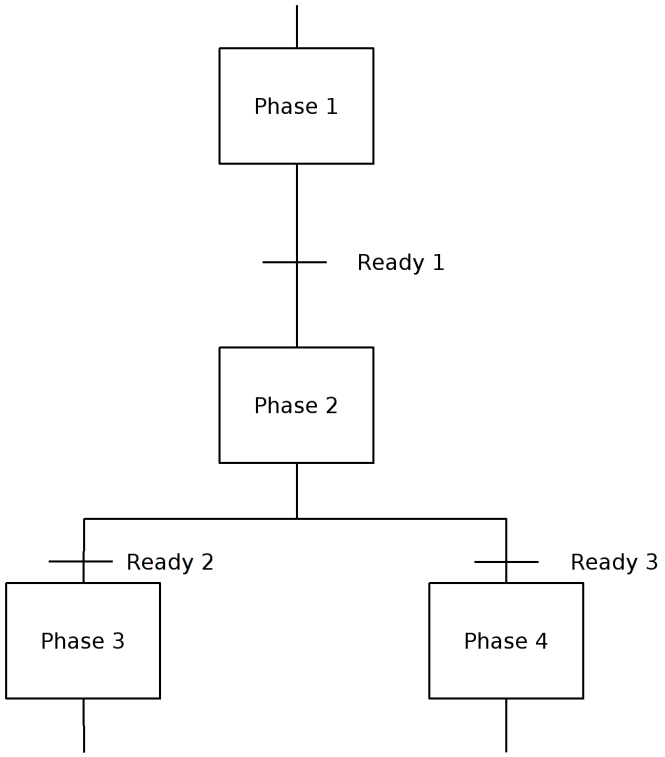
Esimerkki standardin IEC 61131 mukaisesta Function Block Diagram (FBD) ohjelmalistauksesta:



Kuva B2: Function Block Diagram ohjelmalistaus

Liite C

Esimerkki standardin IEC 61131 mukaisesta Sequential Function Chart (SFC) ohjelmalistauksesta:



Kuva C1: Sequential Function Chart ohjelmalistaus