Riktlinjer för verktygsstöd vid felsökning i PLC-system enligt standard IEC 1131-3

(HS-IDA-EA-99-109)

Andreas Hansson (a96andha@student.his.se)

Institutionen för datavetenskap
Högskolan i Skövde, Box 408
S-54128 Skövde, SWEDEN

Examensarbete på Programmet för systemprogrammering under vårterminen 1999.

Handledare: Joakim Eriksson

Riktlinjer för verktygsstöd vid felsökning i PLC-system enligt standard IEC 1131-3

Examensrapport inlämnad av Andreas Hansson till Högskolan i Skövde, för Kandidatexamen (B.Sc.) vid Institutionen för Datavetenskap.

990618

Härmed intygas att allt material i denna rapport, vilket inte är mitt eget, har blivit tydligt identifierat och att inget material är inkluderat som tidigare använts för erhållande av annan examen.

Signerat: _____

Riktlinjer för verktygsstöd vid felsökning i PLC-system enligt standard IEC 1131-3

Andreas Hansson (a96andha@student.his.se)

Sammanfattning

En stor del av tiden vid utveckling av programvaror går idag ut på att leta efter fel i dem. Oavsett hur mycket tid och pengar som läggs ner på denna aktivitet är det inte säkert att alla fel hittas. Detta kan få katastrofala följder, speciellt för realtidssystem då även fysisk apparatur skall skötas. Då ett fel verkligen har påvisats gäller det att försöka lokalisera och identifiera felet. Då detta kan ske under tidspress, till exempel om felet rör ett realtidssystem redan i användning, behövs stort stöd från felsökningsverktyg för att underlätta att snabbt lokalisera och identifiera ett fel.

Detta examensarbete identifierar genom en teoretisk undersökning ett antal riktlinjer för de stöd ett felsökningsverktyg kan ge felsökaren under felsökningsprocessen. Riktlinjerna är utformade så att de kan användas i felsökningsverktyg i alla typer av programvaror. Riktlinjer är även särskilt anpassade till standard IEC 1131-3 för PLC-system, vilka är en typ av realtidssystem.

Nyckelord: Felsökning, felsökningsverktyg, realtidssystem, PLC-system, IEC 1131

Innehållsförteckning

1 Introduktion 1
2 Bakgrund 2
2.1 Målmiljö: BiFas UHS 2
2.2 Standard IEC 1131-3 och BiSoft 1131 2
2.2.1 IEC 1131-3 2
2.2.2 BiSoft1131 4
2.3 Testning med BiSoft 1131 – övergripande 4
2.4 Testvänlighet hos en programvara 5
2.5 Feltyper i realtidssystem 6
2.6 Felsökning 7
2.6.1 Felsökning genom ”brute force” 7
2.6.2 Felsökning genom induktion 8
2.6.3 Felsökning genom deduktion 9
2.6.4 Felsökning genom backtracking 10
2.6.5 Felsökning genom testning 10
3 Problemprecisering 11
3.1 Beskrivning av felsökningsförutsättningar 11
3.2 Beskrivning av riktlinjer 12
3.3 Problembegränsningar 12
3.4 Uppgift 13
4 Val av lösningsmetoder 14
4.1 Riktlinjer för verktygsstöd till felsökningsmetoder 14
4.1.1 Utforskning av felsökningsmetoder 14
4.1.2 Undersökning av felsökningsverktyg 14
4.1.3 Intervju 15
4.1.4 Val av metod 15
4.2 Användning av riktlinjer tillsammans med IEC 1131-3 15
4.2.1 Intervjuer av användare 16
4.2.2 Teoretisk undersökning av standardens möjligheter att stödja riktlinjer 16
4.2.3 Val av metod 16
5 Undersökning av felsökningsmetoder 18
5.1 Brute force 18

5.2 Induktion 20
5.3 Backtracking 22
6 Felsökningsstöd i IEC 1131-3 24
6.1 Lista över identifierade stöd till felsökningsmetoder 24
6.1.1 Brute force 24
6.1.2 Induktion 25
6.1.3 Backtracking 25
6.2 Riktlinjer för felsökning med brute force 25
6.3 Riktlinjer för felsökning med induktion 31
6.4 Riktlinjer för felsökning med backtracking 35
6.5 Användning av riktlinjer tillsammans med IEC 1131-3 37
6.5.1 Riktlinjer i brute force-metoden 37
6.5.2 Riktlinjer i induktionsmetoden 39
6.5.3 Riktlinjer i backtrackingmetoden 40
7 Slutsatser 42
7.1 Riktlinjer för verktygsstöd vid felsökning 42
7.2 Riktlinjernas användbarhet tillsammans med IEC 1131-3 43
7.3 Diskussion 43
7.4 Fortsatt arbete 43
Appendix A Binär elektronik AB 45
Referenser 46

1 Introduktion

Stora delar av tiden för utveckling av programvaror upptas idag av processen att hitta fel i programvaran (Pressman, 1997). Det är nästan en oskriven lag att det i varje program, hur mycket programutvecklaren/programutvecklarna än felsöker, finns minst ett fel som går oupptäckt genom alla tester och som överlever tills det att programmet tas i bruk. För programutvecklarna gäller också att ju senare ett fel upptäcks desto dyrare är det att rätta till felet. Även för de företag som använder sig av program i kritiska system kan det bli väldigt kostsamt om detta inte följer det som programmets specifikation utlovar.

En typ av program är särskilt sårbar för fel som upptäcks för sent. Det rör sig om program som styr maskiner av olika slag, t ex respiratorer inom sjukvården eller robotar inom industrin. Dessa program styr processer som kan vara livsavgörande för vissa (till exempel i fallet med sjukvårdsutrustning) eller förenade med stora ekonomiska och materiella risker eller rent av livsfara (till exempel i fallet med industrimaskiner). Det är i dessa fall viktigt att hitta fel innan de ställer till förödelse ute i den verksamhet programmet är avsedd att stödja. Ett annat problem som existerar och som relaterar till felsökningen är att det är väldigt svårt att utföra realistiska tester på programmet utan att införa programmet i den verksamhet som den skall stödja. Programutvecklarna har ofta inte de maskiner som programmet skall styra till sitt förfogande vid testning, utan får klara sig med beskrivningar om hur det är tänkt att programmet skall fungera. Det hör även till ovanligheterna att programutvecklarna har hjälp av simulatorer som kan erbjuda en någorlunda realistisk testmiljö.

Om nu ett fel i ett kritiskt system inte upptäcks förrän programmet redan har tagits i bruk är det inte alltid felet uppträder så tidigt att den utvecklare som installerar programmet finns kvar på plats och kan ta itu med att identifiera problemet. Många gånger arbetar dessutom programutvecklaren inte ens på företaget, utan har anlitats av ett företag specialiserat på denna typ av programvaror. De första som då kommer i kontakt med problemet är den underhållspersonal som arbetar hos kunden. Det är en stor fördel om felsökning i programvaran då är så lätt att de själva kan ge sig på att försöka hitta felet.

Binär Elektronik AB är ett företag som inriktar sig på att automatisera industriella uppgifter och utvecklar datorbaserade system som används till styrning av olika produktionsanläggningar. Mest handlar detta om positioneringssystem för servomotorer, det vill säga system som bestämmer rörelser till servomotorer så att de är i rätt lägen vid rätt tidpunkt, samt PLC-system (Programmable Logic Controller) som används för att styra maskiner samt sköta kommunikation mellan dem. Se appendix A för vidare beskrivning av företaget.

Detta arbete tar fram riktlinjer för hur felsökningsverktyg till den typ av programvaror företaget utvecklar skall se ut för att fel kan upptäckas så smidigt som möjligt.

2 Bakgrund

Det finns ett antal begrepp läsaren bör vara införstådd med för en ökad förståelse av arbetet samt de förutsättningar som föreligger.

2.1 Målmiljö: BiFas UHS

I inledningen, kapitel 1, nämns att detta arbete syftar till att ta reda på vilket stöd ett felsökningsverktyg kan ge processen att hitta fel i PLC-system. Det är därför viktigt att veta vad en PLC är.

En PLC används för att styra maskiner och/eller tillverkningsprocesser främst inom industrin. PLC:n är en dator till vilken I/O-enheter kan anslutas för att styra maskiner eller serier av maskiner. Kommunikation mot operatörer är ofta sparsam och sker främst genom särskilda operatörspaneler vilka innehåller en bildskärm och ett par knappar. Vid testning och överföring av PLC-program till realistisk miljö måste testaren koppla upp en egen dator mot PLC:n och däröver styra PLC:n.

BiFas UHS är ett kombinerat PLC- och positioneringssystem. Till denna enhet ansluts olika typer av externa enheter såsom operatörspaneler, I/O-enheter och servoförstärkare. BiFas UHS kan hantera upp till 4096 I/O enheter och 10 servoaxlar. Det är genom dessa I/O-enheter maskinerna styrs. Systemet har den egenskapen att oavsett storlek på kundens system kan samma produkt köpas och sedan utökas med nya funktioner allt eftersom kundens behov ändras. Vid uppgradering av systemet byts bara det kort som innehåller det gamla programmet mot ett annat som tillhandahåller den önskade funktionaliteten. Detta innebär att lagren för reservdelar kan minskas och underlättar handhavandet och programmeringen av systemet.

2.2 Standard IEC 1131-3 och BiSoft 1131

Länge var PLC-marknaden utan standarder. Dock, när en marknad eller industri når en viss storlek och flera olika sorters system måste arbeta tillsammans bör standarder införas för att underlätta för både producenter av systemen samt användarna (Lewis, 1996). Detta har inträffat i PLC-industrin.

2.2.1 IEC 1131-3

1979 startade IEC (International Electro-technical Commission) en arbetsgrupp för att ta fram en fullständig design för PLC. Denna design skulle innefatta hårdvarudesign, installation, testning, dokumentation, programmering och kommunikation. 1993 publicerades PLC-standarden IEC 1131. Denna standard består av fem delar (Lewis, 1997). Del ett definierar den grundläggande terminologin och de grundläggande begreppen. Del två handlar om elektronisk och mekanisk konstruktion samt verifikationstester. Del tre handlar om mjukvarustruktur, programmeringsspråk samt programexekvering. Del fyra handlar om urval, installation och underhåll av PLC-system och del fem handlar om kommunikationen mellan olika PLC-system. För detta arbete är det främst del tre som är intressant eftersom denna del är den som rör mjukvarukonstruktion och därmed även testning av och felsökning i mjukvaran (denna del benämns från och med nu IEC 1131-3).

IEC 1131-3 erbjuder flera olika programmeringsspråk (Lewis, 1997). Detta innebär att en programmerare själv kan avgöra vilket språk som främst lämpar sig för den uppgift som skall lösas. Dessa språk är: relälogik (eller Ladder, eng. Ladder Diagram) funktionsblocks-diagram (eng. Function Block Diagram), sekventiellt flödesschema (eng. Sequential Function Charts), instruktionslista (eng. Instruction List) samt strukturerad text (eng. Structured Text). De tre förstnämnda språken är grafiska och de båda sistnämnda är textbaserade. En del tillverkare av PLC-system som följer standarden har även konstruerat program som automatiskt kan översätta mellan dessa olika språk.

Relälogik (RL) används för att beskriva funktionella beteenden, grupper av funktioner samt program. Den används således för programmering både på hög och låg nivå. RL grundar sig på en teknik att beskriva logik genom att använda reläer (Lewis, 1997). Det är ett grafiskt språk där grafiska figurer som symboliserar program, funktioner eller enskilda komponenter, till exempel kontakter, kopplas samman i sekvenser för att leda fram till en händelse, till exempel att en ventil beordras att stängas av programmet. Dock anklagas RL ofta för att ha svårt med bland annat beskrivandet av komplexa programsekvenser, programmerandet av komplexa aritmetiska funktioner, komplexa eller hierarkiska programstrukturer samt med manipulerandet av datastrukturer (Lewis, 1997).

Funktionsblocksdiagram (FBD) uttrycker funktionellt beteende, grupper av funktioner och program som en mängd av kopplade grafiska block. FDB används bäst när problemet även blandar in flödet av signaler mellan olika block av funktioner. Det är väldigt likt eldiagram eller kopplingsscheman för elkretsar. Diagrammen beskriver elektriska kopplingar mellan komponenter, varför en del produktutvecklare, familjära med dessa diagram, föredrar detta språk framför de textbaserade (Lewis, 1997). Likt relälogiken kopplas grafiska figurer som symboliserar program, funktioner eller enskilda komponenter samman. Till skillnad från relälogiken är dock inte funktionsblocksdiagrammen lika strikt sekventiella, varför en del av de svårigheter som drabbas relälogiken inte är lika genomslående i funktionsblocksdiagram.

Sekventiellt flödesschema (SFC) har tagits fram för att grafiskt kunna programmera sekventiellt beteende. Det har inspirerats av den franska standarden Grafcet, vilket var allmänt använt innan sekventiellt flödesschema kom till. SFC kan användas för att beskriva alla olika aspekter på sekventiella flöden. Det kan användas för att beskriva såväl de mest generella aspekterna i ett program som lågnivådetaljer. En sekvens beskrivs med grafiska block som är kopplade med varandra, där blocken representerar olika tillstånd i programmet. Språkets styrka är att det tydligt visar vilka tillstånd programmet kan befinna sig i samt hur dessa skall lämnas (Lewis, 1997).

Instruktionslista (IL) är ett lågnivåspråk som påminner om assembler. IEC skapade språket genom att titta på de olika lågnivåspråk PLC-tillverkarna använde sig av. IL lämpar sig för att programmera korta, lätta problem med få selektioner och där programflödet inte ändrar sig särskilt mycket. Språket anses vara väldigt enkelt att lära sig samt att använda. Ibland behöver programkoden inte ens kompileras, utan kan laddas ner direkt i målmiljön. Många programmerare anser att IL är det språk alla andra språk i standarden relativt enkelt kan översättas till (Lewis, 1997).

Strukturerad text (ST) är ett högnivåspråk som påminner om PASCAL. Språket är dock utvecklat speciellt för att styra system inom industrin. ST används för att uttrycka olika sorters beteenden som involverar många olika typer av data. Språket anses vara ganska enkelt att lära sig samt att använda (Lewis, 1997).

Lewis (1997) räknar upp ett antal fördelar med att använda standard IEC 1131-3:

- Den erbjuder ett antal olika språk som kan användas för att lösa olika typer av problem
- Den uppmuntrar produktion av program med hög kvalitet genom väl strukturerad design samt användandet av inkapsling
- Användandet av IEC 1131-3 möjliggör att kontrollprogrammen kan användas i olika PLC-produkter
- Den erbjuder stöd för framtagandet av återanvändningsbar programvara vilket kan öka produktiviteten hos programmerarna
- Programmeringsverktyg tas fram som kan underhålla PLC-programmen genom hela deras livscykel.

2.2.2 BiSoft1131

BiSoft 1131 stöder alla programmeringsspråk i standarden IEC 1131-3 och är det utvecklingsverktyg som används vid programmering på Binär. Det är framtaget i ett samarbete mellan ett antal PLC-tillverkare, däribland Binär. Detta verktyg är fortfarande delvis under utveckling. Det är bristerna i stödet för testning och felsökning i detta verktyg som föranleder detta arbete. Bland annat är programmerarna som arbetar i verktyget missnöjda med representationen av data, åtkomst och kontroll över variabler vid felsökning med mera.

2.3 Testning med BiSoft 1131 – övergripande

Programvaran till BiSoft-produkterna är till så stor grad som möjligt uppbyggda av moduler. Dessa moduler är ofta programmerade så att de kan användas i många olika program. Modulerna testas separat innan de används. Därefter sker integrationstester, för att utröna så att programmet som helhet beter sig som det skall. Hos Binär sker utveckling och testning till så stor del på företaget som möjligt. Svårigheter är att tillgång till de maskiner som skall styras ofta inte finns. Kommunikationen mellan den enhet programmet exekverar på, målmiljön (se kapitel 2.1) och den dator programmeraren använder, utvecklingsmiljön, riskerar också att vara långsam. Det som kan ses av programmeraren under programmets exekvering är variabelstatus (om dessa uppdateras så sällan att kommunikationen mellan datorerna faktiskt hinner med), samt, i de grafiska språken, även i vilken programmodul programmet exekverar. Ytterligare testmöjligheter som finns är att utföra mätningar på de I/O portar som finns, så att kommunikationen med omvärlden fungerar som den skall.

För att identifiera fel läggs stor vikt vid olika felmeddelanden och statusvisningar på operatörsskärmarna till BiFas-produkterna. Det är dock svårt att få bra och heltäckande felmeddelanden och dessa visar oftast bara följdfel till det egentliga felet.

De sista testerna sker på plats vid installationen av produkten till den maskin som skall skötas. Tiden som finns till hands då är i allmänhet starkt begränsad, produktionen skall helst inte bli lidande över huvud taget. Det är först här som programmeraren kan se hur programmet uppträder i den målmiljö det är tänkt att den skall styra.

2.4 Testvänlighet hos en programvara

För att fel lätt skall kunna hittas och åtgärdas är det nödvändigt att stöd för testning redan fås redan under utvecklingsfasen. Det går inte att betrakta utveckling av ett program och testningen av detsamma som två separata faser, utan en bra utveckling är beroende av en bra testning och vice versa. Pressman (1997) definierar ett antal kriterier som avgör hur testbar programvaran är, det vill säga hur lätt det är att testa den. Nedanstående lista är anpassad med avseende på standard IEC 1131-3 och beskriver strukturen och egenskaperna hos ett program som följer standarden. Det som skiljer sig från källan är att det inte går att göra antagandet att felen inte bryter exekveringen av ett test, att tidigare genomgångna tillstånd och tidigare värden på variabler är åtkomliga vid fel, att funktioner bara innehåller precis det som krävs för att lösa sina uppgifter samt att förändringar i mjukvaran inte riskerar att göra tidigare tester invalida.

- Operabilitet (eng. Operability)
  a) Systemet har få fel
  b) Produkten tillåter testning och utveckling att ske samtidigt.
- Observabilitet (eng. Observability)
  a) Distinkta utdata genereras för varje indata
  b) Systemets tillstånd och variabler är synliga eller åtkomliga under exekvering
  c) Alla faktorer som påverkar utdatan är synliga
  d) Felaktig utdata är lätt att identifiera
  e) Interna fel detekteras genom själv-testande mekanismer
  f) Interna fel rapporteras automatiskt
  g) Tillgång till källkod finns.
- Kontrollerbarhet (eng. Controllability)
  a) Alla möjliga utdata kan genereras genom någon kombination av indata
  b) All kod är exekverbar genom någon kombination av indata
  c) Tillstånd och variabler både i hård- och mjukvara är direkt kontrollerbara av testaren
  d) Formaten på indata och utdata är konsekventa och strukturerade
  e) Tester kan enkelt specificeras, genomföras och reproduceras.
- Strukturerbarhet (eng. Decomposability)
  a) Mjukvaran är uppbyggt av oberoende moduler
  b) Modulerna kan bli testade oberoende av varandra.
- Enkelhet (eng. Simplicity)
  a) Strukturell enkelhet, att modularisering används för att begränsa propagering av fel
  b) Enkel kod, att kodningen är standardiserad för att underlätta granskning och underhåll.

- Stabilitet (eng. Stability)
  a) Förändringar i mjukvaran inträffar sällan
  b) Förändringar i mjukvaran sker på ett kontrollerat sätt
  c) Mjukvaran återhämtar sig från fel på ett bra sätt.
- Lättförståelighet (eng. Understandability)
  a) Designen är väl förstådd
  b) Beroenden mellan interna, externa och delade komponenter är begripliga
  c) Förändringar i designen meddelas till alla berörda
  d) Teknisk dokumentation är direkt åtkomlig
  e) Teknisk dokumentation är väl organiserad
  f) Teknisk dokumentation är detaljerad och noggrann
  g) Teknisk dokumentation är korrekt.

2.5 Feltyper i realtidssystem

Det finns ett antal olika fel som kan drabba en programvara. Dessa är mer eller mindre allvarliga beroende på omständigheterna runt felet samt i vilken omgivning programmet befinner sig. Eftersom de system detta arbete inriktas mot (se kapitel 2.1 om målmiljö samt i inledningen kapitel 1) kraftigt interagerar med omgivningen så till vida att programmen skall styra industrimaskiner tillkommer även tidsaspekter. PLC-systemen är realtidssystem. Därför är det viktigt att vara medveten om vilka olika typer av fel som kan komma att dyka upp i denna typ av programvaror.

När ett program är fel upptäcks detta genom att systemet inte beter sig som förväntat enligt specifikationen (denna nivå av fel heter på engelska ”failure”). Genom att studera hur systemet beter sig när felet uppträder försöker programmerarna att lista ut orsakerna till att systemet inte beter sig som specificerat. Det går således inte att direkt undersöka vad som är fel, utan symptomen systemet ger ifrån sig externt måste användas för att lista ut det interna felet (eng. fault).

Burns och Wellings (1997) identifierar olika typer av fel som kan inträffa i realtidssystem beroende på vilka konsekvenser de får. Två generella typer identifieras.

- Värdefel, det vill säga systemet ger inte ifrån sig önskat värde eller önskade resultat.
- Tidsfel, det vill säga systemet producerar rätt resultat, men inte i rätt tid.

Burns och Wellings nämner även en tredje möjlighet till fel vilket är kombinationen av värdefel och tidsfel.

Ett värdefel kan vara av två olika typer. Dels kan värdet vara fel, men fortfarande godtas av systemet för att det liknar det rätta värdet. Dels kan värdet vara så fel att det ligger utanför de förväntade värdena som systemet skall generera. Detta kallas på engelska för ”constraint error”. Dessa är ofta inte så svåra att hitta. De kan ha olika orsaker: det kan vara ett felaktigt programuttryck alternativt att ett nödvändigt programuttryck saknas.

Det finns även olika typer av tidsfel. Dels kan systemet ge resultat för tidigt, innan det är tänkt att systemet skall ge resultat. Dels kan systemet ge resultat för sent, efter det att systemet skall ge resultat och dels finns en tredje typ där systemet aldrig producerar ett resultat.

Burns och Wellings identifierar även en typ av fel då systemet genererar en tjänst eller ett resultat som är helt oväntat.

2.6 Felsökning

Felsökning avser i detta arbete processen att lokalisera och identifiera fel i ett program. Aktiviteten testning inkluderas inte i begreppet.

Vid framtagandet av riktlinjer för hur felsökningsverktyg för de avsedda systemen skall se ut är det viktigt att veta hur verktygen är tänkta att användas. Om kunskap om den metodik som kan komma att användas saknas kan det hända att de riktlinjer som tas fram hindrar istället för hjälper eller kanske inte har någon praktisk nytta. Om inte riktlinjerna är anpassade till de faktiska arbetsförhållandena är risken stor att arbetet är förgäves.

För att veta hur verktygen kan komma att användas måste de olika felsökningsmetoderna beaktas. När ett test har lyckats, det vill säga påvisat ett fel, påbörjas fasen att försöka se vilken typ av fel det är, att lokalisera felet samt att åtgärda detsamma. Det finns ett antal olika tillvägagångssätt att systematiskt utföra detta. Dessa baserar sig på olika filosofier om hur fel hittas. En del metoder försöker utifrån programmets tillstånd när felet inträffar utröna vart det ligger och av vilken karaktär det är. Andra försöker betrakta beteendet hos programmet i felögonblicket samt innan det var fel och genom att analysera de symtom felet ger upphov till ta reda på vad det är för fel. Analysen sker huvudsakligen utanför datorn och involverar mera tankeprocesser. Andra metoder går ut på att systematiskt arbeta sig igenom programmets olika tillstånd antingen bakåt eller framåt och därigenom spåra felet till sin uppkomstplats. Ytterligare metoder försöker genom utförliga tester lokalisera felet. Nedan ges exempel på olika felsökningsmetoder.

2.6.1 Felsökning genom ”brute force”

Enligt Paul (1997) är felsökningsmetoden ”brute force” den vanligaste, men även den minst effektiva metoden att lokalisera fel. Metoden kräver nästan ingen tankeinsats. Den karaktäriseras av att felsökning sker genom minnesdumpar, utspridda utskriftssatser i programmet eller genom felsökningsverktyg. Paul identifierar ett antal problem med de olika sätten att använda brute force.

Att använda sig av minnesdumpar har följande nackdelar (enligt Paul, 1997):

1. Det är svårt att lokalisera variabler.
2. Stora mängder irrelevant data måste behandlas.
3. Dumpen visar en ”stillbild” av programmets tillstånd. Ofta upptäcks felen lättast genom att se tillståndsförändringar.
4. Dumpen genereras sällan precis när felet uppstår, det vill säga dumpen visar sällan programmets tillstånd när felet uppträder.

5. Det finns ingen metod att systematiskt genomsöka en minnesdump för att lokalisera ett fel.

Utspridda utskriftssatser är enligt Paul (1997) ofta bättre än att undersöka minnesdumpar. Informationen är ofta lättare att förstå samt att den visar programmets dynamiska uppträdande. Dock finns följande nackdelar:

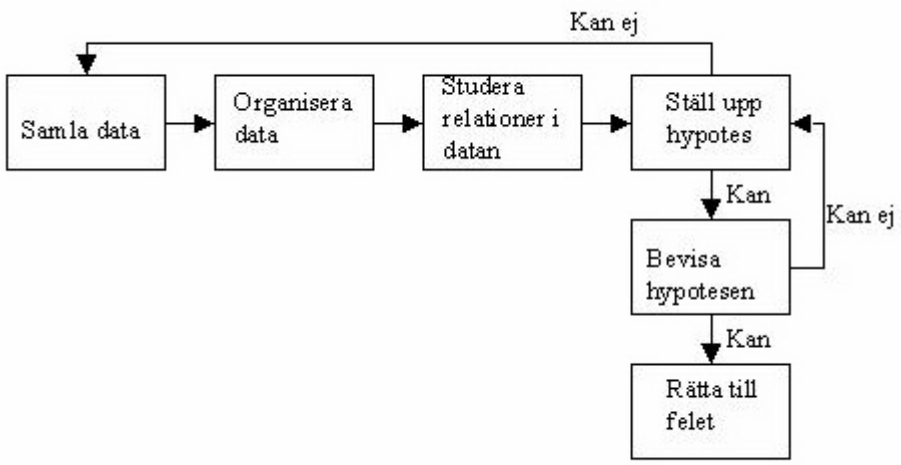
1. Det är fortfarande en metod som innebär att felet antingen hittas eller också missas totalt. Detta innebär att antingen så lokaliseras och identifieras felet eller så kommer felsökaren, den person som letar efter fel, inte närmare en lösning än han var innan.
2. Ofta blir det stora mängder data som måste analyseras.
3. Den kräver ändringar i programmet vilka kanske maskerar fel, påverkar kritiska synkroniseringar eller rent av introducerar nya fel.
4. I realtidssystem är det ofta kostsamt eller rent av omöjligt med denna felsökningsmetod.

Att använda sig av felsökningsverktyg innebär enligt Paul (1997) att, till skillnad från utskriftssatserna, inga programändringar behöver göras. Dock är det fortfarande stora mängder data som behöver analyseras, samt att det fortfarande bygger en del på tur om felet skall lokaliseras eller inte.

Paul konstaterar att *brute force* bara skall användas när de andra metoderna misslyckas eller som ett komplement till övriga metoder.

2.6.2 Felsökning genom induktion

Vid felsökning genom induktion undersöks de symptom felen genererar när de inträffar. För att utröna vad det är för fel samt vart felet är undersöks vilka symptom som felet ger upphov till samt relationer mellan symptom. Detta leder ofta till att slutsatser kan dras som finner felet. Paul (1997) ställer upp ett antal steg som skall genomgås metodiskt (dessa steg beskrivs även schematiskt i figur 1).

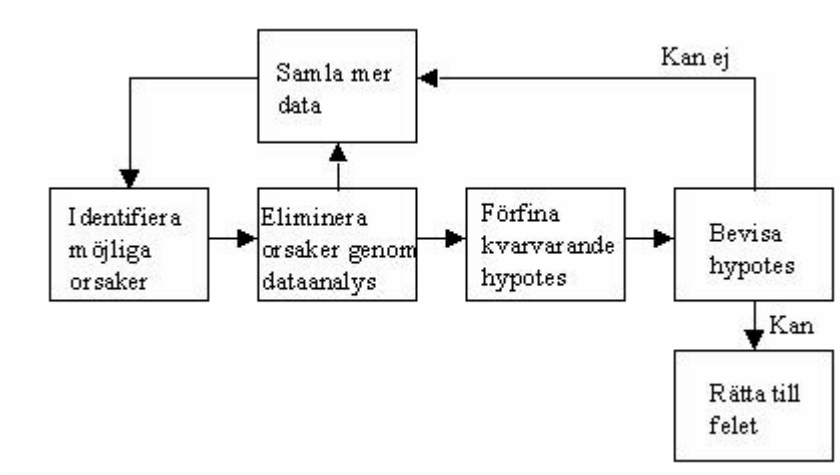


Figur 1. Processbeskrivning över felsökning genom induktion (Paul, 1997).

1. Tag reda på all information som kan fås ut från de tester som har genomförts. Det är lika viktigt att ta reda på vad som blir rätt som det är att ta reda på vad som blir fel i testerna. Testfall som inte genererar fel innehåller många gånger lika mycket information som de tester som genererar fel.
2. Organisera informationen. Tag reda på vad som händer vid fel, vart det händer, när det händer samt omfattning. Tag även reda på motsatsen, det vill säga när felet inte uppträder. Detta för att lättare kunna formulera en hypotes om vad det är för fel, det går kanske att utesluta vissa orsaker och se samband mellan fel.
3. Ställ upp en eller flera hypoteser om vad felet kan vara. Undersök samband mellan de ledtrådar som ställts upp genom att undersöka den information som organiserades i punkt 2. Kan ingen hypotes ställas finns det troligen inte tillräckligt med data, gå tillbaka till punkt 1 och utför fler tester. Om flera hypoteser kan ställas skall den mest troliga undersökas först.
4. Bevisa hypotesen. Paul varnar för att denna punkt allt för ofta hoppas över. Det är viktigt att visa att hypotesen är korrekt. Detta visas genom att se om hypotesen kan förklara alla symptom ur den insamlade datamängden. Kan inte hypotesen förklara allt är den antingen felaktig, ej komplett eller det finns flera fel. I så fall gå tillbaka till punkt 3. Annars, avlägsna felet.

2.6.3 Felsökning genom deduktion

Denna felsökningsmetod går ut på att, utifrån generella teorier och hypoteser om vad felet kan vara, komma fram till ett svar genom uteslutningsmetoder samt förfining av hypoteser. Paul (1997) drar upp följande steg (finns även redovisade i figur 2).



Figur 2. Processbeskrivning över felsökning genom deduktion (Efter Paul, 1997).

1. Räkna upp alla möjliga hypoteser och teorier om vad felet kan vara. Dessa behöver inte vara fullständigt förklarade, utan är till för att kunna strukturera och analysera data.
2. Använd data för att eliminera möjliga orsaker. Genom att analysera data (till exempel genom att organisera den enligt punkt 2 i avsnitt 2.6.2 om felsökning genom induktion) görs ett försök att utesluta alla möjliga orsaker utom en. Om alla

hypoteser utesluts behövs mer data. Om flera möjliga orsaker finns kvar välj då den mest troliga.

3. Förfina hypotesen. Det är inte troligt att hypotesen är exakt på felet. Den behöver förfinas. Använd tillgänglig data för att precisera hypotesen.
4. Bevisa hypotesen. Detta sker genom att jämföra ledtrådar och data med hypotesen. Om hypotesen inte kan bevisas behövs mer data.

2.6.4 Felsökning genom backtracking

Enligt Paul (1997) används denna metod bäst på små program. Den går ut på att starta där felet först uppenbarade sig och sedan stega sig tillbaka i programmet, exekvera programmet baklänges och återskapa tillstånd och variabler. Fortsätt på detta sätt, felet ligger någonstans mellan de punkter där programmet uppförde sig korrekt och där det uppförde sig fel.

2.6.5 Felsökning genom testning

Att använda sig av ytterligare testfall är, enligt Paul (1997), en kraftfull metod som ofta används för att generera, påvisa eller förfina hypoteser enligt induktions- eller deduktionsmetoderna.

Paul konstaterar att tester som används för felsökning skiljer sig från de tester som används för att påvisa fel så till vida att de ofta är mer specialiserade till att undersöka en viss domän eller ett specifikt programtillstånd. Tester för att påvisa fel är ofta generella och undersöker flera aspekter och möjligheter samtidigt, medan felsökningstester fokuserar på en aspekt och/eller en möjlighet. Felsökningstester är till för att lokalisera fel så effektivt som möjligt, medan tester för påvisande av fel är designade att upptäcka existensen av fel så effektivt som möjligt.

3 Problemprecisering

Detta arbete handlar om vad som behövs i utvecklingsverktygen som används vid framtagandet av programvaror för att felsökning skall kunna genomföras så smidigt som möjligt i program för styrning av maskiner som används i produktionsindustrin.

Produktionsmaskiner har hög interaktion med den fysiska miljö de befinner sig i. Till exempel, de maskiner Binär utvecklat program åt är bland annat maskiner för klippning av plåt från stålvalsar och maskiner för limning av fönsterrutor. Om det blir ett fel i denna typ av maskiner är detta förenat med potentiellt stora ekonomiska, materiella och kanske rent av personliga risker. Om ett fel finns i programvaran är det nödvändigt att hitta felet så snabbt som möjligt, särskilt om programmet redan är i användning.

Eftersom de maskiner som skall styras av de program detta arbete inriktar sig mot ofta är stora och otympliga förekommer det särskilda begränsningar vad gäller testning. Ofta har inte utvecklaren fri tillgång till den maskin som skall styras. Testning får i hög grad ske utanför den miljö programmet är anpassat för. Många gånger sker inte testning under realistiska förhållanden förrän utvecklarna befinner sig på plats och skall installera programvaran i maskinen. Detta sker då i allmänhet under stor tidspress; produktionen måste störas så lite som möjligt, varför det är viktigt att testning och särskilt felsökning, om fel hittas, kan ske så smidigt som möjligt. Det faktum att testning av programvaran måste ske i en miljö med begränsade kommunikationsmöjligheter till utvecklingsmiljön (se kapitel 2.1 om målmiljön) sätter begränsningar i hur felsökningsprocessen kan gå till. Minnet i målmiljön är så litet att fullständig loggning av en exekvering är praktiskt taget omöjligt. De tidsaspekter som behandlas när programmet exekveras är så korta att det inte heller går att mata ut alla förändringar i programmets tillstånd och variabler till omgivningen. Det är alltså omöjligt att fritt kunna utföra loggning för att se hur programmet beter sig. Vid testkörning av programmet saknas därför möjligheten att få fullständig information om systemet. Det som fås ut är korta, relativt osammanhängande brottstycken av det som händer i programmet under exekvering.

Det som bör påpekas i detta stadium är att detta arbete inte handlar om testning, det vill säga påvisandet av fel, utan behandlar lokalisering av ett fel som redan konstaterats finns.

3.1 Beskrivning av felsökningsförutsättningar

Detta arbete tar alltså fram riktlinjer för hur utvecklingsverktygen skall kunna ge ett så bra stöd som möjligt för processen att lokalisera fel i en programvara. Att utföra detta arbete på alla tänkbara programsystem och under alla tänkbara förutsättningar blir dock för omfattande för att arbetet skall vara praktiskt tillämpningsbart. Det behövs därför en tydligare precisering av den miljö som riktlinjerna skall fungera på.

Den utvecklingsmiljö som detta arbete fokuserar mot är den miljö som följer standarden IEC 1131-3. Således berörs fem programmeringsspråk: relälogik (RL), funktionsblocks-diagram (FBD), sekventiellt flödesschema (SFC), instruktionslista (IL) samt strukturerad text (ST).

De förutsättningar och antaganden som gäller det testade programmet kommer att enbart vara av sådan karaktär att de följer de kriterier som finns uppräknade i kapitel 2.4.

3.2 Beskrivning av riktlinjer

Syftet med de riktlinjer som detta arbete tar fram är, vilket tidigare påpekats, att om dessa implementeras i ett utvecklingsverktyg, så skall detta ge stöd åt felsökaren för att denne så enkelt som möjligt skall kunna lokalisera ett fel.

Riktlinjerna skall ta upp egenskaper hos utvecklingsverktyget. De skall inte inrikta sig mot felsökningsprocessen, det vill säga hur felsökning skall genomföras, de skall därför inte föreslå något tillvägagångssätt.

Strukturen på en riktlinje skall vara så att den uttalar en egenskap hos utvecklingsverktyget. Vidare skall det till denna egenskap finnas en motivering till varför egenskapen (riktlinjen) är bra. Denna motivering skall kunna härledas från en befintlig felsökningsmetod. Detta för att syftet och ursprunget för riktlinjen skall vara klart.

3.3 Problembegränsningar

Detta arbete inriktar sig främst mot felsökning som innefattar själva programkoden och då endast värdefel (kapitel 2.5). Arbetet tar alltså inte upp den felsökning som sker under design-, modellering- eller kravutvinningsfaserna. Inte heller tas tidsfel eller kombinationen av värdefel och tidsfel (kapitel 2.5) upp. Arbetet inriktar sig enbart på att klarlägga ett antal riktlinjer som skall underlätta lokalisering av värdefel (kapitel 2.5).

Detta examensarbete tar fram det stöd felsökningsverktyg kan erbjuda felsökning i programvaror i standard IEC 1131-3 vid användning av följande metoder (finns övergripligt beskrivna i kapitel 2.6):

- Brute force
- Felsökning genom induktion
- Felsökning genom backtracking.

Det finns många olika metoder att lokalisera fel. Varje programmerare har sin egen tolkning av varje metod. Det blir därför omöjligt att ta fram riktlinjer som tillgodoser varje programutvecklares önskemål. Dessa metoder är valda dels eftersom de är vanligt använda, varför ”brute force” tas med trots sina brister, och dels eftersom de behandlar olika aspekter och filosofier vid felsökning. Sannolikheten att de skall vara till nytta för en programmerare ökar då nästan oavsett vilken filosofi denne har för att lokalisera och identifiera fel.

Det finns två metoder (genomgångna i kapitel 2.6) vilka inte finns med bland de metoder som skall undersökas. Dessa är felsökning genom deduktion och felsökning genom testning. Felsökning genom deduktion bygger på samma felsökningsfilosofi som felsökning genom induktion, det vill säga att båda undersöker symtomen hos ett fel. Att erbjuda stöd för felsökning genom induktion ger därför stöd även för felsökning genom deduktion. Denna metod finns med i bakgrunden för att belysa att flera metoder kan bygga på samma filosofi (i detta fall på undersökning av felsymptom).

Felsökning genom testning erbjuder i sig flertalet metoder. Dessa är alldeles för många för att kunna behandlas under ett examensarbete. Felsökning genom testning finns med i bakgrunden för att visa att det finns ytterligare filosofier i hur fellokalisering kan gå till.

Ingen design eller implementation genomförs av verktyg som uppfyller kraven. Eftersom varje organisation eller företag har sina egna krav vad gäller utveckling av programvara så är det upp till organisationen eller företaget att avgöra vilka riktlinjer som skall följas och hur verktyget skall se ut med avseende på detta.

3.4 Uppgift

Detta examensarbete tar fram en mängd riktlinjer, enligt definition i kapitel 3.2, för det stöd ett utvecklingsverktyg kan ge vid felsökning enligt felsökningsmetoderna brute force, induktion och backtracking. Undersökning görs även för att se om riktlinjerna kan användas tillsammans med IEC 1131-3. De är dessutom anpassade enligt de förutsättningar som finns uppräknade i kapitel 2.4, vilket handlar om de kriterier för testvänlighet som den programvara följer som skall felsökas.

4 Val av lösningsmetoder

Problemet att ta fram felsökningsriktlinjer anpassade till standard IEC 1131-3 kan delas upp i två faser som var och en för sig inte är lika besvärliga.

1. Ta fram riktlinjer för verktygsstöd för de tre felsökningsmetoderna brute force, testning genom induktion och testning genom backtracking.
2. Undersöka om riktlinjerna kan användas tillsammans med standard IEC 1131-3.

4.1 Riktlinjer för verktygsstöd till felsökningsmetoder

Det första som måste utföras är att undersöka vilka riktlinjer som kan stödja de olika felsökningsmetoderna. Det finns ett antal metoder för att ta reda på dessa riktlinjer. Riktlinjerna skall vara av den typen att de talar om vad som kan stödja felsökningsprocessen. Några av de metoder som kan användas finns genomgångna i kapitlen 4.1.1 – 4.1.3.

4.1.1 Utforskning av felsökningsmetoder

En metod som kan användas för att ta fram riktlinjer går ut på att i detalj gå igenom varje felsökningsmetod. Varje felsökningsmetod har egna utmärkande drag och ställer krav på hur den skall genomföras. Det gäller att finna vilka förutsättningar felsökningsmetoden kräver, vilka begränsningar den har samt vilka aktiviteter den medför. Utifrån detta kan det identifieras vilka stöd felsökningsmetoden kan behöva. Dessa stöd undersöks och från dessa kan riktlinjer för vilka stöd felsökningsverktyget kan erbjuda för att underlätta felsökningsprocessen enligt metoden tas fram. Egenskapen som dessa riktlinjer skall ha är att de skall stödja felsökning enligt den metod de härleds ifrån. Vidare kan riktlinjerna ha karaktären att de automatiserar aktiviteter som annars ligger på den som felsöker att utföra.

Fördelen med denna typ av undersökning är att information kan inhämtas från många olika källor. Detta medför att de bedömningar som görs utifrån sammanställningen av dessa källor bygger på en teoretisk grund.

4.1.2 Undersökning av felsökningsverktyg

Det finns redan ett antal felsökningsverktyg som används. Dessa erbjuder stöd på olika sätt för felsökningsprocessen. Genom att undersöka vilka stöd dessa verktyg ger vid felsökning kan riktlinjer för verktygsstöd för att underlätta felsökningsprocessen tas fram.

Fördelen med detta sätt att utvinna riktlinjer är att resultatet redan har producerats av någon annan och riktlinjerna är bevisligen redan tagna i bruk hos verktygen. Arbetet blir i detta fall en sammanfattning av vad flera olika felsökningsverktyg erbjuder för stöd till felsökningsprocessen. Nackdelen är att motiveringen till varför riktlinjerna är bra med avseende på felsökningsmetoder måste efterkonstrueras och riskerar därmed byggas på felaktiga eller svaga resonemang. Vidare är det inte säkert att alla riktlinjer som kan stödja en felsökningsmetod hittas. Många bra riktlinjer kan komma att missas på grund av att det hittills inte finns något felsökningsverktyg som använder sig av

dem. Dessutom kan kvalitén på en del felsökningsstöd bli ifrågasatt. Det är inte säkert att ett hjälpmedel är bra bara för att det redan finns. Det kan vara så att det är enkelt att implementera i ett felsökningsverktyg, men vara onödigt och opraktiskt att använda.

4.1.3 Intervju

Ett sätt att få fram riktlinjer för hur olika felsökningsmetoder kan stödjas är att intervjua personer som använder sig av metoderna. Experter på en metod kan noggrant gå igenom vad för stöd som de använder samt vad för stöd de skulle behöva ha.

Innan intervjun genomförs skall metoderna gås igenom relativt noggrant. De frågor som ställs är av typen att de klarlägger riktlinjer. Det kan vara frågor om arbetsmetodik enligt felsökningsmetoden, vilka krav som ställs på felsökaren, felsökningsmiljön och programmet som skall felsökas. De personer som intervjuas bör tas från grupper med olika erfarenhetsgrader i användandet. De som inte har stor erfarenhet av felsökningsmetoderna kan ge riktlinjer som beaktar nybörjarnas behov. De mer erfarna metodanvändarna kan ge riktlinjer som beaktar olika finesser i felsökningsmetoderna.

Fördelen med denna typ av undersökning är att användarna är de som avgör vilka felsökningshjälpmedel som skall användas. Nackdelen är att den teoretiska förankringen blir svagare då riktlinjerna tas fram enbart genom muntliga åsikter snarare än genom förankring i teoretiska undersökningar i litteratur. Detta innebär att det är svårare att motivera vad för nytta riktlinjerna har för felsökningsmetoderna, då denna motivering ofta måste efterkonstrueras.

4.1.4 Val av metod

En nackdel med att undersöka befintliga felsökningsverktyg för att ta fram riktlinjer för verktygsstöd vid felsökning (se kapitel 4.1.2) är att det kan vara svårt att få tag i tillräckligt många felsökningsverktyg för att en bedömning skall kunna göras utifrån dessa. Vidare riskerar en utförlig undersökning av dessa att ta lång tid om de skall gås igenom noggrant.

En stor nackdel med att ta fram riktlinjer för verktygsstöd vid felsökning genom intervjuer (se kapitel 4.1.3) är att det kan vara svårt att få tag i så många användare som behövs för att kunna göra en objektiv bedömning av resultaten från intervjuerna. Vidare skulle intervjuerna ta väldigt lång tid att genomföra.

Med ovanstående som bakgrund väljs därför metoden (kapitel 4.1.1) att genomföra en teoretisk studie av de valda felsökningsmetoderna (se kapitel 3.3) och utifrån resultatet utvinna riktlinjer för vilka stöd ett felsökningsverktyg kan ge felsökningsprocessen.

4.2 Användning av riktlinjer tillsammans med IEC 1131-3

Även vid undersökning om huruvida det går att använda riktlinjerna tillsammans med IEC 1131-3 krävs en metodisk arbetsgång. Att ostrukturerat försöka använda riktlinjerna tillsammans med standarden kan innebära att standarden oavsiktligt bryts eller att riktlinjer inte används på grund av en onödig rädsla att bryta standarden. Risken finns också att riktlinjerna knyts till ett programmeringsspråk i standarden som i

verkligheten inte klarar av att stödja dem. Det finns även i detta steg ett antal metoder som kan användas.

4.2.1 Intervjuer av användare

Ett sätt att ta reda på om en riktlinje kan användas tillsammans med standard IEC 1131-3 är att intervjua personer som är insatta i och använder standarden. Dessa personer borde ha en kunskap om vilka riktlinjer som kan användas. Intervjun går till så att de intervjuande får säga vilka riktlinjer som passar in tillsammans med standarden. I och med intervjun kan även riktlinjernas ”nytta” i de olika språken i standarden utvärderas.

Fördelen med denna metod är att användarna själva kan avgöra vilka riktlinjer som är bra vid felsökning i programmeringsspråken. Vidare kan de ge en bedömning av hur stor hjälp de skulle ha av riktlinjen. En stor nackdel är att programmerarna redan är invanda med vissa sätt att felsöka. Deras bedömningar kommer att bli synnerligen subjektiva och riktlinjer riskerar att förkastas i onödan bara för att programmerarna inte ännu inser fördelarna med dem.

4.2.2 Teoretisk undersökning av standardens möjligheter att stödja riktlinjer

En tänkbar metod för att utröna om en riktlinje kan användas tillsammans med standarden är att undersöka standardens förutsättningar teoretiskt. En utförlig undersökning baserad på litteratur som beskriver standarden och språken i standarden eller som på annat sätt kan hjälpa till att ta reda på de förutsättningar som gäller genomförs. Utifrån detta kan det avgöras om en riktlinje kan användas i standarden eller inte.

Fördelen med denna metod är att det är standarden som talar om vilka riktlinjer som kan användas, risken för att bryta standarden är liten. En nackdel är att det kan vara svårt att utifrån litteraturstudier avgöra vad som ryms i standarden med avseende på felsökning. Vidare kan ingen bedömning göras utifrån studien hur användbar en riktlinje är vid felsökning i just ett bestämt språk.

4.2.3 Val av metod

Metoden som handlar om att bedöma riktlinjer utifrån intervjuer (se avsnitt 4.2.1) är attraktiv eftersom det som framkommer under intervjuerna i många fall bygger på erfarenheter. Den stora nackdelen är att de bedömningar som görs är synnerligen subjektiva. Därmed riskerar vissa riktlinjer att falla bort i onödan, medan andra riktlinjer accepteras trots att de bryter mot standarden.

Metoden som handlar om att genomföra en teoretisk undersökning (se avsnitt 4.2.2) är attraktiv eftersom den bygger på teoretiska studier av standardens förutsättningar snarare än användaråsikter. Därmed är risken mindre att den rådande standarden bryts. Nackdelen är att det inte går att göra en bra bedömning om hur nyttig en riktlinje är i språket.

Då det inte ingår i detta arbete att avgöra hur nyttig en riktlinje är, utan bara att avgöra om riktlinjerna kan användas tillsammans med standard IEC 1131-3, väljs metoden i avsnitt 4.2.2, om teoretisk undersökning av standardens förutsättningar. Detta på grund av att den, till skillnad från den andra metoden, ger resultat förankrade i stan-

darden och på de personer som undersökt standarden istället för på åsikter hos personer som bara använder standarden utan att undersöka vad den säger. Den teoretiska genomgången av standarden finns i kapitlen 2.2.1, vilket handlar om standarden i stort, och i 2.4 vilket handlar om testvänligheten hos en programvara som följer standarden.

5 Undersökning av felsökningsmetoder

Den valda metoden går ut på att ta fram riktlinjer för det verktygsstöd som kan finnas vid felsökning genom att i detalj utforska de felsökningsmetoder som valts ut (kapitel 3.3). När felsökningsmetoderna är så preciserade att det är lätt att se vilka aktiviteter som utförs och vad det är för information som används är det lätt att se vilka stöd felsökningsverktyget kan erbjuda för dessa aktiviteter. Detta kapitel syftar därför till att gå igenom felsökningsmetoderna så noggrant att riktlinjer lätt kan tas fram. Riktlinjerna i sig presenteras i kapitel 6.

5.1 Brute force

Felsökningsmetoden ”Brute force” är den felsökningsmetod som är mest använd trots att den är den minst effektiva (Pressman, 1997, sid. 528). Metoden går ut på att försöka hitta felet genom att bland annat analysera minnesdumpar, spåra programmets dynamiska egenskaper med hjälp av utskriftssatser eller använda en debugger.

Paul (1997) lägger fram tre sätt att undersöka den tillgängliga datamängden:

- Användning av minnesdump.
- Användning av utskriftssatser.
- Användning av felsökningsverktyg.

Aaby (1998) föreslår en arbetsgång för ”brute force”. Felsökningsmetoden genomförs genom att upprepa följande steg till dess att inga fler fel hittas (Aaby, 1998):

1. Kör programmet.
2. Om fel hittas genomför en eller flera av följande aktiviteter:
    - Undersök källkoden.
    - Lägg till utskriftssatser i programmet och skriv ut värden på intressanta variabler.
    - Lägg till påståenden till programmet (till exempel istället för y=x+1 skriv y=5, om fem är ett värde vars implikation på programmet felsökaren vill undersöka närmare).
    - Använd ett felsökningsverktyg.

De fel som är eftersökta är ofta felaktiga variabelvärden och felaktiga villkor. Det som kan konstateras vid jämförelse mellan Paul (1997) och Aaby (1998) är att båda tar upp tillägg av utskriftssatser och användning av felsökningsverktyg. Dessa är de båda mest använda metoderna. Paul tar också upp minnesdumpar som ett sätt att få fram information om programmet utan att påverka det under exekvering. Aaby tar upp ytterligare sätt att se vad som är fel genom att föreslå förändringar av programsatser och sökningar direkt i källkoden. Nedan beskrivs de olika arbets- och informationssätten i den ordning de räknas upp ovan.

En minnesdump är en ögonblicksbild av programmets tillstånd vid en given tidsenhet. Den består av att en del av det minne som är tilldelat till programmet sparas och gås igenom i jakten på fel. Vid felsökning i minnesdumpar eftersöks främst fel på variabler. Dock, enligt Myers (1979), har minnesdumparna de nackdelar att det kan vara

svårt att knyta minnesadresser till variabler. Eftersom en minnesdump är en del av minnet som lyfts ut måste stora mängder data behandlas vilken kanske inte har med felet att göra. Ytterligare en svårighet är att tidpunkten då minnesdumpen genereras kanske inte inträffar när felet uppstår, det är inte ens säkert att felet är närvarande i dumpen. Vidare finns det ingen metod för att systematiskt genomsöka en minnesdump

Det som behövs vid felsökning i minnesdumpar är:

- stöd för att knyta minnesadresser till variabler;
- stöd för att en minnesdump skall genereras på rätt ställe i rätt tid och
- stöd för att underlätta läsandet av en minnesdump.

Vid användandet av utskriftssatser läggs rader till i koden vilka har till uppgift att tala om i vilket tillstånd programmet befinner sig i för närvarande. Det kan till exempel röra sig om utskrifter som talar om vart i koden programmet exekverar i för närvarande eller talar om värdet på variabler. Det är i denna variant av brute force lättare att se programmets dynamiska uppträdande, det vill säga hur det beter sig under gång, än i felsökning genom minnesdumpar. Nackdelarna detta felsökningssätt medför är att det går fortfarande inte att vara säker på att det är rätt data som visas (Myers, 1979). Vilka variabler som skall skrivas riskerar att bygga på egna gissningar. Vidare riskerar det att vara stora mängder data som måste analyseras. Utskriftssatserna kan också maskera, påverka eller rent av introducera nya fel. Det kan också hända att de påverkar synkroniseringar i systemet, vilket är allvarligt då PLC-system är realtidssystem. Det som också måste bäras i åtanke är att eftersom det rör sig om realtidssystem med begränsade kommunikationsmöjligheter (se kapitel 2.1 om målmiljöer) är det inte säkert att det går att presentera all önskad information utan att allvarligt påverka systemet.

Utifrån detta kan det konstateras att det behövs:

- stöd för att komma åt variabelvärden och programtillstånd under exekvering;
- stöd för tekniker att minska risken att visning av variabelvärden påverkar programmet genom att maskera, påverka eller introducera fel eller påverka synkroniseringar och
- stöd för att underlätta kommunikation mellan målmiljö och utvecklingsmiljö under exekvering.

Felsökningsverktyg erbjuder ofta stora hjälpmedel vad gäller felsökning. De erbjuder många gånger möjligheter att avbryta exekveringen i rätt tid och kan spåra variabelförändringar och vart i koden programmet exekverar. Det är dock svårt att använda dessa i PLC-system. Då sådana system interagerar så kraftigt med omgivningen är det väldigt riskabelt att avbryta exekveringen av programmen, de kan till exempel vara styrenheter till tunga maskiner. Vidare har PLC-systemen begränsade kommunikationsmöjligheter (se kapitel 2.1 om målmiljön) varför information om systemets tillstånd kan vara svår att få tag i. Vad som behövs är alltså:

- möjlighet att kunna avbryta exekveringen av ett program utan att omgivningen kommer till skada;
- möjlighet att kunna få ut information om systemets tillstånd under exekvering.

Att undersöka källkoden kräver att den som felsöker vet ungefär vart i programmet felet är. Detta sätt är inte att rekommendera vid stora program. De stora nackdelarna med denna metod är att det gäller att veta ungefär vart felet är och vilket tillstånd pro-

grammet var i när felet inträffade. Vidare är det svårt att upptäcka fel som inte ger sig till känna i den del av programmet de är, utan har propagerats till andra delar. Det gäller därför att ha en så god uppfattning om vad felet är att slutsatser kan dras genom att bara titta på källkoden. Det behövs därför:

- stöd för att ta reda på vart i koden programmet exekverar vid fel och
- stöd för att få reda på variabelvärden under och efter exekvering.

Att lägga till påståenden i programmet är ett sätt att själv kunna styra exekveringen av programmet. Filosofin går ut på att leta reda på felet genom att introducera ett eget fel som felsökaren har kontroll över och som hjälper till att producera de programtillstånd felsökaren vill uppnå. Dessa fel kan till exempel vara så att de alltid gör ett villkor sant, att en funktion alltid ger samma resultat, med mera. Fördelen med detta sätt är att det hjälper felsökaren att styra programexekveringen in i möjligheter som är intressanta att undersöka. Den stora nackdelen är att det genomförs programändringar i områden som kanske är felfria. Detta kan ha sidoeffekter och påverka andra delar av programmet på ett oväntat sätt och därmed riskera att introducera nya följdfel. Om inte felsökaren är väldigt insatt i programmet är denna metod inte att rekommendera. Det behövs för denna metod :

- stöd för att ta reda på när variabler senast sattes och vilket värde de har;
- stöd för att se vad variabler påverkar, det vill säga vad de har för sidoeffekter;
- stöd för att se om villkor är sanna eller falska och
- stöd för att kunna se resultatet från funktioner.

5.2 Induktion

Felsökning med induktionsmetoden går ut på att undersöka de symptom som felet genererar för att på så sätt utröna vad och vart felet kan vara. Detta är en felsökningsmetod där den största arbetsbördan inte nödvändigtvis kräver en dator. Metoden litar mer på felsökarens egen tankeverksamhet än på de verktyg som kan finnas till hands. Detta innebär en fördel för tillverkare av till exempel PLC-system, då det ofta är svårt att få tillgång till ordentliga testmiljöer och testmöjligheter. Tillverkarna kan utföra sina testserier och därefter på egen hand söka efter felen utan att belasta omgivningen, se inledningen i problempreciseringen (se inledning i kapitel 3).

Sharp (1993) delar in arbetsgången i två stora delar: insamlandet av information om felet och användandet av logiskt tänkande och intuition för att komma fram till vad det är som orsakar felet. Liksom hos Paul (1997) har Sharp även en finare indelning som består av:

1. insamling av data,
2. organisation av data,
3. uppställning av hypotes samt
4. bevisning av hypotes.

Insamlandet av data i ett PLC-system, eller i nästan vilket realtidssystem som helst, är svårt. Realistisk testning tillsammans med ett bra felsökningsverktyg är ingen självklarhet när stora maskiner skall styras. Vidare, vilket nämnts tidigare, är kommunikationsmöjligheterna mot omvärlden samt storleken på minnesutrymme begränsade. För

dessa situationer föreslår Sharp (1993) olika typer av loggningsmekanismer vilka är snåla vad gäller minnesutrymme och dessutom undviker att blanda in kommunikationen med omvärlden till stora delar. Sharp föreslår en loggningsmekanism som bara lagrar nyckelvärden och vilka händelser som ägt rum under exekveringen. Vidare föreslår han en mekanism som möjliggör av- och påsättandet av loggningen. Om felet inträffar sällan föreslår Sharp att loggning skall ske i en cirkulärt länkad lista i minnet hos målmiljön och låta listan bara innehålla den senaste informationen.

För felsökning genom induktion behövs alltså:

- stöd för åtkomst av variabelvärden;
- stöd för åtkomst av vilka delar av ett program som exekverats/exekveras;
- stöd för loggning av programkörningar;
- stöd för av- och påsättning av loggningsmekanismer och
- stöd för åtkomst av lagrad data i målmiljön.

För att kunna dra slutsatser på ett bra sätt måste den information som samlats in organiseras på ett bra sätt (Paul, 1997 och Sharp, 1993). Om informationen är välorganiserad är det lättare att se mönster och samband i den och därmed ta fram hypoteser. Sharp föreslår att följande skall dokumenteras:

- vilka symptom som uppträder för felet;
- vad programmet gör rätt;
- vart symptomen uppträder;
- vart symptom ej uppträder;
- när symptomen uppträder och
- när programmet beter sig korrekt.

Det som skall föras ned är alltså i korthet: vad, vart, när. De tillfällen där programmet beter sig korrekt är lika viktiga att dokumentera som när programmet beter sig felaktigt. Även korrekthet hos ett program hjälper till att visa vad som är fel.

För denna del av felsökningen kan det vara bra med:

- stöd för organiserandet av information samt
- stöd för att se samband i informationen.

Vad gäller formulerandet av en hypotes samt bevisandet av denna är syftet med denna felsökningsmetod att undvika att blanda in felsökningsverktyg så mycket som möjligt. Studier (Pan och Spafford, 1992) visar att hypotesformulerandet och –bevisandet i realiteten går ut på följande:

1. Att ta reda på vilka programuttryck (programrader) som är inblandade i fel.
2. Att ta fram de programuttryck som kan innehålla ett fel.
3. Att ta fram en hypotes om vad som är fel med avseende på variabler och lokalisering.
4. Att återskapa ett programtillstånd till ett specifikt uttryck för att verifiera felet.

Sharp (1993) föreslår i samband med hypotesformulerandet och –bevisandet att felsökaren bör fundera över vad det är som motsäger hypotesen. Felsökaren bör försöka hitta fall som motvisar hypotesen.

För att till fullo undersöka de fyra punkterna Pan och Spafford (1992) redovisar krävs studier av en sådan omfattning att dessa inte blir praktiskt möjliga inom tidsramarna för detta arbete. Däremot går det att konstatera att det behövs:

- stöd för att ta fram vilka programuttryck som är inblandade i ett fel;
- stöd för att ta fram de programuttryck som kan innehålla fel samt
- stöd för att kunna verifiera en hypotes.

Vad gäller hypotesverifiering gäller det, enligt Sharp (1993), att se om hypotesen kan förklara alla symptom samt om hypotesen gör några nya förutsägelser om felsymptom. Ett bra sätt att visa om hypotesen är korrekt är att försöka påvisa förutsägelserna.

5.3 Backtracking

När ett fel uppstår i ett program är det inte alltid så att det förutom källkoden finns mer information att tillgå än eventuell indata och det felaktiga resultatet. I stora program är det otympligt att manuellt börja analysera genom till exempel brute force och försöka finna felet på detta sätt. Agrawal, DeMillo och Spafford (1990) anser att ett logiskt sätt att felsöka med den typen av förutsättningar är att tänka baklänges. Den som felsöker utgår från det stället där felet ger sig till känna och försöker exekvera programmet baklänges uttryck för uttryck till dess felet har identifierats. Det är inte nödvändigt så att varje programrad skall gås igenom, detta kan i stora program vara en mycket tidsödande och svår uppgift. Ibland räcker det med att stega bakåt modulvis (Agrawal, DeMillo och Spafford, 1990). Det senare kräver dock med största sannolikhet att ett felsökningsverktyg med möjligheter att sätta brytpunkter används. Att notera är att Paul (1997) och Pressman (1997) anser att denna metod bäst fungerar på små program på grund av att det i större program riskerar att bli allt för många möjliga vägar bakåt, då antalet genomgångna programrader ökar, för att det skall vara möjligt att behandla.

Backtracking är en felsökningsmetod som till största delen tar hjälp av felsökarens tankekraft. Då det i somliga fall kan vara komplicerat att ”exekvera baklänges i huvudet” (till exempel i avancerade matematiska uttryck) behöver felsökaren många gånger stöd från felsökningsverktyg (Agrawal, DeMillo och Spafford, 1990). Främst utnyttjas då de möjligheter dessa kan erbjuda i form av att sätta brytpunkter, exekvera stegvis och ibland deras förmåga att lagra tidigare programtillstånd (även om antalet programtillstånd som lagras för det mesta inte är särskilt imponerande då detta kräver mycket minnesutrymme).

Vad gäller felsökningsverktygens möjligheter att verka i ett PLC-system konstateras det i kapitel 5.1 om felsökning med ”brute force” att dessa är starkt begränsade. Det går inte att avbryta exekvering i målmiljön utan att riskera konsekvenser i form av materiella tillbud (till exempel stoppandet av ett system som styr tunga maskiner eller system av maskiner). Vidare finns begränsningar i hur mycket information som kan föras mellan målmiljön och utvecklingsmiljön då kommunikationskanalerna inte alltid är så snabba som de behöver vara.

Det går alltså att konstatera att stöd bör finnas för att underlätta verksamheten att stega baklänges i ett program vid läsning av källkoden, både vad gäller programrad för programrad och modulvis. Vidare bör samma förutsättningar gälla som vid användningen av felsökningsverktyg enligt ”brute force” (kapitel 5.1).

Det behövs alltså:

- stöd för att spåra ett misstänkt fel bakåt i programkoden;
- stöd för att spåra ett misstänkt fel mellan moduler samt
- stöd för att stega sig igenom programmet baklänges.

Då stegning av ett program baklänges många gånger kräver felsökningsverktyg, till exempel vid stegning mellan moduler och vid mera komplicerade operationer, behövs även stöd för (tagna från kapitel 5.1 om felsökningsverktyg):

- att kunna avbryta exekveringen av ett program utan att omgivningen kommer till skada samt
- att kunna få ut information om systemets tillstånd under exekvering.

Eftersom även stegning baklänges med hjälp av tankekraft kan vara en ansträngande process behövs:

- stöd för att få reda på tidigare programtillstånd och variabelvärden.

6 Felsökningsstöd i IEC 1131-3

I kapitel 5 gås de tre felsökningsmetoderna brute force, induktion och backtracking igenom. Kapitlen 2.2.1 och 2.4 tar upp de begränsningar och förutsättningar som finns hos standard IEC 1131-3. Det är nu dags att utnyttja den kunskap som inhämtats i dessa kapitel och ta fram riktlinjer för det verktygsstöd en felsökare kan behöva vid felsökning i ett PLC-system enligt standard IEC 1131-3.

6.1 Lista över identifierade stöd till felsökningsmetoder

Kapitel 5 går, vilket nämns ovan, igenom de tre felsökningsmetoderna brute force, induktion och backtracking. För var och en av dessa metoder tas det fram ett antal punkter över vad som kan behövas för stöd vid användning av metoderna. Dessa punkter ligger till grund för de riktlinjer som tas fram. Riktlinjerna syftar till att på olika sätt ge det stöd som kan konstateras vara till nytta. Nedan följer listor på vad de olika felsökningslistorna kan behöva för stöd.

6.1.1 Brute force

Brute force gås igenom noggrant i kapitel 5.1. Genom denna undersökning kan det konstateras att det finns ett antal punkter där stöd från ett felsökningsverktyg kan komma att behövas. Nedan följer en sammanställning av de identifierade punkterna. Det behövs stöd för:

S1.1 att knyta minnesadresser till variabler;

S1.2 att en minnesdump skall genereras på rätt ställe i rätt tid;

S1.3 att underlätta läsandet av en minnesdump;

S1.4 att komma åt variabelvärden och programtillstånd under exekvering;

S1.5 tekniker att minska risken att visning av variabelvärden påverkar programmet genom att maskera, påverka eller introducera fel eller påverka synkroniseringar;

S1.6 att underlätta kommunikation mellan målmiljö och utvecklingsmiljö under exekvering;

S1.7 möjligheten att kunna avbryta exekveringen av ett program utan att omgivningen kommer till skada;

S1.8 möjlighet att kunna få ut information om systemets tillstånd under exekvering;

S1.9 att ta reda på vart i koden programmet exekverar vid fel;

S1.10 att få reda på variabelvärden under och efter exekvering;

S1.11 att ta reda på när variabler senast sattes och vilket värde de har;

S1.12 att se vad variabler påverkar, vad de har för sidoeffekter;

S1.13 att se om villkor är sanna eller falska;

S1.14 att kunna se resultatet från funktioner.

6.1.2 Induktion

Induktion gås igenom i kapitel 5.2. Det identifieras där ett antal punkter där ett felsökningsverktyg kan erbjuda stöd som hjälper felsökaren i sin uppgift. Dessa gäller stöd för:

S2.1 åtkomst av variabelvärden;

S2.2 åtkomst av vilka delar av ett program som exekverats/exekveras;

S2.3 loggning av programkörningar;

S2.4 av- och påsättning av loggningsmekanismer;

S2.5 åtkomst av lagrad data i målmiljön;

S2.6 organiserandet av information;

S2.7 att se samband i information;

S2.8 att ta fram vilka programuttryck som är inblandade i ett fel;

S2.9 att ta fram de programuttryck som kan innehålla fel;

S2.10 att kunna verifiera en hypotes.

6.1.3 Backtracking

Backtracking gås igenom i kapitel 5.3. Det identifieras där ett antal punkter där ett felsökningsverktyg kan erbjuda stöd för att underlätta felsökning med denna metod. Dessa punkter gäller stöd för:

S3.1 att spåra ett misstänkt fel bakåt i programkoden;

S3.2 att spåra ett misstänkt fel mellan moduler;

S3.3 att stega sig igenom programmet baklänges;

S3.4 möjligheten att kunna avbryta exekveringen av ett program utan att omgivningen kommer till skada;

S3.5 möjlighet att kunna få ut information om systemets tillstånd under exekvering;

S3.6 att få reda på tidigare programtillstånd och variabelvärden.

6.2 Riktlinjer för felsökning med brute force

Det finns ett antal sätt på vilka felsökning genom användandet av felsökningsmetoden brute force kan stödjas av utvecklingsmiljön och målmiljön. I kapitel 6.1.1 redovisas 14 punkter (S1.1 – S1.14) där ett felsökningsverktyg kan underlätta felsökningen. De riktlinjer som tas fram nedan är av den karaktären att de kan erbjuda eller hjälpa till att erbjuda ett stöd för dessa punkter.

R1.1 Utvecklingsmiljön skall kunna ta reda på en variabels position i minnet på målmiljön och presentera denna position för felsökaren.

Denna riktlinje erbjuder en lösning på S1.1. Denna punkt handlar om att det behövs stöd för att kunna knyta en variabel till en minnesadress. Detta är nödvändigt då felsökaren försöker identifiera ett fels position genom att leta igenom en minnesdump med hjälp av felsökningsmetoden brute force. En stor svårighet med att leta efter spår

av fel i en minnesdump är just att det är svårt att knyta en variabel till en bestämd minnesadress. Riktlinjen ger felsökaren en möjlighet att få reda på vart en eftersökt variabel är placerad i minnet och därmed lättare kunna hitta variabelns värde.

R1.2 Utvecklingsmiljön skall på egen hand leta reda på en variabels position i en minnesdump och kunna presentera variabelns värde för felsökaren.

Denna riktlinje erbjuder en lösning på S1.1, vilken handlar om att det behövs stöd för att knyta en variabel till en viss minnesadress. Detta för att underlätta felsökandet vid letandet av fel i en minnesdump genom felsökningsmetoden brute force. En stor svårighet med att leta efter fel i en minnesdump är svårigheten att få reda på vart i minnet en viss variabel befinner sig. Denna svårighet undviks nu genom att utvecklingsmiljön själv identifierar variabelns position, tar reda på variabelns värde och kan presentera detta för felsökaren. Riktlinjen ger en möjlighet för utvecklingsmiljön att kunna presentera variabelvärden på ett bättre och mer lättläst sätt än vad en vanlig minnesdump kan göra.

R1.3 En minnesdump skall produceras när exekvering på målmiljön avbryts på ett annat sätt än att programmet är avslutat.

Denna riktlinje erbjuder en dellösning på S1.2. Denna punkt handlar om att en minnesdump skall genereras på rätt ställe och vid rätt tidpunkt för att ett fel så enkelt som möjligt skall upptäckas vid felsökning i minnesdump med felsökningsmetoden brute force. En svårighet med att felsöka i en minnesdump är att denna inte alltid är producerad på det ställe där felet lättast kan hittas. Riktlinjen innebär att en minnesdump produceras till exempel när programmet har utfört ett så allvarligt fel att det inte kan fortsätta eller när ett användaravbrott sker. Det senare exemplet möjliggör för felsökaren att själv kunna avgöra när en minnesdump skall produceras. Riktlinjen är felorienterad emedan den bara producerar en minnesdump då ett oväntat avbrott uppträder. Sannolikheten att detta sker vid ett fel är relativt stor jämfört med vid ett ställe där programmet inte är fel.

R1.4 En minnesdump skall produceras då exekvering avslutas.

Denna riktlinje är en mer generell variant av riktlinje R1.3, då den innebär att en minnesdump alltid genereras när ett program avslutas. Riktlinjen är till för att lösa S1.2. Denna punkt handlar om att en minnesdump skall genereras så att ett fel skall hittas så enkelt som möjligt. Detta är till stor hjälp vid felsökning i en minnesdump genom brute force. I detta fallet är det enkelt för användaren att få tillgång till den status programmet har när det avslutas oavsett hur det avslutas. Riktlinjen innebär att utvecklingsmiljön automatiskt tar fram en minnesdump att felsöka i då programmet avslutas utan att detta explicit behöver göras av felsökaren.

R1.5 Felsökaren skall kunna ta en minneskopia när som helst.

Denna riktlinje erbjuder en möjlig lösning för S1.2. Denna punkt handlar om att en minnesdump skall genereras vid en sådan tidpunkt och sådan plats att felsökning i minnesdumpen genom brute force är så enkel som möjligt. I detta fallet kan användaren få tillgång till aktuell information om programmets status, förhoppningsvis utan att programmets exekvering behöver avslutas. Detta innebär att felsökaren kan be om en minneskopia och ändå fortsätta exekvera programmet. Nackdelen med denna riktlinje är att eftersom kommunikationen mellan utvecklingsmiljö och målmiljö är begränsad i hastighet (se kapitel 3.0) innebär detta att information kan komma att uppdateras medan minneskopian fortfarande tas. Detta innebär att minneskopian kan innehålla rester av flera olika programtillstånd samtidigt. Detta gör det svårare att se

samband i informationen samt att det bryter mot definitionen av minnesdump som en ögonblicksbild av programmets tillstånd.

R1.6 Utvecklingsmiljön skall kunna analysera en minnesdump och presentera en lista med variabler och tillhörande värden hämtade från minnesdumpen.

Denna riktlinje erbjuder en lösning för S1.3. Punkten handlar om att presentera informationen från en minnesdump, vid felsökning med brute force, på ett sådant sätt att informationen blir lättläst och lättförståelig. Då en av de stora nackdelarna med att använda en minnesdump för felsökning är att informationen är väldigt svår att förstå innebär en lista där variabler står redovisade tillsammans med respektive värden en avsevärd förbättring jämfört med att läsa ett direkt utdrag ur minnet. Riktlinjen förutsätter givetvis att det finns en minnesdump att analysera. Detta innebär att en av riktlinjerna R1.3, R1.4 och R1.5 eller varianter av dem redan bör följas.

R1.7 Utvecklingsmiljön skall kunna analysera en minnesdump och presentera det värde minnesdumpen har på en variabel som är markerad i källkoden.

Denna riktlinje erbjuder en lösning för S1.3. Denna punkt handlar om att presentera informationen från en minnesdump, vid felsökning med brute force, så att den blir mer lättläst och enklare att förstå sig på. En av nackdelarna med att använda en minnesdump är att informationen är väldigt svårbegriplig. Det gäller dels att knyta variabler till minnesadresser och därefter att kunna tolka det som står på denna adress. Denna riktlinje lägger över detta arbete på utvecklingsmiljön. Vidare presenteras denna information genom att felsökaren på något sätt markerar den variabel vars värde är intressant. Detta innebär att felsökaren själv kan avgöra vilka värden som är intressanta att titta på och kommer undan all extra information som annars finns i en vanlig minnesdump. Vidare knyts denna information till intressanta delar av programmet genom att markering av en variabel kan ske direkt i koden. Denna riktlinje förutsätter att en av riktlinjerna R1.3, R1.4 och R1.5 eller varianter av dem redan följs, det vill säga att tillgång till en minnesdump finns.

R1.8 Uppdatering av variablers värden på målmiljön skall skickas till utvecklingsmiljön under exekveringens gång.

Denna riktlinje erbjuder en dellösning för S1.4. Denna punkt handlar om att underlätta sökandet efter fel med brute force genom att analysera ett programs dynamiska beteende. Det är lättare att hitta ett fel genom att studera ett programs förändringar över tiden än att titta på bara ett programtillstånd (Paul, 1997). Genom att ha möjlighet att få kontinuerliga uppdateringar om variablers värden underlättas felsökandet genom att oväntade förändringar lättare upptäcks. Variablernas värden kan till exempel presenteras i en lista där de kombineras med variabelnamnen. En fördel är att om informationen om variabelvärden når utvecklingsmiljön så kan de även lagras och en variabels historik följas. En svårighet med denna riktlinje i PLC-system är att kommunikationshastighet mellan målmiljö och utvecklingsmiljö är begränsad. Detta innebär att det inte är säkert att all information hinner överföras. En möjlig lösning är att felsökaren själv kan välja vilka variabler som skall följas. Kan felsökaren koncentrera sitt sökande till ett litet, begränsat antal variabler är det större sannolikhet att de intressanta förändringarna kan överföras till utvecklingsmiljön och registreras utan att detta innebär att deadlines hos målmiljön försätts i fara att brytas.

Riktlinjen ger även en lösning till S1.8. Denna punkt handlar om att det skall gå att komma åt systemets tillstånd under exekvering för att underlätta felsökandet med brute force.

Riktlinjen erbjuder dessutom en lösning för S1.10. Denna punkt handlar om möjligheten att få reda på variablers värden under och efter exekvering. Detta för att lättare kunna spåra fel med felsökning genom brute force. I och med att uppdateringar av variablers värden ständigt skickas till utvecklingsmiljön är det möjligt att följa variablerna genom programmets gång.

R1.9 Vid moduluppdelade program skall information om vilken modul som exekveras för tillfället skickas till utvecklingsmiljön.

Denna riktlinje erbjuder en dellösning för S1.4. Denna punkt handlar om att underlätta sökandet efter fel med brute force genom att analysera ett programs dynamiska beteende. Om ett fel manifesterar sig på ett sådant sätt att det kan ses från omvärlden innebär denna riktlinje att felsökaren har en möjlighet att se i vilken modul programmet exekverar i och därmed kunna begränsa sökandet efter felet. Det går att tänka sig andra varianter på denna riktlinje. Istället för att överföra information om aktuell modul kan en mer preciserad information skickas, till exempel den funktion som för närvarande exekveras. Vidare kan information om vilka selektioner som utförts skickas. All denna information hjälper felsökaren att begränsa sökrymden efter felet. Den information som skickas till utvecklingsmiljön kan lagras och senare användas i felsökningen. Nackdelen med PLC-system är dock att kommunikation mellan målmiljö och utvecklingsmiljö är starkt begränsad i hastighet. Detta innebär att viktig information inte nödvändigtvis hinner skickas över.

Denna riktlinje erbjuder även en lösning för S1.9. Denna punkt handlar om att kunna få reda på i vilken del av programmet ett fel finns. Om detta fel manifesterar sig på ett sätt att det går att se när det inträffar ger denna riktlinje en möjlighet att se vilken modul som för närvarande exekveras. En förutsättning är dock att programmet befinner sig i modulen så länge att det är möjligt för felsökaren att se detta.

R1.10 Information om variablers värden under exekvering på målmiljön skall kunna vara åtkomliga från utvecklingsmiljön utan att detta explicit måste anges av felsökaren genom ändringar i programkoden.

Denna riktlinje erbjuder en lösning på S1.5. Denna punkt handlar om undvikandet att introducera fel i programkod eller synkroniseringsfel genom att felsökaren måste ta reda på intressanta variabelvärden, vid felsökning med brute force, genom att lägga till utskriftssatser i programmet. En av riskerna med att lägga till kod i ett färdigt program är att den riskerar att introducera fel i ett program. Det är opraktiskt att riskera att införa nya fel i ett program för att leta efter gamla. Riktlinjerna R1.8 och R1.9 kan realiseras genom att felsökaren själv i programkoden får lägga till utskriftssatser. Denna riktlinje förhindrar detta genom att konstatera att risken för fel minskar om information kan överföras utan att den ursprungliga programkoden förändras. Låt kommunikation av denna karaktär skötas av utvecklingsmiljön och målmiljön med så lite inblandning av felsökaren som möjligt.

R1.11 Låt så mycket av kommunikationen mellan målmiljön och utvecklingsmiljön som möjligt skötas utan inblandning från användare.

Denna riktlinje erbjuder en möjlig lösning på S1.6. Denna punkt handlar om att störa målmiljön så lite som möjligt under exekvering. Vid felsökning med brute force är det fördelaktigt att kontinuerligt få ut information från systemet medan det körs. Detta för att kunna analysera hur programmets dynamiska beteende är. Ett problem med PLC-system är dock att de är realtidssystem. Det finns deadlines som måste hållas. Varje extra uppgift i systemet, till exempel uppgiften att kontinuerligt ge utvecklingsmiljön information, innebär att det blir allt svårare att hålla en deadline – tidsmarginalerna

minskar. Ett sätt att snabba upp kommunikationen är att låta så mycket av kontrollen som möjligt skötas automatiskt, utan att blanda in användaren mer än absolut nödvändigt.

R1.12 Möjlighet att använda en simulator för att simulera den miljö programmet skall styra samt simulera målmiljö.

Denna riktlinje erbjuder en lösning för S1.7. Denna punkt handlar om att det skall gå att kunna avbryta exekveringar utan att omgivningen kommer till skada, vilket ofta är risken med PLC-system. För att kunna felsöka med brute force är det i vissa fall nödvändigt att felsökaren själv kan starta och stoppa exekvering av programmet. Detta för att på ett effektivt sätt undersöka ett programs dynamiska egenskaper. Det finns flera fördelar med att använda en simulator. All information i systemet kan loggas. Det kan erbjudas möjligheter att stega sig igenom programmet. Tidskrav kan testas. I stort sett ger en simulator möjlighet att kunna utföra uttömmande tester så att risken för att programmet inte fungerar när det introduceras i målmiljön är väldigt liten i förhållande till om det direkt introduceras. En stor nackdel med en simulator för PLC-tillverkarna är att de maskiner eller serier av maskiner som skall styras för det mesta ser olika ut. Att fullständigt simulera ett helt PLC-system innebär i många fall en extrakostnad i både tid och pengar för att tillverkarna skall anse sig ha råd. I simulatorn kan det röra sig om tusentals komponenter som måste simuleras och koordineras (se kapitel 2.1 om målmiljön).

R1.13 Möjlighet att simulera delar av händelseförlopp.

Denna riktlinje erbjuder en lösning för S1.7. Denna punkt handlar om möjligheten att kunna avbryta ett programs exekvering utan att omgivningen kommer till skada. I riktlinje R1.12 föreslås detta att lösas med en simulator. Dock konstateras det i det vidare resonemanget kring riktlinjen att det är väldigt svårt att simulera ett helt PLC-system. Det kan finnas alldeles för många komponenter som måste koordineras. En möjlig förenkling på en simulator är att erbjuda en möjlighet att simulera delar av PLC-systemet. Detta innebär att simuleringar är möjliga att utföra. Dock är det inte lika sannolikt att dessa simuleringar är lika fullständiga. Denna variant på simulator erbjuder bara delar av ett program att simuleras. Det är inte alltid möjligt att göra denna förenkling.

R1.14 Felmeddelanden skall vara läsbara.

Denna riktlinje kan, rätt använd, erbjuda en lösning för S1.9. Denna punkt handlar om att felsökaren skall kunna få reda på i vilken del av ett program ett fel inträffar. Detta underlättar mycket då felsökningsrymden förhoppningsvis kan minskas avsevärt. Information som skall finnas i felmeddelandet skall tydligt tala om vilket fel som har inträffat. Vidare, för att ytterligare hjälpa till med felsökningsprocessen, kan information presenteras om i vilken del av programmet som felet upptäcktes. Det kan till exempel anges vilken programmodul eller funktion som gav felmeddelandet. Detta innebär förvisso inte att felet kan begränsas till just den modul som pekats ut. Det kan hända att felet är ett följdfel från en tidigare modul. Dock kan stora delar av programmet förhoppningsvis uteslutas och om tillgång till vilka moduler som tidigare exekverats finns kan felet ytterligare ringas in.

R1.15 Det skall gå att se vart en variabel senast sattes och vilket värde den har.

Denna riktlinje ger en lösning till S1.11. Denna punkt konstaterar att det behövs stöd för när en variabel senast sattes och vilket värde den har. Detta underlättar kraftigt felsökning genom brute force. En svårighet med denna riktlinje då det gäller PLC-

system är dock att kommunikationshastigheten mellan mål- och utvecklingsmiljön är begränsad (se kapitel 3.0). Detta innebär att det finns en gräns i hur mycket information som kan lämna målmiljön. Denna gräns riskerar att snabbt nås då information om alla variabler skall lämnas ut. En möjlighet är att felsökaren specificerar en eller ett par variabler som skall undersökas. Vidare kan det vara svårt att få reda på vart variabeln senast ändrades, särskilt om variabeln ofta uppdateras. En loggningsmöjlighet hos utvecklingsmiljön kan därför vara till stor hjälp.

R1.16 Visa i vilka programuttryck en specificerad/markerad variabel förekommer.

Denna riktlinje löser S1.12. Denna punkt handlar om att det är till stor hjälp att kunna se var och vad i ett program en variabel påverkar. På detta sätt är det relativt lätt att ta fram kodrader som eventuellt kan vara inblandade i ett fel eller som orsakar felet. En variabel som är inblandad i ett fel kan spåras bakåt för att se vilka programrader som kan innehålla felet och spåras framåt för att undersöka följdfel. Ett exempel på hur riktlinjen kan implementeras i utvecklingsmiljön är att varje rad som innehåller den variabel felsökaren markerat blir markerad så att felsökaren kan se på vilka rader den förekommer. När riktlinjen används i sin vidaste form kan en global variabel spåras genom hela programmet. Riktlinjen kan också ge möjlighet att spåra variabler mellan funktioner och moduler. Dock krävs det då att utvecklingsmiljön även vet varifrån modulerna och funktionerna anropas.

R1.17 Om tillgång till variabelvärden finns skall det gå att från utvecklingsmiljön ta reda på om villkor är sanna eller falska.

Denna riktlinje löser S1.13. Denna punkt handlar om att felsökning genom brute force underlättas mycket om felsökaren vet vilka delar av programkoden som exekverats. Detta ges av att få reda på om de villkor som kan ge olika programflöden är sanna eller falska. Om utvecklingsmiljön har tillgång till de variabelvärden som styr om ett villkor är sant eller falskt är det således till stor hjälp om denne själv räknar ut om ett villkor är sant eller falskt. Nackdelen är att detta förutsätter att de variabelvärden som programmet räknar med är samma som de som målmiljön använder. Då det inte är säkert att de korrekta variabelvärdena hinner föras över till utvecklingsmiljön är det inte säkert att beräkningarna i utvecklingsmiljön ger samma resultat som de i målmiljön.

R1.18 Målmiljön talar under exekveringens gång om för utvecklingsmiljön om ett villkor är sant eller falskt.

Denna riktlinje löser S1.13. Denna punkt handlar om att felsökningen med brute force underlättas om felsökaren vet vilka delar av koden som exekverats. Detta ges av att få reda på om de villkor som kan ge olika exekveringsvägar genom programmet är sanna eller falska. I denna riktlinje talar målmiljön om för utvecklingsmiljön om villkor är sanna eller falska. Vidare går det att tänka sig att målmiljön även skickar värdet på villkoret, om detta är av en sådan typ att de avgörs genom en aritmetisk uträkning. En fördel med denna riktlinje är att det blir väldigt lite information som skall skickas mellan målmiljön och utvecklingsmiljön. Det ligger sedan på utvecklingsmiljön att presentera detta på ett bra sätt. Olika alternativ kan vara att om ett villkor markeras så visas värdet på detta villkor, de delar av programmet som exekverats blir markerade eller en lista visas där villkor och deras värden visas.

R1.19 Felsökaren skall kunna få reda på resultatet från funktioner under och efter exekvering.

Denna riktlinje löser S1.14. Denna punkt handlar om att felsökaren skall kunna få reda på resultatet på funktioner som använts i programmet. Detta görs för att lättare kunna spåra fel vid felsökning med brute force. Detta förutsätter dock att programmet är väl moduluppdelat, så att inte det finns massor av globala variabler som ändras eller att användningen av minnesreferenser vid parameteröverföring är omfattande. Detta därför att det då produceras resultat som är svåra att meddela till omvärlden. Det finns inte längre en entydig punkt där resultatet från funktionen presenteras, utan resultatet kan komma var som helst i funktionen och det behöver inte ens vara ett entydigt resultat, utan kan vara en av flera. Ytterligare en svårighet med denna riktlinje är att eftersom det handlar om PLC-system så behöver inte funktionerna enbart producera resultat i form av värden. Sannolikheten att de även används för att styra en maskin eller del av maskin är relativt stor. Denna typ av resultat kan bara förmedlas till användaren om det även går att få reda på variabelvärden i utvecklingsmiljön. Resultatet från denna riktlinje kan ges i form av att användaren får detta visat när denne på något sätt markerar en funktion. Ett annat alternativ är att resultatet, tillsammans med funktionsnamnet, presenteras i en lista. Fördelen med det sistnämnda är att det är lättare att se vilka delar av programmet som exekverats, samt i vilken ordning de exekverades, då under förutsättning att listan är i kronologisk ordning efter tid.

6.3 Riktlinjer för felsökning med induktion

Det finns ett antal sätt på vilka felsökning genom användandet av felsökningsmetoden induktion kan stödjas av utvecklingsmiljön och målmiljön. I kapitel 6.1.2 redovisas 10 punkter där ett felsökningsverktyg kan underlätta felsökningen. De riktlinjer som tas fram nedan är av den karaktären att de kan erbjuda eller hjälpa till att erbjuda ett stöd för dessa punkter.

R2.1 Felsökaren skall ha möjlighet att ta reda på variabelvärden under exekveringens gång.

Denna riktlinje erbjuder en lösning på S2.1. Denna punkt handlar om att felsökaren måste ha tillgång till variabelvärden vid felsökning med induktion. Detta eftersom induktionsmetoden går ut på att analysera de symptom ett fel ger upphov till för att på så sätt identifiera och lokalisera ett fel. Om felet visar sig i en eller flera variabler är detta ett bra sätt att kunna spåra ett fel.

Denna riktlinje erbjuder också en lösning för S2.5. Denna punkt handlar om att det är lättare att ta fram en hypotes om felet med induktionsmetoden om felsökaren kan få tillgång till de data som är lagrade på målmiljön. Detta därför att minnet kan innehålla viktiga spår efter felet.

Riktlinjen är samma som riktlinje nummer R1.8. En svårighet med riktlinjen är att det inte är säkert att all information om variablerna hinner föras ut till utvecklingsmiljön från målmiljön eftersom kommunikationshastigheten mellan de båda miljöerna är en begränsande faktor. En alternativ lösning är att felsökaren får specificera några variabler över vilka det skall hållas särskild uppsikt. På detta sättet behöver inte lika stora datamängder överföras.

R2.2 När exekveringen på målmiljön av någon anledning avslutas skall variabelvärden läsas av och skickas till utvecklingsmiljön.

Denna riktlinje erbjuder en del av lösningen till S2.1. Denna punkt handlar om att felsökaren måste ha tillgång till variabelvärden vid felsökning med induktion. Detta

eftersom induktionsmetoden går ut på att analysera de symptom ett fel ger upphov till för att på så sätt identifiera och lokalisera ett fel. Om felet visar sig i en eller flera variabler är detta ett bra sätt att kunna spåra ett fel.

Denna riktlinje erbjuder också en dellösning för S2.5. Denna punkt handlar om att det är viktigt att felsökaren får tillgång till information som är lagrad i målmiljön för att lättare kunna ta fram en hypotes om felet med induktionsmetoden. Då det inte är säkert att all information kan skickas till utvecklingsmiljön under exekveringens gång kan det hända att det finns spår av ett fel kvar i minnet utan att detta tidigare kommit fram under körning av programmet.

Denna riktlinje är i stort sett samma som riktlinje R1.4 kombinerad med riktlinje R1.6. Det finns ett antal möjligheter till varför ett program avslutas. Det kan avslutas på grund av att det kört klart, på grund av att ett så allvarligt fel uppstått att programmet inte klarar av att fortsätta eller på grund av att den som testar avbryter exekveringen. Denna riktlinje gör så att värdena för alla variablerna är nåbara för felsökaren då programmet på något sätt har avslutats. Om programmet har avslutats på grund av ett fel finns spåren av detta förmodligen kvar bland variablerna och kan därmed användas för att ta fram en hypotes om felet.

R2.3 Utvecklingsmiljön skall under exekveringens gång hållas informerad om vilken del av programmet som för närvarande exekveras på målmiljön.

Denna riktlinje erbjuder en lösning för S2.2. Denna punkt handlar om att felsökning genom induktion underlättas om felsökaren kan få reda på vilka delar av programmet som exekveras. Det går att få fram mycket information om ett fel om det går att finna samband mellan vilka delar i programmet som exekveras i kombination med när felet uppträder. Information om var felet inte uppträder säger nästan lika mycket om felet som information om var felet uppträder (Sharp, 1993).

Denna riktlinje kan införas med olika grader av detaljering. Då kommunikationen mellan målmiljön och utvecklingsmiljön är begränsad vad gäller hastighet är det inte säkert att det går att redovisa exakt vilken del som exekverats. För att minska mängden data som skickas mellan miljöerna går det att tänka sig att endast information om vilken modul som för närvarande exekveras visas. Vidare, om detta är möjligt, kan till exempel information om vilken funktion som exekveras för närvarande skickas över. Det finns många fler detaljnivåer som kan användas om detta tillåts av kommunikationsförhållandena mellan miljöerna.

R2.4 All information som skickas till utvecklingsmiljön från målmiljön skall loggas.

Denna riktlinje löser S2.3. Denna punkt handlar om att det är till stor hjälp för felsökningsprocessen med induktionsmetoden om information om exekveringen finns tillgänglig även efter det att denna har avslutats. Detta av flera orsaker. Om felet inte manifesterar sig på ett exceptionellt sätt är det inte säkert att felsökaren kan hitta den nödvändiga informationen under exekveringens gång. Om information om exekveringen finns kvar är det möjligt att felsökaren ändå kan hitta denna information och därmed komma närmare en lösning på felet. Vidare, eftersom informationen om felet skall organiseras då det är lättare att se samband om felet och därigenom spåra detsamma, är det bra om felsökaren inte behöver organisera informationen om felet medan programmet är igång, utan kan koncentrera sig på programkörningen och därefter kunna organisera informationen på det sätt han/hon själv vill.

Om det inte går att logga all information från målmiljön kan det vara möjligt för felsökaren att avgöra vilka data det är som skall lagras. Detta gör att utrymme sparas.

Denna riktlinje hjälper till att lösa S2.6. Denna punkt handlar om att det behövs stöd från utvecklingsmiljön för att felsökaren skall kunna organisera information om systemet enligt induktionsmetoden. Detta gäller för denna riktlinje eftersom den gör att informationen bibehålls, så att felsökaren i lugn och ro själv kan välja på vilket sätt informationen bäst kan organiseras.

R2.5 Felsökaren skall själv kunna välja när loggning av information från målmiljön skall ske.

Denna riktlinje löser S2.4. Denna punkt handlar om att felsökaren själv skall kunna avgöra när loggning av information skall ske (varför loggning bör ske tas upp av induktionsriktlinje 4). Då det kan röra sig om stora datamängder som skickas till utvecklingsmiljön kan det vara olämpligt av flera anledningar att loggning sker hela tiden. Det kan vara så att felet inträffar så sällan att enorma mängder information lagras i onödan. Detta innebär att information om felet ligger så inbakad i all onödig information att det kan vara svårt att hitta den. Vidare kan det vara så att minnesutrymmet inte nödvändigtvis är så stort att informationen på ett behändigt sätt kan lagras. Denna riktlinje gör att felsökaren själv får avgöra vilka delar som är intressanta att behålla information om. Detta gör det lättare att ta fram och strukturera relevant information om felet vid felsökning med induktionsmetoden.

Riktlinjen hjälper även till att stödja S2.6. Denna punkt handlar om att utvecklingsmiljön bör stödja felsökaren vid organiserandet av information enligt induktionsmetoden. Denna riktlinje gör detta genom att den ger en möjlighet för felsökaren att själv avgöra vilken information som är viktig att spara på och vilken information som är nödvändig. Vidare underlättar den organiserandet genom att hålla nere de datamängder som lagras.

R2.6 En sammanställd lista med tillgänglig information om en exekvering skall kunna presenteras för felsökaren.

Denna riktlinje hjälper till att lösa S2.6. Denna punkt handlar om att utvecklingsmiljön skall hjälpa felsökaren att organisera information om ett fel enligt induktionsmetoden. Detta därför att de datamängder som kan stå till förfogande kan vara så stora att de är svåra att sätta sig in i.

Riktlinjen kan realiseras på ett antal sätt. Värdet på variablerna kan, tillsammans med variabelnamnen, redovisas. Om det finns tillgång till en variabels historik, det vill säga att variabelns förändringar finns lagrade, kan dessa redovisas i kronologisk ordning. Vidare, om detta finns tillgängligt, kan information redovisas om vart i programmet variabeln fick sina värden. Dessutom kan information om vilka delar av programmet som exekverats redovisas. Vidare kan en tidsskala visas som talar om när förändringar och programtillstånd har inträffat.

Denna riktlinje hjälper även till att stödja S2.7. Denna punkt handlar om att utvecklingsmiljön skall stödja felsökarens verksamhet när det gäller att se samband i informationen med induktionsmetoden. Detta understöds genom att information om programkörningen kan redovisas på ett sådant sätt att relationer och samband kan åskådliggöras. Om informationen kan redovisas i kronologisk ordning går det även att spåra fel med denna riktlinje.

R2.7 När en variabel i koden markeras skall dess senaste värde visas för programmeraren.

Denna riktlinje hjälper till att lösa S2.6. Denna punkt handlar om att utvecklingsmiljön skall hjälpa felsökaren att organisera information om ett fel enligt induktionsmetoden. Detta därför att de datamängder som kan stå till förfogande kan vara så stora att de är svåra att sätta sig in i. Denna riktlinje gör det möjligt för felsökaren att i koden kunna få reda på en variabels värde. Om en variabels historik finns lagrad, det vill säga att det finns information som visar alla förändringar hos en variabel under programmets gång, kan det tänkas att denna information kan användas så att variabelns värde som den hade vid just det stället i programmet som är markerad visas. Denna riktlinje ger således en möjlighet att i efterhand spåra sig tillbaka i programkoden efter felet och på så sätt ta fram ytterligare information om det.

Denna riktlinje hjälper också till att stödja S2.7. Denna punkt handlar om att utvecklingsmiljön skall stödja felsökarens verksamhet när det gäller att se samband i informationen med induktionsmetoden. Genom denna riktlinje kan felsökaren, om förutsättningarna är de rätta, spåra ett fel genom programkoden. Om inte detta är möjligt går det i alla fall att se vissa samband genom att kunna jämföra variabler med varandra.

R2.8 Utvecklingsmiljön skall kunna visa i vilka programuttryck en markerad variabel finns med i.

Denna riktlinje stödjer S2.8. Denna punkt går ut på att det är bra att kunna se vilka delar av programmet som påverkas vid ett fel. Om ett fel kan spåras till en variabel ger denna riktlinje en möjlighet att dels kunna spåra variabeln bakåt för att se varifrån felet har kommit och därigenom spåra sig tillbaka till felet och dels kunna spåra framtida effekter av felet. Detta gör att förutsägelser om felet relativt enkelt kan göras för att på så sätt, enligt induktionsmetoden, kunna bevisa en hypotes om felet. En utvidgning av denna riktlinje kan vara att även övriga variabler som finns med på de rader den markerade variabeln finns spåras bakåt och framåt för att ytterligare kunna göra förutsägelser.

Då denna riktlinje ger felsökaren en möjlighet att göra förutsägelser om den hypotes denne har ställt angående felorsaken och fellokaliseringen stöds även S2.10. Denna punkt handlar om att utvecklingsmiljön skall hjälpa felsökaren att bevisa den hypotes som dragits om felet. Förutsägelserna medför att hypotesens riktighet kan verifieras.

R2.9 Utvecklingsmiljön skall kunna visa i vilka tidigare programuttryck en markerad variabel finns med i och även spåra övriga variabler som är inblandade i dessa programuttryck.

Denna riktlinje löser S2.9. Denna punkt går ut på att utvecklingsmiljön skall hjälpa felsökaren att ta reda på vilka programuttryck som är inblandade i ett fel. Detta innebär förhoppningsvis att stora delar programmet kan uteslutas vid försök att lokalisera ett fel vid felsökning med induktionsmetoden.

Om felsökaren vet att den markerade variabeln innehåller ett fel vid den tidpunkt då exekveringen har kommit till det ställe där variabeln är markerad innebär detta att felet bara kan ha introducerats i variabeln i de programuttryck den har varit med i tidigare (Agrawal, DeMillo & Spafford, 1990). Detta innebär att en mer preciserad hypotes om felet kan dras. Vidare, om även de rader markeras som innehåller variabler som varit inblandade i att påverka den markerade variabeln, finns förmodligen felet på en av dessa programrader.

6.4 Riktlinjer för felsökning med backtracking

Det finns ett antal sätt på vilka felsökning genom användandet av felsökningsmetoden backtracking kan stödjas av utvecklingsmiljön och målmiljön. I kapitel 6.1.3 redovisas 6 punkter där ett felsökningsverktyg kan underlätta felsökningen. De riktlinjer som tas fram nedan är av den karaktären att de kan erbjuda eller hjälpa till att erbjuda ett stöd för dessa punkter.

R3.1 Utvecklingsmiljön skall kunna visa vilka tidigare programuttryck en markerad variabel finns med i.

Denna riktlinje löser S3.1. Denna punkt går ut på att det behövs stöd för att spåra ett fel bakåt i programkoden. Detta är en av de huvudsakliga uppgifterna vid felsökning med backtracking. Om en variabel med ett felaktigt värde eller som misstänks innehålla ett fel identifieras går det att konstatera att detta fel har introducerats i variabeln på den aktuella eller någon av de tidigare programuttryck variabeln finns med i. Detta fel kan ha introducerats på två sätt. Antingen har det kommit från en annan variabel eller också är ett av programuttrycken variabeln är med i felaktigt. Denna riktlinje ger en möjlighet att ta ut de programuttryck ett fel kan ha introducerats på för variabeln. Om felet har introducerats av en annan variabel går det att spåra även denna variabel bakåt. En nackdel med denna riktlinje är att den kan ha svårt med program där exekveringen ”hoppar” fram och tillbaka eller när det finns flera exekveringsvägar eller iterationer. Det blir i dessa fall svårt att avgöra vilka programuttryck som föregår det aktuella.

R3.2 Utvecklingsmiljön skall kunna visa i vilka tidigare programuttryck en markerad variabel finns med i och även spåra övriga variabler som är inblandade i dessa programuttryck.

Denna riktlinje löser S3.1. Denna punkt säger att det behövs stöd för att spåra ett fel bakåt i programkoden, vilket är en av de huvudsakliga uppgifterna vid felsökning med backtracking. Om en variabel som innehåller ett felaktigt värde eller misstänks kan innehålla ett fel hittas går det att konstatera att detta fel har introducerats i något av de programuttryck variabeln tidigare varit med i. Om felet har introducerats av en annan variabel kan det konstateras att det felaktiga programuttrycket kan finnas på någon av de uttryck denna variabel tidigare varit med i. Denna riktlinje ger en möjlighet att ta ut precis de programrader som kan ha introducerat ett fel. Om en felaktig variabel har konstaterats vid en viss tidpunkt i programmet ger denna riktlinje en möjlighet att begränsa sökningen efter felet till de programuttryck denna riktlinje identifierar. Detta därför att dessa uttryck är de enda som kan påverka värdet hos den felaktiga variabeln.

R3.3 Utvecklingsmiljön skall kunna tala om för felsökaren från vilka ställen i ett program en specifik modul anropas.

Denna riktlinje löser S3.2. Denna punkt går ut på att det behövs ett stöd från utvecklingsmiljön för att kunna spåra fel mellan moduler. Detta är ett problem vid felsökning med backtracking, då moduler kan anropas från många ställen i ett program. Spårning av fel mellan moduler försvåras därmed. Denna riktlinje ger en möjlighet för felsökaren att få reda på från vilka ställen i ett program en viss modul anropas. På detta sätt slipper felsökaren själv ta reda på detta i ett potentiellt stort program, detta sköts av utvecklingsmiljön och presenteras vid förfrågan för felsökaren, och tid kan därmed sparas.

R3.4 Utvecklingsmiljön skall kunna tala om vilka moduler som exekverats och i vilken ordning dessa exekverats vid programkörning på målmiljön.

Denna riktlinje erbjuder en lösning för S3.2. Denna punkt går ut på att det behövs ett stöd från utvecklingsmiljön för att kunna spåra fel mellan moduler. Detta är ett problem vid felsökning med backtracking, då moduler kan anropas från många ställen i ett program. Spårning av fel mellan moduler försvåras därmed. Följandet av denna riktlinje ger en möjlighet för felsökaren att, om ett fel visas i en modul, och det kan konstateras att felet inte härrör från modulen utan propagerats från en annan, spåra felet till den tidigare modulen. Detta är till stor hjälp då felsökaren metodiskt kan stega sig tillbaka i programmet och veta vilken modul felet kommer ifrån, även om det finns flera moduler som anropar den aktuella under en programkörning. Denna riktlinje kräver att information om vilken modul som exekveras samt när denna exekveras ständigt skickas från målmiljön till utvecklingsmiljön under exekveringens gång.

Riktlinjen erbjuder också ett stöd för S3.3. Denna punkt går ut på att ge stöd för felsökaren att stega sig bakåt i koden i sökandet efter fel med backtrackingmetoden. Riktlinjen ger en möjlighet för felsökaren att kunna stega sig bakåt modulvis. Dock krävs det då att tillräckligt med information finns om felet för att kunna spåra det mellan moduler.

R3.5 Uppdatering av variablers värden på målmiljön skall skickas till utvecklingsmiljön och lagras under exekveringens gång.

Denna riktlinje ger ett stöd för S3.3. Denna punkt går ut på att ge stöd för felsökaren att stega sig bakåt i koden i sökandet efter fel med backtrackingmetoden. Riktlinjen ger en möjlighet att få reda på variablernas historia i ett program och därigenom kunna ta sig bakåt i programmet. Om de uttryck som orsakade en förändring hos en variabel kan spåras kan även det uttryck som orsakade felaktigheten spåras. Nackdelen med denna riktlinje är att kommunikationen mellan målmiljö och utvecklingsmiljö är begränsad (se inledningen i kapitel 3). Detta innebär att det inte är säkert att all information om variabelförändringar kommer till utvecklingsmiljöns vetskap, då det kan vara mycket information som behöver skickas. Ett sätt att minska mängden information är att ge felsökaren en möjlighet att välja vilka variabler som skall följas.

Om informationen kan presenteras för felsökaren under tiden programmet körs på målmiljön erbjuder denna riktlinje även en dellösning till S3.5. Denna punkt handlar om att det behövs stöd för att kunna få reda på ett programs tillstånd under exekvering på målmiljön. Då denna riktlinje ger en möjlighet att få reda på variabelvärdena under tiden går det innebär detta att programmets tillstånd kan synliggöras för felsökaren.

Riktlinjen ger även ett stöd för S3.6. Denna punkt går ut på att felsökaren skall kunna få reda på tidigare programtillstånd för att lättare kunna felsöka med backtracking. Då variablerna lagras är informationen om programmets tillstånd sparad. En förutsättning är dock att det går att få reda på all information om programmet.

R3.6 Felsökaren skall ha möjlighet att använda sig av en simulator för att simulera den miljö programmet skall styra samt simulera målmiljö.

Denna riktlinje ger stöd åt S3.4. Denna punkt går ut på att felsökaren skall kunna avbryta exekveringen av ett program utan att omgivningen kommer till skada, vilket ofta är risken med PLC-system. Detta kan ibland behövas för att systematiskt kunna stega sig bakåt i ett program med backtrackingmetoden för att testa vart felet först manifesterar sig. Riktlinjen ger en möjlighet att utföra denna aktivitet då en simulator undviker en fysisk omgivning. Nackdelen med att försöka simulera PLC-system är dock

att de kan vara väldigt stora med tusentals komponenter som måste samordnas (se kapitel 2.1 om målmiljö). Det blir då svårt att kunna göra en realistisk simulation. Det riskerar att kosta alldeles för mycket i dessa fall. En möjlig lösning är att kunna simulera bara små delar av systemet.

6.5 Användning av riktlinjer tillsammans med IEC 1131-3

Det är dags att avgöra om de identifierade riktlinjerna kan användas tillsammans med standard IEC 1131-3 eller inte. Kapitlen om standard IEC 1131-3 (se kapitel 2.2.1) och om testvänlighet och struktur hos en programvara som följer standarden (se kapitel 2.4) ger svar på om riktlinjerna kan användas vid felsökning i standarden.

6.5.1 Riktlinjer i brute force-metoden

Riktlinjerna som är framtagna för felsökning med brute force-metoden, kapitel 6.2 kan delas in i två grupper: dels de riktlinjer som inte blandar in något av programmeringsspråken i standarden, det vill säga kan följas utan att använda sig av något av programmeringsspråken och dels de riktlinjer som visualiseras eller följs vid användandet av programmeringsspråken i standarden.

De riktlinjer från kapitel 6.2 som inte blandar in några programmeringsspråk är följande:

R1.1. Utvecklingsmiljön skall kunna ta reda på en variabels position i minnet på målmiljön och presentera denna position för felsökaren.

R1.2. Utvecklingsmiljön skall på egen hand leta reda på en variabels position i en minnesdump och kunna presentera variabelns värde för felsökaren.

R1.3. En minnesdump skall produceras när exekvering på målmiljön avbryts på ett annat sätt än att programmet är avslutat.

R1.4. En minnesdump skall produceras då exekvering avslutas.

R1.5. Felsökaren skall kunna ta en minneskopia när som helst.

R1.8. Uppdatering av variablers värden på målmiljön skall skickas till utvecklingsmiljön under exekveringens gång.

R1.9. Vid moduluppdelade program skall information om vilken modul som exekveras för tillfället skickas till utvecklingsmiljön.

R1.10. Information om variablers värden under exekvering på målmiljön skall kunna vara åtkomliga från utvecklingsmiljön utan att detta explicit måste anges av felsökaren genom ändringar i programkoden.

R1.11. Låt så mycket av kommunikationen mellan målmiljön och utvecklingsmiljön som möjligt skötas utan inblandning från användare.

R1.12. Möjlighet att använda en simulator för att simulera den miljö programmet skall styra samt simulera målmiljö..

R1.13. Möjlighet att simulera delar av händelseförlopp.

R1.14. Felmeddelanden skall vara läsbara.

R1.17. Om tillgång till variabelvärden finns skall det gå att från utvecklingsmiljön ta reda på om villkor är sanna eller falska.

R1.18. Målmiljön talar under exekveringens gång om för utvecklingsmiljön om ett villkor är sant eller falskt.

För riktlinjerna R1.1- R1.5, R1.8- R1.10 gäller att det är tillåtet att ta fram information som ligger i minnet hos målmiljön. Detta kan uttolkas ur kapitel 2.4 som handlar om testvänlighet hos ett program. Där står det att interna tillstånd är åtkomliga under exekvering av programmet, vilket innebär att det är tillåtet att läsa i minnet på målmiljön. Det är därför inte heller förbjudet att föra över informationen till utvecklingsmiljön. Informationen kan presenteras för felsökaren på många sätt varför det sätt kan väljas som är mest bekvämt för det programmeringsspråk som används.

För R1.11 gäller att automatisera kommunikationen. Standarden täcker inte in kommunikation mellan ett PLC-system och ett system som inte är ett PLC-system (se kapitel 2.2.1). Detta innebär att denna riktlinje inte bryter standarden.

För R1.12 och R1.13 gäller att då de är simuleringar av PLC-system så kan de räknas som egna PLC-system (dock utan att styra verkliga maskiner, bara fiktiva). Detta innebär att de, om de skall vara realistiska, automatiskt måste följa standarden.

För R1.14 gäller att det inte är förbjudet enligt standard IEC 1131 att skicka information om systemet till omvärlden (se kapitel 2.4). Felmeddelanden innehåller information om systemet och bryter därmed inte mot standarden.

För R1.17 och R1.18 gäller att det inte är förbjudet enligt standard IEC 1131 att skicka information om ett PLC-systems tillstånd under exekvering. Enligt kapitel 2.4, om testvänlighet hos programvara som följer standarden, är det möjligt att kunna se PLC-systemets tillstånd från omvärlden. Detta innebär att riktlinjerna ligger inom ramen för vad standarden tillåter.

De riktlinjer från kapitel 6.2 som blandar in programmeringsspråken i standarden är riktlinjerna nummer:

R1.6. Utvecklingsmiljön skall kunna analysera en minnesdump och presentera en lista med variabler och tillhörande värden hämtade från minnesdumpen.

R1.7. Utvecklingsmiljön skall kunna analysera en minnesdump och presentera det värde minnesdumpen har på en variabel som är markerad i källkoden.

R1.15. Det skall gå att se vart en variabel senast sattes och vilket värde den har.

R1.16. Visa i vilka programuttryck en specificerad/markerad variabel förekommer.

R1.19. Felsökaren skall kunna få reda på resultatet från funktioner under och efter exekvering.

För R1.6 och R1.7 gäller att det finns inget i standard IEC 1131 som säger att det är förbjudet att leta reda på en eller flera variabler i en minnesdump. Då informationen som tas fram ur minnesdumpen kan presenteras på olika sätt kan det sätt väljas som är mest bekvämt för det programmeringsspråk som används.

För R1.15 gäller att det inte är förbjudet enligt standard IEC 1131 att skicka information om ett PLC-systems tillstånd under exekvering. Enligt kapitel 2.4, om testvänlighet hos programvara som följer standarden, är det möjligt att kunna se PLC-systemets tillstånd från omvärlden. Detta innebär att information om variabelvärden kan skickas till utvecklingsmiljön under exekveringens gång. Då riktlinjen inte påverkar någon notation i standarden kan den realiseras utan att riskera att bryta standarden.

För R1.16 gäller att denna riktlinje varken blandar in händelser i ett PLC-system eller notation i något av de fem programmeringsspråken och riskerar därmed inte att bryta standarden.

För R1.19 gäller att det inte enligt standard IEC 1131 är förbjudet att skicka information om ett PLC-systems tillstånd under exekvering. Enligt kapitel 2.4, om testvänlighet hos programvara som följer standarden, är det möjligt att kunna se PLC-systemets tillstånd från omvärlden. Då även resultaten från en funktion ingår i systemets tillstånd är denna riktlinje tillåten enligt standarden. Då riktlinjen inte påverkar någon notation i standarden kan den realiseras utan att riskera att bryta standarden.

6.5.2 Riktlinjer i induktionsmetoden

Riktlinjerna som är framtagna för felsökning med induktionsmetoden, kapitel 6.3 kan delas in i två grupper: dels de riktlinjer som inte blandar in något av programmeringsspråken i standarden, det vill säga kan följas utan att använda sig av något av programmeringsspråken och dels de riktlinjer som visualiseras eller följs vid användandet av programmeringsspråken i standarden.

De riktlinjer från kapitel 6.2 som inte blandar in några programmeringsspråk är följande:

R2.2. När exekveringen på målmiljön av någon anledning avslutas skall variabelvärden läsas av och skickas till utvecklingsmiljön.

R2.3. Utvecklingsmiljön skall under exekveringens gång hållas informerad om vilken del av programmet som för närvarande exekveras på målmiljön.

R2.4. All information som skickas till utvecklingsmiljön från målmiljön skall loggas.

R2.5. Felsökaren skall själv kunna välja när loggning av information från målmiljön skall ske.

R2.6. En sammanställd lista med tillgänglig information om en exekvering skall kunna presenteras för felsökaren.

För R2.2 och R2.3 gäller att det är tillåtet att ta fram information som ligger i minnet hos målmiljön. Detta kan uttolkas ur kapitel 2.4 som handlar om testvänlighet hos ett program. Där står det att interna tillstånd är åtkomliga under (och därmed även efter i fallet med R2.2) exekvering av programmet, vilket innebär att det är tillåtet att läsa i minnet på målmiljön. Det är därför inte heller förbjudet att föra över informationen till utvecklingsmiljön. Informationen kan presenteras för felsökaren på många sätt varför det sätt kan väljas som är mest bekvämt för det programmeringsspråk som används.

För R2.4 och R2.5 gäller att de inte blandar in standard IEC 1131 i och med det att de inte uttalar sig om hur de får vetskap om informationen från PLC-systemet och därför fritt kan användas vid utveckling och körning av ett PLC-system.

För R2.6 gäller att den inte nödvändigtvis blandar in standard IEC 1131 och kan därför användas vid utveckling och körning av ett PLC-system.

De riktlinjer från kapitel 6.3 som blandar in programmeringsspråken i standarden är riktlinjerna nummer:

R2.1. Felsökaren skall ha möjlighet att ta reda på variabelvärden under exekveringens gång.

R2.7. När en variabel i koden markeras skall dess värde visas för programmeraren.

R2.8. Utvecklingsmiljön skall kunna visa i vilka programuttryck en markerad variabel finns med i.

R2.9. Utvecklingsmiljön skall kunna visa i vilka tidigare programuttryck en markerad variabel finns med i och även spåra övriga variabler som är inblandade i dessa programuttryck.

För R2.1 gäller att det är inte enligt standard IEC 1131 förbjudet att skicka information om ett PLC-systems tillstånd under exekvering. Enligt kapitel 2.4, om testvänlighet hos programvara som följer standarden, är det möjligt att kunna se PLC-systemets tillstånd från omvärlden. I och med det att den inte behöver blanda in vare sig händelser i ett PLC-system eller notation i något av de fem programmeringsspråken riskerar den inte att bryta standarden.

För R2.7 gäller att det inte är förbjudet enligt standard IEC 1131 att skicka information om ett PLC-systems tillstånd under exekvering. Enligt kapitel 2.4, om testvänlighet hos programvara som följer standarden, är det möjligt att kunna se PLC-systemets tillstånd från omvärlden. Detta innebär att information om en variabels värde kan finnas tillgänglig. Då riktlinjen inte påverkar någon notation i standarden kan den realiseras utan att riskera att bryta standarden.

För R2.8 och R2.9 gäller att då de inte blandar in notation i något av de fem programmeringsspråken, utan bara presenterar information, riskerar de därmed inte att bryta standarden. R2.9 är lättast att realisera i strikt sekventiella programmeringsspråk (se kapitel 2.2.1 om IEC 1131) då det är lättare att tala om vad som ligger tidigare i koden.

6.5.3 Riktlinjer i backtrackingmetoden

Riktlinjerna som är framtagna för felsökning med backtrackingmetoden, kapitel 6.4 kan delas in i två grupper: dels de riktlinjer som inte blandar in något av programmeringsspråken i standarden, det vill säga kan följas utan att använda sig av något av programmeringsspråken och dels de riktlinjer som visualiseras eller följs vid användandet av programmeringsspråken i standarden.

De riktlinjer från kapitel 6.4 som inte blandar in några programmeringsspråk är följande:

R3.4. Utvecklingsmiljön skall kunna tala om vilka moduler som exekverats och i vilken ordning dessa exekverats vid programkörning på målmiljön.

R3.5. Uppdatering av variablers värden på målmiljön skall skickas till utvecklingsmiljön och lagras under exekveringens gång.

R3.6. Felsökaren skall ha möjlighet att använda sig av en simulator för att simulera den miljö programmet skall styra samt simulera målmiljö.

För R3.4 och R3.5 gäller att det är tillåtet enligt standarden att ta reda på information om PLC-systemet under exekveringens gång. Detta kan uttolkas ur kapitel 2.4 som handlar om testvänlighet hos ett program. Där står det att interna tillstånd är åtkomliga under exekvering av programmet, vilket innebär att det är tillåtet att hämta information på målmiljön. Det är därför inte heller förbjudet att föra över informationen till utvecklingsmiljön. Informationen kan presenteras för felsökaren på många sätt varför det sätt kan väljas som är mest bekvämt för det programmeringsspråk som används.

För R3.6 gäller att då den gäller simuleringar av PLC-system så kan simuleringen räknas som ett eget PLC-system (dock utan att styra verkliga maskiner, bara fiktiva). Detta innebär att den, om den skall vara realistisk, automatiskt måste följa standarden.

De riktlinjer från kapitel 6.4 som blandar in programmeringsspråken i standarden är riktlinjerna nummer:

R3.1. Utvecklingsmiljön skall kunna visa vilka tidigare programuttryck en markerad variabel finns med i.

R3.2. Utvecklingsmiljön skall kunna visa i vilka tidigare programuttryck en markerad variabel finns med i och även spåra övriga variabler som är inblandade i dessa programuttryck.

R3.3. Utvecklingsmiljön skall kunna tala om för felsökaren från vilka ställen i ett program en specifik modul anropas.

För R3.1, R3.2 och R3.3 gäller att då de inte blandar in notation i något av de fem programmeringsspråken, utan bara presenterar information, riskerar de därmed inte att bryta standarden. R3.2 är lättast att realisera i strikt sekventiella programmeringsspråk (se kapitel 2.2.1) då det är lättare att tala om vad som ligger tidigare i koden.

7 Slutsatser

En stor del av tiden som läggs ned på att ta fram programvaror går idag ut på att testa och söka efter fel i desamma. Trots att företagen lägger ner mellan 30 och 40 procent av arbetsmödan (Pressman, 1997) på testning och felsökning är det långt ifrån säkert att alla fel har hittats. Det är till och med så att det förmodligen finns kvar fel i programvaran då tester bara kan påvisa närvaron av fel, ej frånvaron.

När ett test påvisar ett fel gäller det att lokalisera, identifiera och reparera felet. Detta kan vara en nog så svår aktivitet då programmen kan bestå av miljoner rader kod skrivna av dussintals olika programmerare. Det kan mycket väl vara så att den som försöker lokalisera felet inte alls är bekant med det kodavsnitt felet ligger i. Denne felsökare behöver därför så mycket stöd som möjligt för att kunna hitta och reparera felet till exempel av ett felsökningsverktyg.

En typ av programvaror är särskilt känsliga för fel. Det rör sig om sådana programvaror som skall användas för att kontrollera processer och företeelser som inträffar fysiskt utanför datavärlden och där även reaktionstider för programmen är viktiga för att undvika skador, så kallade realtidssystem. Testning och felsökning i denna typ av programvaror är ofta särskilt svår då programförloppen påverkas av den fysiska verkligheten och det är svårt att kontrollera och få information om vad det är som händer under testerna.

PLC-system är en typ av realtidssystem som används för att styra maskiner och tillverkningsprocesser inom industrin. PLC-systemen finns beskrivna i standard IEC 1131 och dess programvaror i IEC 1131-3.

Detta arbete tar på sig två uppgifter:

1. att ta fram det stöd ett felsökningsverktyg kan erbjuda felsökningsprocessen enligt felsökningsmetoderna brute force, induktion och backtracking;
2. att undersöka om det identifierade stödet kan användas tillsammans med standard IEC 1131-3.

7.1 Riktlinjer för verktygsstöd vid felsökning

Den första delen av uppgiften handlar om att ta fram vilket stöd ett felsökningsverktyg kan erbjuda (egenskaper hos verktyget) vid felsökning enligt ett antal felsökningsmetoder. Denna uppgiftsdel är generell så till vida att, även om många begränsningar sätts av PLC-systemet i denna fas, så är de riktlinjer som tas fram inte riktade direkt mot PLC-system. De egenskaper som tas fram kan användas som stöd även i felsökningsverktyg för andra system.

Det identifieras i detta arbete ett antal punkter där felsökning enligt felsökningsmetoderna kan behöva stöd från ett felsökningsverktyg (se kapitel 5) för att underlätta lokalisering av fel. För varje sådan identifierad punkt finns minst en egenskap genomgången som erbjuder just detta stöd. Denna del av uppgiften anses därmed genomförd. (För de identifierade egenskaperna se kapitel 6.2 till 6.4).

7.2 Riktlinjernas användbarhet tillsammans med IEC 1131-3

Den andra delen av arbetet går ut på att undersöka de stöd (egenskaper) som tas fram under den första delen för att se om de kan användas tillsammans med standard IEC 1131-3. Då denna del av uppgiften är inriktad mot PLC-system är delen inte lika användbar för andra system.

Denna del av arbetet kan anses vara klart då det i arbetet konstateras att varje identifierat stöd (egenskap) kan användas tillsammans med standard IEC 1131-3 utan att bryta den. (För genomförandet av denna del av uppgiften se kapitel 6.5.)

7.3 Diskussion

Detta arbete tillför ett antal saker till felsökning i programvaror och särskilt då realtidssystem med inriktning mot PLC-system.

Arbetet genomför en teoretisk undersökning och identifierar vilka stöd ett felsökningsverktyg kan erbjuda för att underlätta processen att lokalisera och identifiera ett fel. Undersökningen är metodisk och alla stöd, även benämnda riktlinjer, finns motiverade utifrån de utvalda felsökningsmetoderna. Detta innebär att stöden finns teoretiskt förankrade och att det finns en klar anledning till varför de kan räknas som stöd till felsökningsprocessen.

Det går alltså att säga att detta arbete producerar ett antal riktlinjer för vad ett felsökningsverktyg kan erbjuda felsökningsprocessen utan hänsyn till vad tidigare felsökningsverktyg innehåller. Detta innebär att flera riktlinjer kan vara av en typ som ej tidigare är använda, medan andra riktlinjer används frekvent. Styrkan med detta arbete är att alla riktlinjer förankras teoretiskt till minst en felsökningsmetod vilket visar att de fyller ett väl definierat syfte.

Arbetet undersöker även på ett formellt sätt om de stöd för felsökning ett felsökningsverktyg kan erbjuda kan användas tillsammans med standard IEC 1131-3. Detta innebär att även de som använder sig av denna typ av programsystem som är väldigt begränsade vad gäller felsökningsförutsättningar kan använda arbetet trots systemens begränsningar.

7.4 Fortsatt arbete

Det finns ett antal områden där vidare undersökning som antingen bygger på eller har anknytning till detta arbete kan vara av intresse.

De riktlinjer för det stöd ett felsökningsverktyg kan erbjuda för felsökningsmetoderna brute force, induktion samt backtracking behöver utvärderas. Det är inte säkert att alla riktlinjer kommer att göra någon praktisk nytta vid felsökning ur användarens synvinkel. Det behövs därför en utvärdering ur användarens synvinkel hur bra riktlinjerna är. Detta kan ske genom enkäter, intervjuer eller genom att realisera riktlinjerna och låta typiska användare prova. Typiska användare är:

- De som utvecklar program, särskilt PLC-program.
- De som skall underhålla programsystemen när de är i drift.
- Övriga som kan tänkas komma i kontakt med programsystemen vid felsituationer.

Detta arbete genomför ingen design eller implementation av riktlinjerna som tas fram. Ett framtida arbete är att realisera riktlinjer i ett felsökningsverktyg som kan användas av dem som kan tänkas ha nytta av att de realiseras. Ett alternativ är att bygga ut befintliga felsökningsverktyg så att de innefattar intressanta riktlinjer som inte finns realiserade tidigare. Denna uppgift kan knytas till uppgiften i stycket ovan då realiseringen möjliggör bra användarundersökningar.

Arbetet begränsar sig till enbart tre felsökningsmetoder: brute force, induktion samt backtracking. Det finns givetvis ett antal felsökningsmetoder utöver dessa samt intressanta kombinationer av felsökningsmetoder. En ytterligare undersökning skulle kunna göras för att identifiera riktlinjer även för dessa felsökningsmetoder. Det är långt ifrån säkert att detta arbete identifierar alla möjliga riktlinjer för verktygsstöd som är nyttiga oavsett felsökningsmetod. De som felsöker behöver inte bara använda sig av de felsökningsmetoder som är genomgångna här.

Detta arbete, trots att det är särskilt inriktat mot PLC-system vilka är realtidssystem, tar inte upp tidsaspekter i sitt stöd för felsökning. Då det även kan förekomma tidsfel i programvaror för realtidssystem, se kapitel 2.5 om feltyper i realtidssystem, utelämnas en bit av felsökningsprocessen i denna typ av system. Ett fortsatt arbete kan inrikta sig att ta fram riktlinjer för att underlätta felsökning då även tidsfel behöver tas i åtanke. Ett felsökningsverktyg som ger stöd för sökning efter värdefel och tidsfel samt kombinationen av dessa (se kapitel 2.5 om olika typer av fel i realtidssystem) är mycket användbart för utvecklare av realtidssystem.

Appendix A Binär elektronik AB

Företaget Binär elektronik AB har sitt huvudkontor i Trollhättan. De hade under 1997 i genomsnitt 31 anställda och en försäljningsvolym på 40 Mkr. Företagets affärside bygger på att tillgodose tillverkningsindustrins behov av att effektivisera sin produktion. De produkter företaget utvecklar, tillverkar och marknadsför är datorbaserade system för avancerad styrning av automatiserade produktionsanläggningar, till exempel svetsrobotar och staplingsmaskiner. Företagets kunder finns främst inom verkstadsindustrin och särskilt då inom fordonsindustrin. Företagets främsta konkurrenter är främst större koncerner, till exempel Siemens och Mitsubishi. Binärs produkter används av företag i hela världen. Till företagets kundkrets kan nämnas företag såsom Audi, BMW, Ford, Saab och Volvo.

Binär tillhandahåller ett antal olika produkter. Dessa är främst till för PLC- (Programmable Logic Controller) och positioneringssystem. De stora produkterna företaget tillhandahåller är BiFas 40 och BiFas UHS (Uni Hardware System). Vidare tillhandahåller företaget ett antal olika operatörspaneler samt, för vissa kunder, specialtillverkade applikationer till exempel testprogram för kvalitetssäkran av produkter. Vid programmering av BiFas-produkterna används BiSoft 1131, en utvecklingsmiljö som följer standarden IEC 1131-3. Binär tillhandahåller även tjänster avseende idrifttagning av anläggningar, stand by-service för anläggningar med BiFas-system samt utbildning i programmering och underhåll av BiFas-systemen.

Referenser

Aaby, A [aabyan@wwc.edu], *Debugging*, http://moonbase.wwc.edu/~cs_dept/KU/SEBOOK/Debugging.html, 980227, (990407).

Agrawal, H, DeMillo, R & Spafford, E (1990). An Execution Backtracking Approach to Program Debugging. W.Lafayette: Purdue University, Department of Computer Science.

Burns, A & Wellings, A (1997). *Real-Time Systems and Programming Languages*. Second edition. Addison-Wesley.

Lewis, R (1996), *Programming industrial control systems using IEC 1131-3*. London: The Institution of Electrical Engineers.

Myers, G (1979). *The Art of Software Testing.* John Wiley and Sons Inc.

Pan, H & Spafford, E (1992). Heuristics for Automatic Localization of Software Faults. W. Lafayette: Purdue University, Department of Computer Science.

Paul, J [jody@acm.org], *General Testing and Debugging Guidelines*, http://apex.cudenver.edu/SWE/TD/TestDebug.html, 970521, (990208).

Pressman, R (1997), *Software Engineering: A Practitioner's approach*. Fourth edition. McGraw-Hill Inc.

Sharp, A (1993), *Software quality and productivity*. New York: Van Nostrand Reinhold.