Systemering av driftdatabas: vidareutveckling och omkoppling till central lagring

Jonas Håkansson

1 mars 2006
Master's Thesis in Computing Science, 20 credits
Supervisor at CS-UmU: Ola Ågren
Examiner: Per Lindström

UMEÅ UNIVERSITY
DEPARTMENT OF COMPUTING SCIENCE
SE-901 87 UMEÅ
SWEDEN

Sammanfattning

Fordonsdiagnostik är ett viktigt område för dagens fordonsutvecklare. Att kunna logga data från ett fordons styrsystem och analysera detta kan ge svar på många frågor rörande felorsaker, bränsleförbrukning och hållbarhet. Denna rapport avhandlar ett exmensarbete med uppgift att konstruera ett analysverktyg för att analysera driftdata, dvs. diverse mätvärden som loggas kontinuerligt, från populationer av fordon. Den färdiga produkten är en Windowsapplikation utvecklad med C# och arbetar mot en SQL-databas där diagnosdata från fordonen automatiskt lagras. Rapporten beskriver de olika stegen i produktutvecklingen men ger även en inblick och bakgrund till de verktyg som använts och motiven bakom valen av dessa.

Duty-Cycle Data Analysis: Development and switching to central storage

Abstract

Vehicle diagnostics is an important issue for the vehicle developers of today. To be able to log the behaviour of a vehicle system and to be able to analyse it can give answers to many questions regarding errors, fuel consumption and durability. Duty-cycle data is one kind of diagnostic data that are logged once each second. This master thesis has the goal of constructing an analysis tool for analysis of populations of vehicles. The finished product is a windows application developed in C# and connects to an SQL database where duty-cycle data from the vehicles are stored automatically. The report contains descriptions of the different stages of the product development but also gives insight and background to the tools that was used and the motives for choosing them.

Innehåll

1 Introduktion **1**
1.1 Bakgrund . . . 1
1.2 Sammanhang . . . 2
1.3 Indelning . . . 2

2 Problembeskrivning **3**
2.1 Mål . . . 3
2.2 Metod . . . 3
2.2.1 Analys . . . 4
2.2.2 Krav och behovsinsamling . . . 4
2.2.3 Prototyp för användargränssnittet . . . 4
2.2.4 Val av utvecklingsmiljö och verktyg . . . 4
2.2.5 Implementation . . . 4
2.2.6 Utvärdering och testning . . . 4
2.2.7 Dokumentation . . . 5

3 Utvecklingsplattformen .net **7**
3.1 Inledning . . . 7
3.1.1 Ett grundläggande exempel . . . 7
3.1.2 Standardisering av .net . . . 8
3.2 CLR . . . 8
3.2.1 Virtuella maskiner . . . 8
3.3 Hur kompileras kod och till vad? . . . 9
3.3.1 Metadata . . . 10
3.3.2 Assemblies . . . 10
3.3.3 Versionshantering . . . 12
3.3.4 Styrd och ostyrd kod . . . 12
3.4 Intermidiate Language (IL) . . . 12
3.4.1 Instruktioner . . . 13
3.4.2 Verifiering . . . 13
3.4.3 Exekvering av ett .Net program i Windows . . . 13

3.5 Common Type System 14
3.5.1 Referenstyper 14
3.5.2 Värdetyper 15
3.5.3 Members 16
3.5.4 Accessability 17
3.5.5 Typsäkerhet 17
3.5.6 Common Language Specification 17
3.6 Prestanda och jämförelser med andra exekveringsmiljöer 18

4 Genomförandebeskrivning **21**
4.1 Analys 21
4.1.1 Driftdata 21
4.1.2 Felkoder 22
4.1.3 Hur kommer datan till Scania? 23
4.1.4 Analys av SVAP-databasen 23
4.1.5 Analys av de tidigare versionerna 24
4.2 Behovsinventering 24
4.2.1 NMEB 24
4.2.2 OPC/Retarder-utvecklare 25
4.2.3 Slutsatser 25
4.3 Prototyp av användargränssnitt 25
4.4 Val av verktyg 25
4.4.1 Webapplikation i ASP.net 26
4.4.2 Windowsapplikation 26
4.4.3 Grafritare 27
4.4.4 Slutgiltigt val 27
4.5 Implementation 27
4.6 Utvärdering/Test 28

5 Resultat **29**
5.1 Databas 29
5.1.1 Fordonsdata 29
5.1.2 Driftdata 30
5.1.3 Säkerhet och accesslösning 31
5.2 Applikation 32
5.2.1 Sökning av fordon 33
5.2.2 Sökresultat 34
5.2.3 Fördelning 34
5.2.4 Enskilt fordon 35

6 Sammanfattning **41**
6.1 Möjliga förbättringar och optimeringar 41
6.1.1 DCDA 41
6.1.2 Beräkningar 41
6.2 Önskemål för framtida versioner 42
6.2.1 Rapportfunktion 42
6.2.2 Andra sökmöjligheter 42
6.2.3 EcuID 42
6.2.4 Grafer med populationer och enskilda chassin 42
6.2.5 Felkoder (DTC) 42
6.2.6 Fältprovbilar 43
6.3 Lösningens Begränsningar 43
6.3.1 Angående omräkning av intervall för matriser/vektorer 43

7 Tack **45**

Referenser **47**

A Förkortningar **51**
A.1 Scaniaspecifika 51
A.2 .net 51
A.3 Övriga 51

Figurer

3.1 Översikt över de olika kompileringstegen för CLR 9
3.2 Exempel på Assemblys och länkade moduler 11
3.3 Uppdelningen av datatyperna i CTS 14
3.4 Bild över sambandet mellan CTS och CLS 18

4.1 Screenshot över en vektor representerad som ett histogram 22
4.2 Översikt över komponenterna i en Webapplikation 26
4.3 Komponenterna i en Windowsapplikation 26

5.1 Översikt över databaserna 30
5.2 Vy över databastabellerna med fordonsdata 31
5.3 Vy över databastabellerna med driftdata 32
5.4 Bild över sökfönstret 33
5.5 Screenshot på sökresultatsfönstret 35
5.6 Bild på en driftdatavektor representerat som ett histogram 36
5.7 Bild på matris-presentation 37
5.8 Vy över fördelningstaben med två leds fördelning 38
5.9 Bild över sökresultatfönstret vid chassisökning 39

Tabeller

3.1 Inbyggda fördefinierade typer i CTS 15

4.1 Exempel på en vektor 22
4.2 Jämförelse mellan en Webapplikation och en windowsapplikation 28

Kapitel 1

Introduktion

Denna examensarbetesrapport är en del av ett examensarbete för Civilingenjörsprogrammet i Teknisk Datavetenskap vid Umeå Universitet. Examensarbetet utfördes under hösten 2005 på Scania CV AB, Södertälje. Institutionen för Datavetenskap, Umeå Universitet, är ansvariga för kursen.

1.1 Bakgrund

Scania är ett av världens ledande tillverkare av lastbilar och bussar såväl som marin- och industrimotorer. Scania har sitt huvudkontor i Södertälje där även tillverkning och utveckling sker. Andra fabriker finns bland annat i Katrineholm, Falun och Zwolle (Nederländerna). Scania är ett internationellt företag med etablerad marknad i Europa, Latinamerika, Asien, Afrika och Australien.

I varje modernt fordon, oavsett om det är en bil eller en lastbil, finns ett antal styrenheter som har till uppgift att styra olika delar av fordonet. Styrenheterna styr sådant som motor, bränsleinsprutning, växellåda etc. Dessa styrenheter har givare och loggar ett antal olika parametrar om hur bilen körs, motorvarvtal, temperaturer etc. Dessa parametrar kallas för driftdata. Olika styrenheter loggar olika data och på olika sätt. Viss data sparas i vektorer eller matriser där varje cell representerar ett mätintervall, andra data kan vara räknarvärden.

Driftdatan används av verkstäderna när de servar fordon och kan då användas för att felsöka dessa och ge svar till kunder varför just deras fordon beter sig som de gör. En del av den loggade driftdatan ger information om hur bilen har körts, vilka varvtal, hastigheter etc. som bilen har haft under sin drifttid och för utvecklarna på Scania kan denna information bland annat användas till att förbättra bränsleförbrukningen i kommande lastbilar.

En del av de loggade driftdatavariablerna är klassade som hemliga av Scania och är enbart avsedda för speciella grupper internt. Dessa variabler är naturligtvis inte tillgängliga ute på verkstäderna.

VERA (*Vehicle Electronic Reconfiguration Application*) är namnet på ett projekt som Scania driver. VERA innehåller flera del-projekt och applikationer. Ett av dessa projekt är att bygga upp en databas där driftdatan för alla Scania-producerade fordon i drift ska lagras. Detta projekt (och dess databas) kallas SVAP (*Scania Vechicle Analysis Portal*) Ett av målen med projektet

går ut på att driftdatan ska lagras automatiskt så fort en bils driftdata läses ut på en verkstad. Det ska inte krävas någon insats från mekanikerna för att datan ska skickas till Scania.

1.2 Sammanhang

Att kunna nyttja driftdatan genom att titta på bilpopulationer istället för enskilda fordon kan ge svar på många frågor rörande fordonens beteenden. Denna information kan då användas för att anpassa och skräddarsy fordon för viss tillämpning, exempelvis kan fordon som körs i vissa klimat bete sig likartat och uppvisa samma typer fel. Andra tillämpningsområden kan vara att jämföra ett enskilt fordon mot andra fordon med liknande konfiguration för att se avvikelser och utifrån dessa dra slutsatser rörande fordonets tillstånd.

Behovet av ett analysverktyg för driftdata från fordonspopulationer har funnits under en längre tid hos Scania. 1999 utvecklade Scania en relationsdatabas i Access där driftdata lagrades tillsammans med information om fordonen. På så sätt kunde populationer av fordon fås fram och medelvärdet av deras driftdata beräknas. Driftdatan kunde visualiseras med grafer tack vare kopplingar till Excel. En nackdel med denna applikation var att programmet inte tog hänsyn till att driftdatan kunde komma från olika versioner av samma styrenhet. Driftdatavariablerna var de samma men mätintervall kunde skilja sig åt vilket medförde att för att få ut medelvärden krävdes det att datan bearbetades i mer avancerade datorprogram som Matlab.

Under 2003 utfördes ett examensarbete på motorutvecklingsenheten NMEB på Scania med målet att skapa ett verktyg som skötte alla beräkningar, visualiserade dessa samt kunde spara undan driftdatan i ett matlab-kompatibelt format. Programmet utvecklades i Matlab och var kopplat till en accessdatabas som manuellt uppdaterades med filer från verkstäderna. Problemet med denna version var att databasen aldrig fylldes med särskilt mycket data. För att filerna skulle ta sig från verkstäderna till Scania krävdes det att de mailades, något som inte skedde regelbundet.

1.3 Indelning

Rapporten är indelad i ett antal kapitel. I kapitel 2, problemspecifikation, defineras uppgiften och den tänkta utvecklingmetoden. I Kapitel 3 analyseras plattformen .net djupare. Kapitel 4, genomförandebeskrivning, redovisar hur arbetet genomförts i kronologisk ordning. Resultatet finns beskrivet i kapitel 5 med en enkel genomgång av produktens funktioner. I kapitel 6 analyseras och utvärderas arbetet och produkten. Dessutom ges förslag ges på eventuella förbättringar till senare versioner. I kapitel 7 tackas personer som varit till hjälp och stöd under arbetets gång. I appendix finns en förtecking över förkortningar som används i rapporten.

Kapitel 2

Problembeskrivning

2.1 Mål

Målet är att skapa en applikation för att analysera driftdata från populationer av fordon. Denna applikation ska vara tillgänglig genom någon form av grafiskt användargänssnitt. Applikationen ska hämta sin data från VERAs databas samt klara av att hantera alla typer av driftdata.

En utredning ska göras ifall det nuvarande verktyget (matlab-applikationen) är kraftfull nog för att hantera datan från VERAs databas, kraftfull i avseende på responstider och funktionalitet, eller om ett nytt verktyg måste utvecklas.

Det färdiga programmet kommer att användas internt på Scania av personer som sysslar med utveckling och dessa ska också få möjlighet att ställa krav på applikationen. Utifrån krav, valt verktyg och examensarbetets tidsram ska sedan en applikation konstrueras till en färdig användbar produkt. Om andra grupper på Scania har intresse av programmet ska deras behov tillgodoses beroende på om tid finns.

Möjligheterna för att denna applikation kan presentera andra typer av data som finns tillgänglig i databasen ska utredas och om möjligt ska de inkluderas i det färdiga programmet. En sådan undersökning ska inkludera vilka andra typer av data som finns tillgängliga från styrenheterna, om de finns i databasen och hur det är önskvärt att de presenteras.

2.2 Metod

Arbetsgången för programutvecklingen följer följande ordning.

1. Analys
2. Kravinsamling
3. Prototyputveckling(Användargränssnitt)
4. Val av verktyg och utvecklingsmiljö
5. Implementation

6. Utvärdering
7. Dokumentation

Dessa sju steg är inte helt avgränsade från varandra men representerar sju distinkta delar som i sig avgränsar arbetet.

2.2.1 Analys

De olika databaserna och den tidigare applikationen måste först och främst utvärderas och analyseras så att den kommande behovsinsamlingen blir meningsfull.

Den äldre applikationen hade utvecklats, utvärderats och anpassat efter en viss användargrupp (Motorutvecklingsenheten NMEB) och kan därför användas som utgångspunkt för den nya applikationen. Det som fungerade bra och var bra funktionalitet för utvecklare på motorutvecklingsenheten behöver nödvändigtvis inte vara bra för de andra styrsystemsutvecklarna.

2.2.2 Krav och behovsinsamling

Intervjuer görs med potentiella användare. Med utgångspunkt från den tidigare versionen samt information som framkommit under analysdelen diskuteras olika tillämpningar och önskemål. Dessa krav måste sedan utvärderas ytterligare och ses över ifall de ens är möjliga.

I slutet av denna fas ska en kravspecifikation sammanställas med de funktioner som ska implementeras.

2.2.3 Prototyp för användargränssnittet

Utifrån kravspecen byggs en prototyp till ett användargränssnitt som sedan kan utvärderas och revideras med hjälp av användarna.

2.2.4 Val av utvecklingsmiljö och verktyg

Vilket utvecklingsverktyg som ska användas och vilka övriga program som kan behövas för applikationen utreds i denna fas. Eventuellt ska tester göras mellan olika verktyg för att avgöra dess lämplighet.

2.2.5 Implementation

Programmet implementeras enligt specifikationen.

2.2.6 Utvärdering och testning

Programmet demonstreras för och testas av användarna. Utifrån dessa tester klargörs vad som saknas, om nya krav finns och om dessa enkelt kan förverkligas. Vidare bör användarna få möjlighet att rapportera buggar i programmet.

2.2.7 Dokumentation

Ett antal dokument måste skrivas, dels användarhandledning men även någon form av systembeskrivning så att andra lätt kan sätta sig in i programmet och modifiera det vid behov.

Kapitel 3

Utvecklingsplattformen .net

3.1 Inledning

Utvecklingsplattformen .net, uttalas dotnet, utvecklades av Microsoft med målet att vara plattformsoberoende såväl som språkoberoende. Med utvecklingsplattform menas ett ramverk och en exekveringsmiljö mot vilken programmen körs. Ett .net-program ska kunna exekveras på alla maskiner som har installerat plattformen.

Utvecklingsplattformen består av två delar, CLR (*Common Language Runtime*) vilken är själva infrastrukturen och FCL (*Framework Class Library*) vilken är ett antal klassbibliotek som är tillgängliga för utveckling i de språk som har stöd för .net. Med stöd menas de språk som har kompilatorer för CLR [22].

.net baserar sig på en öppen standard, standardiserad av ECMA(*European Computer Manufacturers Association*), och .net är egentligen en implementation av denna standard. .net i allmänhet och CLR i synnerhet består utav ett antal komponenter som har skilda namn i standarden och i Microsofts implementation. Plattformen .net är ett omfattande område och detta kapitel är tänkt som en översikt över arkitekturen med fokus på de detaljer som gör .net språk- och plattformsoberoende. I slutet av kapitlet finns också ett avsnitt om jämförelser med JVM(*Java Virtual Machine*) som är en liknande lösning.

3.1.1 Ett grundläggande exempel

För att demonstrera vissa delar av plattformen kommer ett enkelt programexempel användas. Det är ett enkelt program som skriver ut strängen ”Hello World” i en konsol. Programmet är skrivet i C#[1] och har följande kod:

```
Using system;

Public class Example
{
        public static void Main(string[] args)
        {
                Console.WriteLine("Hello");
                Console.WriteLine("World");
        }
}
```

[1] C# är ett objektorienterat programmeringspråk som till största delen har identisk syntax som Java [25].

3.1.2 Standardisering av .net

I augusti 2000 presenterade Microsoft tillsammans med Hewlett-Packard och Intel .net för standardiseringsorganet ECMA [11, 22, 5]. ECMA tillsatte ett antal arbetsgrupper dels för att standardisera CLI (*Common language Infrastructure*, vilket är en delmängd av CLR) och dels för att standardisera Microsofts programmeringspråk C# (uttalat c sharp).

ECMA släppte en första version av standarden för CLI i december 2001 och sände denna vidare till ISO vilka stadsfäste standarden 2003 [11].

Genom att släppa standarden öppen har vem som helst rätt att utveckla kompilatorer för vilka maskiner som helst och så länge de följer standarden är program skrivna i en miljö fullt körbara i andra miljöer. Ett antal olika projekt för att skapa exekveringsmiljöer i andra operativsystem än windows pågår [4, 12] och många programmeringsspråk har numera CLR-kompatibla kompilatorer.

3.2 CLR

CLR är en språkoberoende exekverings-miljö som sköter minneshantering, trådar, kompilering och kommunikation [17]. Förutom att vara språkoberoende strävar CLR också efter att vara plattformsoberoende.

CLR möjliggör sitt plattformsoberoende genom att använda sig av flera steg av kompilering, se figur 3.1.

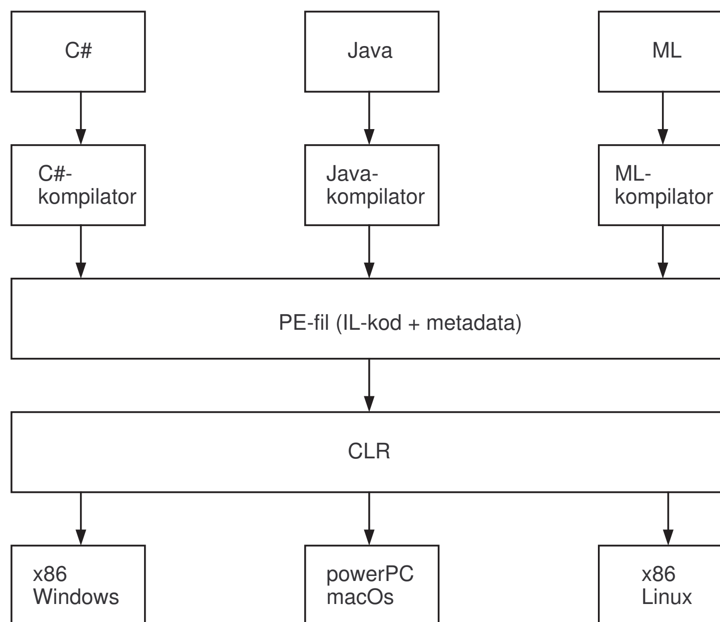
Kod skriven i något programmeringsspråk kompileras ned till ett mellanliggande och universellt medium kallat MSIL (*Microsoft Intermidiate Language*). Denna MSIL-kod kompileras ned till maskinnivå under exekvering med hjälp av en sk. JIT-kompilator (*Just In Time*). Vid kompilering sparar kompilatorerna MSIL-koden i en sk. PE-fil (*Portable Executable format*), vilket är ett språkoberoende format. [17]

3.2.1 Virtuella maskiner

Konceptet med flerstegskompilering är inget nytt utan härrör från 70-talet [7]. Tanken var att ha en språkbunden

”frontend” som producerade kod till en maskinberoende ”backend”. En av fördelarna med att kompilera i två steg och inte direkt ned till maskinkod var att istället för att behöva N * M kompilatorer för att kompilera kod från N språk till M maskiner behövdes istället bara N + M kompilatorer [7, 18]. Flertalet implementationer av denna teknik baserades på abstrakta stackmaskiner. I dessa fall blev koden efter att ha kompilerats ned till abstrakt maskinnivå mycket mer kompakt och lättare att ladda in i den tidens datorer, som hade begränsat minne. En annan fördel var lättheten att portera kod och möjligheten att emulera andra maskinmiljöer [7].

JVM (*Java Virtual Machine*) är en plattform som lanserades i slutet av 90-talet av Sun Microsystems. JVM är i grunden inte språkoberoende men väl plattformsoberoende. Java-kod kompileras ned till ”java bytecode” vilket är JVMs mittformat. Bytecoden som paketeras i Java class-filer kompileras vid körning av en JIT-kompilator. Precis som sina föregångare är JVM stackbaserad. JVM har även skräphanterare och är typsäker [25]. Begreppet typsäker kommer att beskrivas mer senare.



Figur 3.1: Översikt över de olika kompileringstegen för CLR

Förutom rena virtuella exekveringsmiljöer har ett flertal varianter för att skapa generell portabel kod genomförts. Med detta menas att objekt skapade genom ett programmeringsspråk då ska kunna användas i ett annat. IBM skapade SOM (*System Object Model*) vilken använde samma gränsyta som CORBAS IDL (*Interface Defintition Language*. SOM var en språkoberoende exekveringsmiljö med syfte att möjliggöra portabilitet men lyckade aldrig slå igenom ordentligt [8]. Microsofts COM (*Common Object Model*) är en lösning med liknande syften men begränsar sig till att vara en specifikation för hur gränssnitt och objektinstanser ska bete sig i fråga om returvärden. COM kräver också att ett antal grundläggande funktioner ska finnas med i gränssnitten [7].

3.3 Hur kompileras kod och till vad?

Vid kompilering av programkod med hjälp av en CLR-kompatibel kompilator skapas något som heter moduler. En modul är en byggsten som kan användas i en assembly. En assambly är i sin tur ett körbart program eller en del av ett större system.

Modulen innehåller flera olika delar. Dels innehåller modulen MSIL-kod men även en CLR-header vilken innehåller ett antal flaggor och även pekar på var i MSIL-koden exekveringen ska starta dvs. vars main-metoden har sin början. I övrigt innehåller modulen tabeller som beskriver de typer som definierats eller refererats till. Dessa tabeller kallas för metadata.

3.3.1 Metadata

Metadata finns alltid tillsammans med MSIL-koden och gör att modulerna är självständiga enheter som inte behöver externa komponenter utanför CLR för att användas. All information som behövs för att använda modulen finns i metadatan. På så sätt behövs inga typbibliotek eller IDLs (*Intermidiate Definition Libraries*) [22]. Metadatan är också till hjälp för att verifiera typsäkerhet, tar bort behovet av headerfiler och används för att genomföra serialisering av objekt.

Metadatans tabeller kan delas in i tre grupper, definitionstabeller, referenstabeller och manifesttabeller. De två förstnämnda finns i alla moduler medan manifesttabeller används av assemblyn [22].

Definitionstabeller: Är tabeller med index över modulens alla definierade typer, metoder, parametrar och events. Det finns alltid en tabell ModuleDef som innehåller identifikationsinformation för modulen såsom filnamn, versionsinformation, etc.

Referenstabeller: Innehåller tabeller med index till alla assemblys, moduler, typer och annat som refereras till i modulen.

Manifesttabeller: Innehåller referenser till alla resurser som används samt information om den aktuella assemblyn.

3.3.2 Assemblies

En modul utgör i sig inget körbart program utan måste paketeras i en assembly för att kunna exekveras. En assembly är en fysisk gruppering och utgör den minsta enheten som CLR arbetar med. En assembly kan bestå av en eller flera moduler paketerade tillsammans med eventuella resursfiler som kan vara bilder, xml, etc. En Manifesttabell sparas också i assemblyn. Manifestet ligger som en tabell i en moduls metadata och innehåller information om länkade resurser.

Ett exempel på en sådan länkning är följande utdrag från vårt exempelprograms manifest

```
AssemblyRef #1
-------------------------------------------------------
        Token: 0x23000001
        Public Key or Token: b7 7a 5c 56 19 34 e0 89
        Name: mscorlib
        Major Version: 0x00000001
        Minor Version: 0x00000000
        Build Number: 0x00001388
        Revision Number: 0x00000000
        Locale: <null>
        HashValue Blob:
        Flags: [none] (00000000)
```

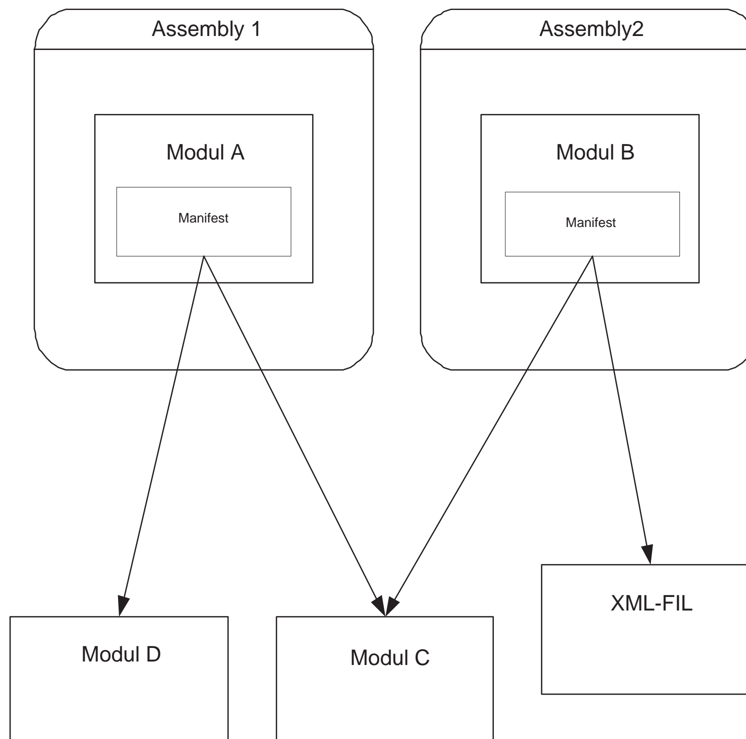
Informationen i manifestet hjälper assemblyn att hålla reda på all nödvändig info. I exemplet är det referensen till mscorlib som listas.

Versionsnummret består av fyra värden (major version, minor version, build number och revision number). De första två representerar den publika versionsinformationen. Assemblyn ovan är version 1.0. Detta sätt att beteckna versioner är Microsofts rekommenderade metod och de två sista värdena används internt för att skilja olika kompilerade versioner från varandra [22].

En assembly har bland annat följande funktioner:

- Sätter upp en säkerhetsram för interaktion med resurser på maskinen den exekveras på. Alla program har inte samma rättigheter vid körning, exempelvis kan kod som laddats ned från Internet ha begränsade rättigheter i jämförelse med program som skapats lokalt.
- Är en yttre gräns för räckvidden för typer. Datatyper och klassobjekt har en naturlig gräns för hur långt metoder och funktioner kan påverka (se 3.5.4).
- Utgör den minsta enheten för versionshantering (se 3.3.3).
- Utgör den minsta enheten för en körbar applikation. En assembly kan vara ett körbart program.

Speciella verktyg används för att kombinera moduler till assmblies men många kompilatorer gör detta automatiskt. Då flera moduler används i assemblyn kan dessa vara skrivna i olika programmeringsspråk, detta är transparent för assemblyn.



Figur 3.2: Exempel på Assemblys och länkade moduler

I figur 3.2 finns två assemblys (1 och 2). Modul A tillhör assembly 1 och har dess manifest. På samma sätt tillhör modul B Assembly 2 och innehåller dess manifest. Modul D länkas in till Assebly 1 och finns refererat i manifestet hos Modul A. XML-filen refereras i Modul Bs manifest. Modul C finns refererat av både Assembly A och Assembly B. Då Modul C inte innehåller något manifest kan den användas av flera olika assemblys. Vissa kompilatorer (exempelvis C#s) skapar automatiskt varje modul som en egen assembly (dvs ger den ett manifest) och speciella verktyg måste då användas för att koppla ihop flera moduler till ett gemensamt assembly [22].

3.3.3 Versionshantering

Vid exekvering av ett program laddas inte nödvändigtvis alla assemblys till minnet på en gång. Resurser som inte används ofta kanske inte ens finns sparade på användarens maskin utan laddas ned över internet vid behov. Denna decentralisering av kod och programresurser gör att versionshanteringen blir väldigt viktig. Vissa komponenter i ett program kan exempelvis vara tillverkade av externa utvecklare och när de sedan kommer med en ny version eller en uppdatering är det inte säkert att den uppdaterade komponenten är bakåtkompatibel med den gamla versionen.

Windows har haft detta problem länge med sina dll-filer (*dynamic link librarys*) och problematiken med versionshanteringen brukar kallas ”DLL Hell” [23, 15].

Versionshanteringen med assemblys försöker lösa detta genom att spara ett versionsnummer för de refererade assemblys i manifestet. CLR försöker vid körning att ladda en så matchande version av assemblyn som möjligt. Om de tillgängliga versionerna inte är kompatibla uppstår fel ändå [23].

3.3.4 Styrd och ostyrd kod

De moduler som tidigare nämnts bör egentligen heta styrda moduler då dessa kräver CLR för att kunna användas. Det är dock inte nödvändigt att styrd kod skapas vid kompileringen. För vissa språk ex. C++ finns möjligheten att skapa ostyrd kod, vilken inte kräver CLR för att användas. Det är även möjligt att låta delar av ett program bestå av ostyrd kod och delar av styrd. Nackdelen med ostyrd kod är att den inte är porterbar på samma sätt som styrd kod, då den är kompilerad för en viss maskin, samt att den inte kan utnyttja finesser i CLR såsom skräphantering under körning och typverifiering [29, 22].

3.4 Intermidiate Language (IL)

IL är den maskinoberoende kod som CLR-kompilatorerna skapar. Detta är ett objektorienterat språk och innehåller 220 olika instruktioner [18, 5].

IL är stackbaserat och använder sig inte av några register. Instruktioner lägger operander på stacken och hämtar resultat från denna.

ILs kod består av assembler instruktioner och direktiv till assemblatorn. Alla rader med direktiv börjar med ”.” och följaktligen är alla andra rader instruktioner (sanning med modifikation då kommentarer också kan användas).

Funktioner i IL börjar med direktivet `.method` följt av returtypen och sedan parametrar inramade i parenteser. Själva funktionskroppen är avgränsad med hakparenteser. Ex.

```
.method void funktionsnamn()
{
        .entrypoint
        instruktioner...
}
```

Kommandot `.entrypoint` måste alltid finnas med i någon av funktionerna i assemblyn men bara i en av dessa. Entrypoint betecknar startfunktion för programmet på liknande sätt som en main-funktion fungerar i C/C#/Java.

Klasser deklareras med direktivet .class följt av attribut för klassen och sist klassnamnet. Assemblies deklareras på liknande sätt med `.assembly`.

3.4.1 Instruktioner

ILs instruktioner kan variera i storlek sinsemellan men upptar ett exakt antal bytes. CLR vet beroende på instruktionskod hur lång en specifik instruktion är och vet därmed var i byteströmmen nästa instruktion börjar [5].

IL har generella instruktioner som inte är typberoende. Istället beror resultatens typ på operanderna. IL opererar enbart på ett antal bestämda typer men har möjlighet att hantera andra typer genom konvertering [7].

3.4.2 Verifiering

För att avgränsa program och processer från varandra verifierar CLR IL-koden innan den kompileras ned till maskinnivå. Verifieringen kontrollerar att inget minne accessas utanför processens avsatta minnesområde, att programmet inte försöker komma åt eller förändra variabler utanför sitt egna adressområde.

3.4.3 Exekvering av ett .Net program i Windows

Vi återvänder till det tidigare programexemplet för att se hur dess MSIL-kod ser ut och exekveras.

Efter kompilering med C# kompilatorn skapas följande MSIL-kod (Här har programmet ILDasm.exe, IL Disassembler, använts för att få fram IL-koden):

```
.method private hidebysig static void  Main(string[] args) cil managed
{
  .entrypoint
  .custom instance void [mscorlib]System.STAThreadAttribute::.ctor() = ( 01 00 00 00 )
  // Code size      21 (0x15)
  .maxstack  1
  IL_0000:  ldstr      "Hello "
  IL_0005:  call       void [mscorlib]System.Console::WriteLine(string)
  IL_000a:  ldstr      "World!"
  IL_000f:  call       void [mscorlib]System.Console::WriteLine(string)
  IL_0014:  ret
} // end of method Class1::Main
```

Först deklareras main-metoden med ett antal flaggor. Returvärdet `void` står alldeles innan funktionsnamnet. Efter argumenten finns direktiven `cil managed`. Det visar att koden är styrd (managed) och behöver CLR för att exekveras.

Själva kodstycket börjar efter direktivet `.maxstack`. Denna anger det maximala antalet objekt som kommer att befinna sig på stacken samtidigt.

I det följande kodstycket som inleds med instruktionen ldstr kan tydligt ILs stackbaserade beteende skådas. Med funktionen ldstr läggs en sträng på Stacken, eller en referens till denna om strängen skulle vara ”boxad” till en referenstyp (se nedan), och med kommandot call anropas en metod WriteLine. WriteLine ärvs från `[mscorlib]System.Console` och hämtar sitt argument (string) från stacken.

Vid exekvering av programmet sker följande. Initialt anropas funktionen _CorExeMain vilken initierar CLR och tittar i CLR-headern efter vart i IL-koden som mainfunktionen (.entrypoint) finns. IL-koden innehåller två stycken anrop till Console::WriteLine. När det första anropet påträffas startas JIT-kompileringen och den går till på följande sätt:

- Metadatan för det aktuella metodanropet granskas, verifieras och minne allokeras.
- Metoden kompileras ned till maskinberoende kod som sparas i det allokerade minnesblocket
- Metadatan ändras så att referenser till metoden hänvisas till minnesblocket med kompilerade maskinberoende instruktioner.

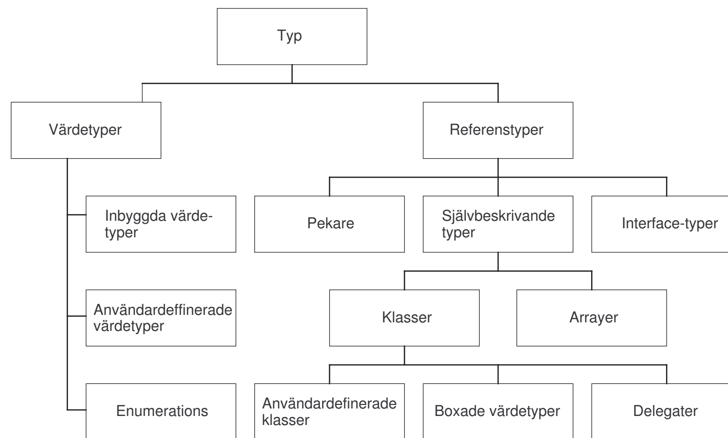
Efter dessa steg återvänder exekveringen till IL-koden igen. Andra gången WriteLine anropas behövs ingen JIT-kompilering ske då den redan är utförd och metodens metadata hänvisar den till det maskinberoende kodblocket.

3.5 Common Type System

CTS (*Common Type System*) är en generell standard för datatyper för CLR som ska fungera tillsammans med alla typer av språk, objektorienterade såväl som funktionella. Datatyperna i CTS kan delas upp i två kategorier. Referenser och värden. Referenser är typer såsom klasser, arrayer och interface medan värden är grundläggande typer såsom integers och booleans. En överblick över uppdelningen syns i figur 3.3.

3.5.1 Referenstyper

Referenser kan bestå av flera andra referenser eller värden. Referenser allokeras alltid på heapen vilket kan innebära extra tidsåtgång då minnet måste lokaliseras och eventuellt frigöras.



Figur 3.3: Uppdelningen av datatyperna i CTS

Tabell 3.1: Inbyggda fördefinierade typer i CTS

Namn	Beskrivning
Bool	Booleanskt värde (sant/falskt)
Char	Unicode 16-bitars tecken
Class System.object	Objekt-typ
Class System.string	En unicode-sträng
Float32	IEEE 32-bits float
Float64	IEEE 64-bits float
Int8	Signed 8bits integer
Int16	Signed 16bits integer
Int32	Singed 32bits integer
Int64	Signed 64bits integer
Native int	Signed integer, native size
Native unsigned int	Unsigned integer, native size
Typedref	Pekare med runtime typ
Unsigned int8	Unsigned 8bits integer
Unsigned int16	Unsigned 16bits integer
Unsigned int32	Unsigned 32bits integer
Unsigned int64	Unsigned 64bits integer

3.5.2 Värdetyper

Värden är datatyper som integers, flyttal och booleans och dessa allokeras oftast på stacken. De kan också allokeras som en del av en referenstyp och allokeras då på heapen. Anledningen till att ha värdetyper är för att öka effektiviteten genom att slippa allokera (och avallokera) från heapen. I CTS finns ett antal värdetyper, även kallade ”Basic CLI types”, vilka är listade i tabellen nedan (se tabell 3.1). Dessa är de grundläggande primitiva datatyperna i CLR. Dock finns det språk som har andra typer som primitiva och de är i de flesta fall hanterade som värdetyper i CTS (även fast de inte tillhör CLS).

Alla värdetyper kan ”boxas” till en referenstyp. Med boxas menas helt enkelt att en referens skapas (liknande pekare i C). Ett exempel på när detta händer är när ett värde läggs till i en dynamisk lista. Fördelen med detta är att dessa boxade värdetyper då omfattas av garbage collectorn.

De olika datatyperna är associerade med varandra genom arv. Alla kan härledas från typen System.Object. System.Object har ett antal grundläggande funktioner som alla typer stödjer. Dessa funktioner är:

- **`Equals:`** jämförelse om två object är lika. Två värden är identiska om de båda refererar till samma minne och ekvivalenta om de har samma värde.
- **`GetHashCode:`** returnerar ett hashvärde för det givna objektet
- **`Clone:`** skapa en kopia av aktuellt objekt

- **`GetType:`** returnerar objektets typ. Denna metod gör det möjligt att alltid ta reda på ett objekts typ vilket möjliggör reflektion[2].

- **`ToString:`** returnerar en textsträng innehållande relevant, beroende på objekttyp, text. Exempelvis skulle en toString som används på en integer returnera dess värde som en sträng.

3.5.3 Members

Hur typen fungerar och vad den innehåller bestäms av dess members. En typ innehåller noll eller flera members. Members kan vara följande:

Konstanter

Konstanter deklareras vid kompilering och kan därefter inte ändras. De upptar inget av programmets minne vid körning utan refereras från metadatan vid behov. De kan inte heller skickas med som parametrar vid funktionsanrop. Primitiva typer kan användas som konstantvärden.

Fält(Fields)

Fält påminner om konstanter men kan ändras under exekvering då deras värde laddas upp i programminnet. Det finns två typer av fält dels ”read-only” och dels ”read/write”. Read-only-fält deklareras i en konstruktormetod men kan inte ändras utanför denna. Read/write-fälten har inga sådana begränsningar.

Konstruktorer

Konstruktorer används för att instansiera typer. Alla referenstyper måste ha en konstruktor för att möjliggöra verifiering av CLR

Metoder

Metoder är operationer som påverkar den aktuella instansen av typen eller någon annan instans av samma typ [5]. CTS stödjer metoder som vissa programeringspråk inte gör. Överlagring och parameterlistor är två sådana exempel. Med parameterlistor menas att antalet parametrar kan variera och inte är fixt.

Metoder kan ha samma namn så länge som parametrarna eller returvärdena skiljer sig åt.

Events

Events eller händelser möjliggör andra objekt att reagera på förändringar hos ett objekt. Detta görs genom att de ”prenumererar” på ett event. Att använda sig av händelser vid programmering underlättar arbetet med exempelvis användargränssnitt.

[2] Reflektion innebär att objektet kan ta reda på och ändra sin egen eller programmets struktur [10].

3.5.4 Accessability

En viktig egenskap som medlemmar av instansierade typer har är dess synlighet eller räckvidd. Detta sätts i en flagga vilken avgör vilka som får använda sig av exempelvis en metod eller läsa ett fält. Följande värden är möjliga:

Private: Enbart metoder i samma klass har tillgång till medlemmen.

Family: Alla härledda typer har tillgång till medlemmen oavsett om de befinner sig i samma Assembly eller inte. Ofta betecknad som protected i programmeringsspråk.

Family and Assembly: Samma som ovan med begränsningen att den härledda typen måste vara i samma assembly .

Assembly: Alla is assemblyn har tillgång till medlemmen.

Family or assembly: Alla härledda typer samt alla medlemmar i assemblyn har tillgång till den.

Public: Alla typer i alla assemblys har tillgång till medlemmen.

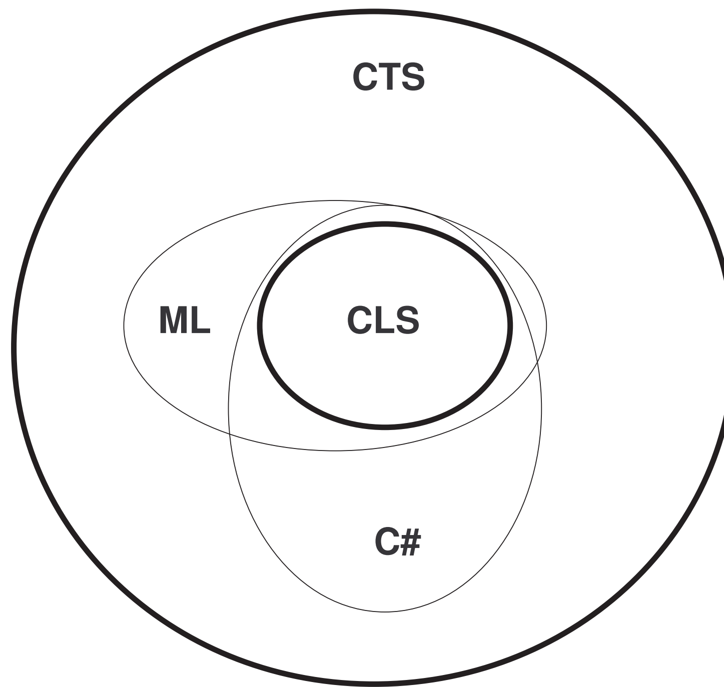
3.5.5 Typsäkerhet

Både objekt och värden är typsäkra. Med det menas att det inte går att utföra operationer utanför typernas specifikation eller anropa minnesområden som inte är skrivna till etc. Dessa specifikationer eller regler bestäms av något som kallas för kontrakt. Typer har skrivit på dessa kontrakt och följer dess regler. För att validera detta används en del av kontrakten som kallas för signaturer.

3.5.6 Common Language Specification

CLS (*Common Language Specification*) är en delmängd av CTS och är den del av CTS som alla CLR-kompatibla språk måste stödja [9]. Anledningen till detta är att de olika programmeringsspråken kan vara synnerligen olika och för att göra det möjligt för program skrivna i två olika programmeringsspråk att använda varandras funktioner krävs att bara funktionalitet som båda språken klarar av att hantera används i funktionerna. CLS är just denna mängd av funktioner och om en funktion skrivs med enbart CLS-typer/metoder är det garanterat att den fungerar i alla andra CLR-språk. För att funktionen ska fungera ska typer som funktionen tar som parametrar eller skickar som returvärden vara typer i CLS. Internt kan vilka typer som helst (som funktionens programmeringsspråk stödjer) användas. Som exempel implementerar ML och C# båda en delmängd av CTS men i den delmängden ingår hela CLS. Dock behöver inte MLs och C#s delmängder av CTS vara densamma, se figur 3.4.

Det kan dock vara så att två språk överlappar varandra i implementationen av typer som är utanför CLS. Det är då inget som hindrar ett program skrivet i språk A att anropa funktioner skrivna i språk B som använder typer utanför CLS som parametrar/returvärden.



Figur 3.4: Bild över sambandet mellan CTS och CLS

3.6 Prestanda och jämförelser med andra exekveringsmiljöer

Många jämförelser har gjorts mellan .net och JVM (*Java Virtual Machine*) [25, 7, 17, 16]. Att jämföra dessa två är inte ett slumpmässigt val. Både .net och JVM är plattformsoberoende och använder sig av stack-baserade virtuella maskiner. De är också konkurrenter som Singer uttrycker det om ”. . . for the accolade of the premier next-generation computing platform” [25].

I det stora hela finns inge större prestandaskillnad mellan de två plattformarna. JVM är äldre, och designmålen var inte riktigt de samma som för .net. JVM var från början avsett för att enbart hantera Java-kod. Utifrån den premissen har sedan möjligheterna vidgats. I kontrast var .net konstruerat för att klara av många olika språk och har funktionalitet som klarar av imperativa såväl som funktionella språk.

Singer och Gough har undersökt storleken på den kompilerade koden samt prestandaskillnader på olika benchmarks. Båda undersökningarna utfördes med skilda benchmarks och skilda kompilatorer men visade båda på att det inte råder några större skillnader på någon av områdena [25, 7].

Delamaro menar att båda arkitekturerna stödjer mobil kod men att .nets assemblys är en klumpigare eller mer grovkornig variant än JVMs class-filer. Med mobil kod menas i detta fall att komponenter i ett program inte nödvändigtvis behöver vara samlade. Vissa kan laddas under körning (runtime) och om behov finns hämtas från externa källor, ex. Internet. Denna typ av program kräver dock säkerhetsrutiner och Delmaro anser att .net har mer sofistikerade sådana något som kan bero på att .net är en modernare produkt och har kunnat lära sig från de misstag

som föregångarna (exempelvis JVM) gjort [17]. Samma slutsats gör Paul och Evans i [16] som har undersökt ifall kända säkerhetshål i JVM finns i .net.

Som en slutsats av detta verkar det inte som om .net är särskilt utmärkande prestandamässigt. .Nets möjligheter att låta utvecklarna fortsätta att arbeta i de programmeringsspråk de känner till bäst eller välja programmeringsspråk utifrån tillämpning kan vara en betydande faktor i kampen mellan de två plattformarna. Att .net är backat av Microsoft samt är en öppen standard kan också vara faktorer som talar för .nets fördel.

Personligen anser jag efter att ha både byggt en webapplikation och en windowsapplikation i .net att utvecklingsmiljön är relativt lättanvänd. Att utveckla grafiska användargränssnitt kändes mycket enkelt. De mer avancerade funktionerna i .net är avskärmade från användaren men finns där för de som vill utnyttja dessa. Min gissning är att de flesta användarna inte kommer att bekymra sig över MSIL utan enbart fortsätta att utveckla i sina favoritspråk. Det innebär då att de aldrig kommer att utnyttja all funktionalitet som .net faktiskt erbjuder.

I jämförelserna ovan är .net jämfört med JVM som är en liknande arkitektur. En jämförelse mellan ett .net-program och en traditionell maskinberoende variant är också intressant. Richter har i [22] belyst några av fördelarna med styrd kod gentemot ostyrd. Han nämner visserligen att JIT-kompilering medför en prestandasänkning för exekveringen men påstår också att styrd kod många gånger kan exekvera snabbare än ostyrd. Dessa fall skulle då kunna vara:

- En JIT-kompilator kan känna av och anpassa kompileringen beroende på maskinens tillstånd till skillnad från ostyrd kod som ofta kompileras för den enklaste varianten av den aktuella processorarkitekturen.
- En JIT-kompilator kan kompilera om koden under körning och optimera koden beroende på den tidigare exekveringen.

CLRs JIT-kompilator kompilerar vid behov och sparar den kompilerade koden ifall samma metodanrop sker igen. Detta gör att det initialt blir en overheadtid och för små program kan den vara av vikt. Hur stor denna är skriver inte Richter om utan han koncentrerar sig på de positiva bitarna kring .net.

Kapitel 4

Genomförandebeskrivning

4.1 Analys

De olika delarna av projektet analyserades för att ge en förståelse om deras omfattning och deras olika delar. Databaserna och de tidigare versionerna av programmet undersöktes. Begreppet driftdata utreddes och jämförelser gjordes mellan driftdata från olika styrenheter med avseende på struktur och användande. Andra relaterade diagnosdata studerades också, då främst det som kallas för felkoder. Anledningen till detta var att felkoder också läses ut och skickas till Scania tillsammans med driftdatan.

4.1.1 Driftdata

Driftdata finns för sex olika styrenheter och tre av dessa jämfördes (S6, Opticruise, Retarder). Att det var just de tre styrenheternas driftdata som jämfördes berodde på att det var de tre vars data som först skulle tolkas och lagras i SVAP.

För att analysera dessa användes produktspecifikationer och utläsningar av driftdata. Driftdatan kan generaliseras till fyra olika typer:

Matriser av olika storlekar

Vektorer med olika storlekar

Min/maxvärden vektorer med 2 värden

Räknarvärden enskilda värden

Dock visade det sig att speciellt matriserna och vektorerna kunde skilja sig åt en del mellan de olika styrenheterna. För exempelvis styrsystemet OPC (*Opticruise*) kunde axlarna beteckna olika tillstånd (beskrivna i teckenform) såväl som brytpunkter för histogram. Hos styrsystemet S6 var alla axlar brytpunkter.

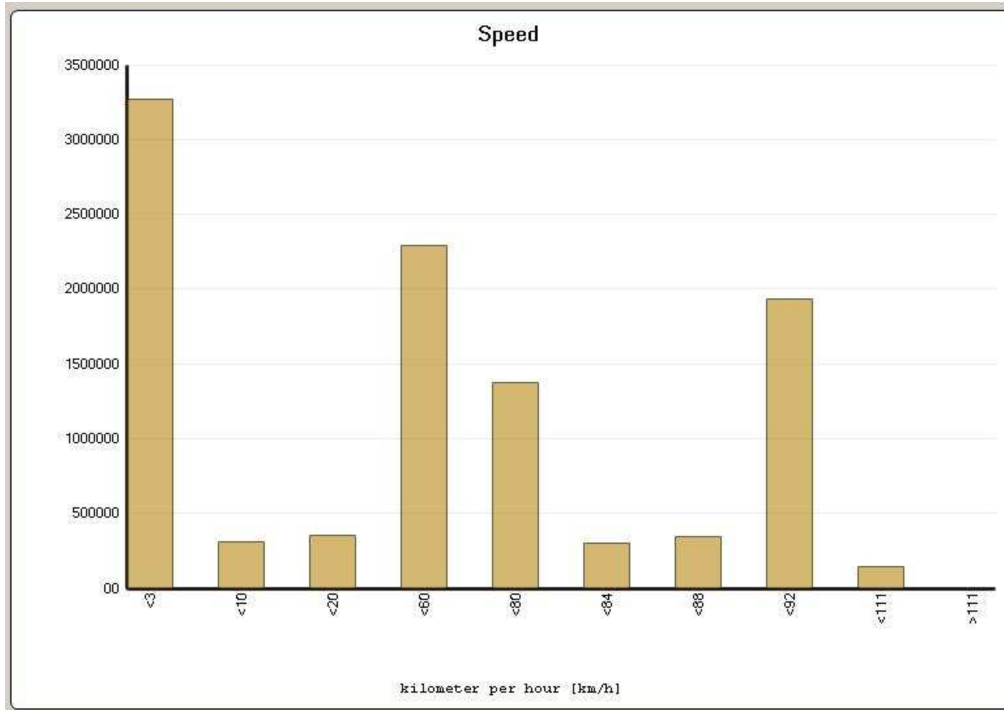
Under drift loggas driftdata med frekvensen 1Hz. Det innebär att varje sekund ökar aktuella räknare för variablerna. För enskilda värden räknas räknaren upp om fordonet uppfyller rätt kriterium medan för vektorer och matriser krävs att givarna ska ge värden inom ett visst intervall.

Tabell 4.1: Exempel på en vektor

<3	<10	<20	<60	<80	<84	<88	<92	<111	>111
3276828	310275	358842	2292469	1379316	307275	350957	1941394	149356	84

Ett exempel på en vektor visas i tabell 4.1. Vektorn är hämtad från ett fordon sålt i Schweiz och visar i vilka hastigheter bilen körts och hur länge.

Representerat som ett histogram ser vektorn ut som i figur 4.1



Figur 4.1: Screenshot över en vektor representerad som ett histogram

I figuren ovan kan vi se att fordonet har stått stilla en stor del av dess körtid (stapeln längst till vänster) och så gott som ingen tid alls körts i hastigheter över 111km/h. Då uppdateringsfrekvensen är 1Hz representerar staplarna följaktligen antalet sekunder som fordonet befunnit sig i givna hastighetsintervall.

4.1.2 Felkoder

En felkod, eller som de också kallas DTC (*Diagnostic Trouble Code*), är en hexadecimal kod som lagras när ett visst fel uppstår i ett fordon. I snapshotfilerna lagras felkoderna tillsammans med antalet förekomster och tid för senaste förekomst.

När en felkod uppstår lagras även något som kallas för ”freeze frame” vilket är en ögonblicksbild av hur bilen ser ut vid tiden för felet. Dessa kan ge svar på om felet är ett reellt fel eller något som uppstått exempelvis på en verkstad när bilen stått stilla.

För att ta reda på mer angående detta intervjuades personer på avdelningen RESA. RESA är en avdelning på Scania som arbetar med diagnostik och systemintegration. RESA har en felkodsdatabas där felkoder från testbilar sparas. Möten hölls med personer på RESA för att utvärdera huruvida denna databas skulle kunna användas för applikationen. Dock hade RESAs databas ingen koppling till SVAP och sättet som data lästes in i felkodsdatabasen var inte heller relaterat.

4.1.3 Hur kommer datan till Scania?

Programmet SDP3 (*Scania diagnos och programmering 3*) kan användas av verkstäderna för att läsa ut och undersöka driftdatavariabler. Med SDP3 finns möjligheten att skapa en sk. ”snapshotfil” vilken innehåller alla driftdatavariabler såväl som felkoder och en del annan information. Snapshotfilen innehåller otolkad data och med det menas att datan är i hexadecimalt format.

Den otolkade datan, snapshotfilen, skickas till Scania där den tolkas och lagras i VERAs databas. Målet är att snapshotfilerna skickas automatiskt till Scania utan att verkstadsarbetarna behöver göra något själva dock är inte verkligheten så. Att skapa en snapshotfil med hjälp av SDP3 tar flera minuter är något som verkstäderna helst inte gör. Förhandlingar pågår för att enas om hur ofta snapshotfilerna ska läsas ut och skickas till Scania. Det finns också planer på att eventuellt låta utläsningen ske trådlöst vilket skulle underlätta för verkstäderna.

4.1.4 Analys av SVAP-databasen

SVAP (*Scania Vehicle Analysis Portal*), är den databas där all driftdata kommer att lagras för Scanias fordon. Enligt direktiv ska driftdatan lagras i databasen i 10 år och alltså måste systemet kunna hantera stora datamängder.

SVAP är under utveckling under ett projekt kallat VERA och ska enligt planering vara färdig med hanteringen och lagring av driftdata från styrsytemen S6, opticruise och Retarder under hösten/vintern 2005/2006.

För att få klarhet rörande vilken data som finns tillgänglig och inte i SVAP intervjuades medlemmar i VERA-projektet. Efter detta kunde datan sorteras in i tre grupper: det som redan finns i SVAP, det som kommer att finnas inom en snar framtid och det som ska implementeras någon gång om behoven finns.

Tolkad data som finns i SVAP

I denna kategori finns all data som redan nu finns tillgänglig och kan användas av den kommande applikationen. Denna data har en specifikation för hur datan ska lagras och de program som ska tolka datan är färdiga (deras kod är fryst). Vid projektets början visade det sig att det fanns mycket lite som var färdigt i SVAP och endast vektorer för styrsytemet S6 fanns tillgängligt. Under arbetets gång kommer mer och mer data att placeras in i denna kategori.

Data som ska läggas till i SVAP inom en nära framtid

Data som planeras att implementeras under hösten 2005 finns med i denna kategori. Denna data kan tas med i applikationen. Om någon förändring sker under arbetets gång kan den tas hänsyn till. Med förändring menas ändring av den planerade strukturen och tabellerna i databasen. I denna kategori kunde all driftdata för S6, OPC och RET placeras initialt.

Data som kanske ska implementeras

Data som inte har struktur eller specifikation kan inte ses som tillgänglig för applikationen. Dessa data saknar lagringsformat och tabeller i SVAP. Felkoder, freeze frames och EcuID är data som hamnar i denna kategori. Om det under kravinsamlingen skulle visa sig att någon av dessa datatyper är väldigt viktig finns möjligheten att föreslå ändringar i planeringen för SVAP.

4.1.5 Analys av de tidigare versionerna

Båda de tidigare versionerna fungerade som så att initialt utfördes en populationsökning bland fordonen där användaren fick göra en sökning på alla fordon. Sökningen förfinades genom att välja alternativ i olika rullgardinsmenyer. Sökalternativen var exempelvis produktionsort, motortyp, marknad (vilket land fordonet var sålt i) etc. Vidare stödde sökningen även enklare booleanska kombinatorer (AND och OR) och reguljära uttryck.

För den sökta populationen går det sedan att välja variabler att titta närmare på. Dessa variabler är driftdata av typen vektorer, matriser, enskilda värden och min/max-värden. När en variabel väljs beräknar programmet medelvärdet hos denna variabel och presenterar den. För matriser och vektorer ges även en grafisk representation med stapeldiagram.

Den tidigare matlabapplikationen kunde enbart hantera data från S6 och hade därför designats enbart med S6 i åtanke. Driftdatan från S6 innehåller en matris och ett antal vektorer. Alla vektorerna har samma storlek. användargränssnittet hade därför anpassats för att hantera data med fix storlek.

Matlabapplikationen hade utvecklats utifrån krav ställda från motorutvecklingsenheten NMEB och de hade även valt ut de sökkriterier som de ville ha tillgängliga för att förfina sökningar.

Båda de tidigare versionerna var kopplade till en Accessdatabas. Databasen för Matlabapplikationens databas var uppbyggd som så att fordonsinformationen matades in i databasen med hjälp av ett script som kördes en gång i veckan. Fordonsinformationen kommer från en databas vid namn CHIN (*Chassi Information*) som innehåller produktionsinformation för alla Scania-producerade fordon. Driftdatadelarna av databasen går att uppdatera manuellt med hjälp av ett interface i Access.

4.2 Behovsinventering

Intervjuer/möten hölls med potentiella användare. Generellt sett skedde dessa genom möten med ett antal personer som arbetade inom samma område. Ofta var inte dessa grupper bekanta med de äldre applikationerna och hade i vissa fall inte heller reflekterat över möjligheterna med medelvärden av driftdata.

4.2.1 NMEB

Initialt intervjuades personer på NMEB då de hade använt de äldre versionerna och hade önskemål om förändringar på dessa. Deras största önskemål var att ha mer data att arbeta med då de tidigare versionerna hade alltför lite lagrat. Andra önskemål var att programmet skulle ha bättre svarstider.

Rent funktionsmässigt var det viktigt att erhållen data skulle gå att spara i ett format så att data senare går att bearbeta i Matlab. Att kunna se hur den framsökta populationen förhåller sig mot Scanias produktion i stort var bra då det ger möjligheten att verifiera att en population är representativ för den typen av bilar.

4.2.2 OPC/Retarder-utvecklare

Generellt var inte OPC och retarder-utvecklarna speciellt intresserad av att titta på driftdata för populationer. De behov de hade var att kunna titta på enskilda bilar och även jämföra olika utläsningar för denna bil.

Felkoder och felkodsökning var något de gärna skulle se implementerat i programmet. Exempelvis ville de att möjligheten skulle finnas att söka på en viss felkod och ett visst variabelvärde och få fram de fordon som matchade detta.

4.2.3 Slutsatser

Från flera håll kom önskemål att kunna titta på felkoder och söka bland dessa. Rent praktiskt är felkoderna inte komplicerade men om även freeze frames ska infogas blir det genast svårare. Det finns ingen fast struktur på freezeframes utan de är olika för olika felkoder. För att freezeframes ska vara intressant att titta på behövs dess data vara tolkad och arbetet för att genomföra en sådan tolkning ligger inte inom examensarbetets tidsram.

Att kunna bearbeta de värden som räknas fram ansågs viktigt av flera parter. Dock varierade behoven, en del användare såg det som viktigt att värden kunde sparas i Matlab-format medan andra ville använda sig av Microsoft Excel.

4.3 Prototyp av användargränssnitt

Som basis för användargränssnittets utformning användes den äldre matlabapplikationen. Den har utformats och förfinats för motorutvecklarnas behov och innehåller de flesta av de funktioner som användarna vill ha. Att det användargränssnittet var utvecklat i Matlab var dock en begränsande faktor. Om ett annat verktyg används kommer fler möjligheter finnas för presentationen.

Initialt gjordes en skiss på papper över applikationens olika fönster där relaterade funktioner/information grupperades tillsammans.

4.4 Val av verktyg

Valet av utvecklingsverktyg är viktigt. Den slutgiltiga produkten måste kunna uppfylla vissa användarkrav samtidigt som det måste vara nog kraftfullt för att klara av att implementera kraven.

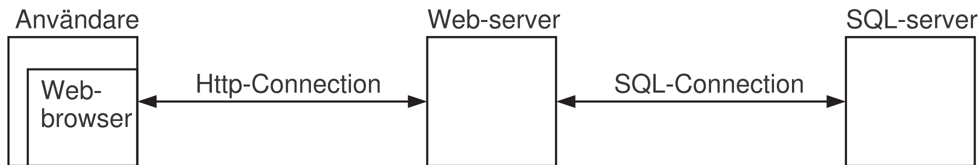
De grundläggande krav som ställts var att applikationen inte skulle kräva licensnycklar för att användas, dvs inte som matlab, samt vara användarvänligt, då speciellt svarstiderna ska vara acceptabla. På Scania används Microsofts produkter för utveckling och det är också en faktor att ta med i beräkningen då det begränsar urvalet av verktyg.

De två alternativ som ansågs möjliga var att göra en webapplikation med Asp.net eller en windowsapplikation. Båda dessa kunde utvecklas med hjälp av verktyget Visual Studio.net vilket är en IDE (*Integrated Development Enviroment*) som finns tillgänglig på Scania.

4.4.1 Webapplikation i ASP.net

ASP.net är en vidareutveckling av Microsofts äldre teknologi ASP (Active Server Pages). ASP.net används för att bygga webapplikationer och klarar av att exekvera C#/Visual Basic-kod. Alla beräkningar sker hos webservern och det enda som användarens maskin gör är att presentera html.

Den stora fördelen med en webapplikation är att vem som helst som har tillgång till en webläsare kan nyttja programmet. Då det enbart är html som skickas till användaren skyddas koden.



Figur 4.2: Översikt över komponenterna i en Webapplikation

I figur 4.2 visas strukturen på en Web-lösning. All kommunikation med SQL-servern sker från Web-servern och användaren kopplar aldrig själv upp sig mot denna.

4.4.2 Windowsapplikation

Genom .net finns verktygen för att enkelt bygga upp användargränssnitt och koppla dessa till den bakomliggande koden, affärslogiken. Dock kräver en .net-applikation att .net framework finns installerat på användarens dator och även att eventuella refererade biblioteksfiler finns tillgängliga.

Fördelen med denna lösning gentemot en weblösning är att alla beräkningar sker på användarens maskin och man slipper kommunicera mot en server så fort användaren interagerar med programmet. En annan fördel är att antalet komponenter som måste kommunicera med varandra blir färre, se figur 4.3.

Figur 4.3: Komponenterna i en Windowsapplikation

4.4.3 Grafritare

Ingen av de två alternativen ovan innehåller funktionalitet för att rita och presentera grafer. För att lösa detta beslutades det att en tredjeparts-komponent skulle användas. Marknaden för sådana bibliotek och plug-ins är ganska stor och många alternativ finns tillgängliga. De alternativ som var möjliga var att nyttja OWC (*Office Web Components*) eller Infragistics NetAdvantage. OWC har nackdelen att det kräver att office-paketet finns installerat på den dator där programmet körs och dessutom har Microsoft, tillverkaren av OWC, väldigt dåligt med manualer och användarhandledning för OWC.

NetAdvantage tillverkat av Infragistics är en typisk tredjepartskomponent som klarar av att visa grafer både i 2D och 3D och har ett enkelt väldokumenterat API. Nackdelen med NetAdvantage är att det kostar pengar för en utvecklingslicens. Dock har Scania ett antal utvecklingslicenser tillgängliga.

4.4.4 Slutgiltigt val

Två testversioner av applikationen gjordes och utvärderades. Då båda varianterna har .net som utvecklingsmiljö kunde mycket av koden användas av båda versionerna. Testimplementationen innefattade att göra funktionalitet för fordonsökning samt att demonstrera dessa resultat och att göra fördelningsfönstret.

Båda alternativen hade de verktyg som krävdes för att visualisera användargränssnittet. Dock var det märkbart krångligare att implementera de mer avancerade finesserna i ASP.net. Exempel på avancerade finesser är sk treeviews (trädmenyer liknande utforskaren i Windows) och tabviews (tabindelning av sidor).

Vad gäller svarstider var skillnaden väsentlig. Webapplikationen kräver ständig kommunikation med Webservern eftersom alla beräkningar sköts på denna. I praktiken innebär det att när användaren trycker på en knapp, ändrar en radioknapp eller skriver in ett värde skickas denna data till Webbservern som hanterar denna. Webservern returnerar html till webläsaren som presenterar den för användaren. Om nätverket är långsamt av någon anledning kan minsta lilla knapptryckning innebära flera sekunders svarstid.

Sammanfattningen av jämförelsen mellan de två alternativen kan ses i tabell 4.2.

Trots att en Windowsapplikation kräver en del förutsättningar för att kunna användas gjordes bedömningen att detta alternativ ska användas. Implementationsarbetet underlättas vilket ger större möjlighet att skapa en stabil produkt inom tidsramen.

Vad gäller kravet på .net framework 1.1 är det troligt att kommande windowsversioner kommer att inkludera denna automatiskt. Det är också enkelt att skapa en installationswizard för analysprogrammet som installerar .net framework vid behov på användarens arbetstation.

4.5 Implementation

Implementationen genomfördes enligt de riktlinjer och planer som framkommit genom de tidigare stegen i utvecklingen.

Tabell 4.2: Jämförelse mellan en Webapplikation och en windowsapplikation

	Webapplikation	**Windowsapplikation**
Databaskoppling	Använder sig av ADO.nets[a] funktioner	Använder sig av ADO.net
Krav på användarens arbetsstation	Inga förutom att en webläsare måste finnas tillgänglig	Kräver att .net framework 1.1 finns installerat samt eventuella dll-filer för externa bibliotek.
Verktyg och funktionalitet	ASP.net klarar av att visualisera det specificerade användargränssnittet. Den bakomliggande koden är skriven i C#	Har funktioner för att bygga och visualisera användargränssnittet genom C# funktionsbibliotek.
Användarvänlighet och svarstider	Kräver ständig kommunikation med webservern samt har problem att generera dynamisk kod. Beräkningar görs hos Webservern.	Beroende på användarens maskin. Beräkningar sker på den maskin som programmet exekveras på.

[a] ADO.net är en del av .net framework som innehåller objekt för att enkelt kommat åt data. Namnet kommer från den äldre teknologin ADO (*ActiveX Data Objects*).

4.6 Utvärdering/Test

Applikationen testades och utvärderades på flera olika sätt. Dels demonstrerades programmet för utvalda grupper av användare som fick komma med synpunkter och önskemål och dels tilläts användare själva testa programmet och utan vägledning pröva dess funktioner. Båda metoderna genererade en hel del förslag på ändringar. Dock visade det sig att användargrupper kunde ha skilda önskemål som inte båda var möjliga att genomföra.

Programmet gjordes tillgängligt för nedladdning för alla inom utvecklingsenheten på Scania. Syftet med detta är att låta utvecklare använda programmet skarpt och på så sätt upptäcka eventuella buggar och felaktigheter.

Kapitel 5

Resultat

5.1 Databas

För att ha möjlighet att göra SQL-frågor med data från flera olika databaser gjordes valet att bygga upp en egen SQL-databas i Microsoft SQL Server för applikationen. Denna fick namnet DCDA (*Duty Cycle Data Analysis*) och är en Microsoft SQL-databas.

Då många av sökningarna kräver matchning mellan en sökt population av fordon och driftdata behöver både fordonsdata (information om de olika fordonen) och driftdata finnas i DCDA. Fordonsdatan kopierades från CHIN med hjälp av ett script och på liknande sätt kopierades data från VERA. En överblick över hur databaserna är relaterade kan ses i figur 5.1.

Då DCDA i stort sett enbart innehåller data som redan finns i andra databaser kunde det möjligen ha varit lämpligt att DCDA enbart innehöll referenser till de två andra databaserna. Detta ansågs dock inte vara en bra lösning då för mycket beroenden mellan databaserna skulle skapas.

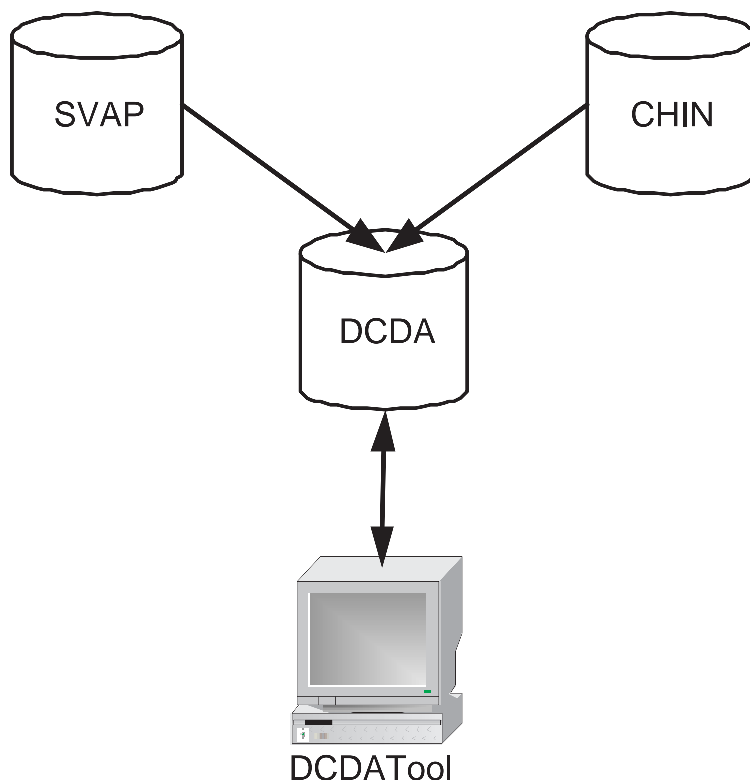
Tabellerna i DCDA kan sorteras in i två grupper huruvida de har koppling till CHIN och fordonsdata eller om de har koppling till driftdata från VERA. Den gemensamma länken mellan fordonsdatan och driftdatan är chassinummret på fordonen. Detta är ett nummer som unikt identifierar alla fordon.

5.1.1 Fordonsdata

CHIN lagrar information om alla fordon som producerats av Scania. Informationen som lagras är bilens konfiguration vid den tidpunkt då den lämnar Scania. Om förändringar görs på bilen vid ett senare tillfälle (reservdelar etc.) uppdateras CHIN. Konfigurationsvariablerna är ett par tusen stycken och av dessa har 22 valts ut för att användas för populationsökning. Urvalet av dessa konfigurationsvariabler baserade sig på de konfigurationsvariabler som användes av matlabapplikationen.

Scriptet som kopierar data från CHIN till DCDA är schedulerat till att köras en gång i veckan och körs på söndag natt. Anledningen till att köra scriptet denna tid var att begränsa kopieringens inverkan på övrig verksamhet.

I figur 5.2 syns en vy över den delen av DCDA som rör fordonsdata. Den viktigaste tabellen av dessa är Chassi_Variant. Den innehåller chassinummer för alla fordon och för varje chas-



Figur 5.1: Översikt över databaserna

sinummer finns kolumner för dess konfiguration. Konfigurationsinformationens kolumner är namngivna efter den variantkod som konfigurationsvariabeln har.

Variantkoderna finns beskrivna i de tre tabellerna VAR_CODES, VAR_CDE och VAR_FAM. De fyra tabellerna som alla har prefixet FAM (FAM_Descr_Truck, FAM_Descr_All, FAM_Descr_Bus och FAM_Descr_Common) används för att söka fram valbara alternativ när en fordonsökning ska utföras.

De tre kvarvarande tabellerna (Continents, Results och Mount_Location) används för specialsökningar.

5.1.2 Driftdata

Tabellstrukturen på den del av DCDA som innehåller driftdata är lite mer invecklad, se figur 5.3. Tabellerna har prefixet sv för att visa att deras data kommer från SVAP.

Tabellen svVector innehåller den data som avgör ifall en variabel är publik eller inte. svVector är inte synlig för applikationen utan används för säkerhetsfunktionerna i databasen. Tabellen svTruck är en tabell med chassinummer för de fordon som har driftdata. Denna tabell tillsammans med tabellen Chassi_variant används för att matcha en populationsökning mot driftdata.

Figur 5.2: Vy över databastabellerna med fordonsdata

Tabellen svSnapshot innehåller en rad för varje snapshotfil som lästs in. Dess relation med svTruck är många till en. Varje snapshotfil har en rad i svEcu för varje styrsystem. Varje styrsystem har i sin tur en rad i svEcuChart för varje driftdatavariabel. Tabellen SvEcuChart är en central tabell som har relationer med ett antal andra tabeller. Varje driftdatavariabel har koppling till tabellerna svBreakPointX och svBreakPointY vilka representerar de olika brytpunkterna i X-led (för matriser och vektorer) och Y-led (enbart för matriser). Brytpunkternas värden finns i tabellerna svBreakPointValueX respektive svBreakPointValueY.

Tabellen svSample innehåller värdena för alla typer av driftdata. Beroende på driftdatatyp relaterar olika många rader i svSample till varje rad i svEcuChart. Ett räknarvärde har bara en rad i svSample medan en vektor och en matris har flera.

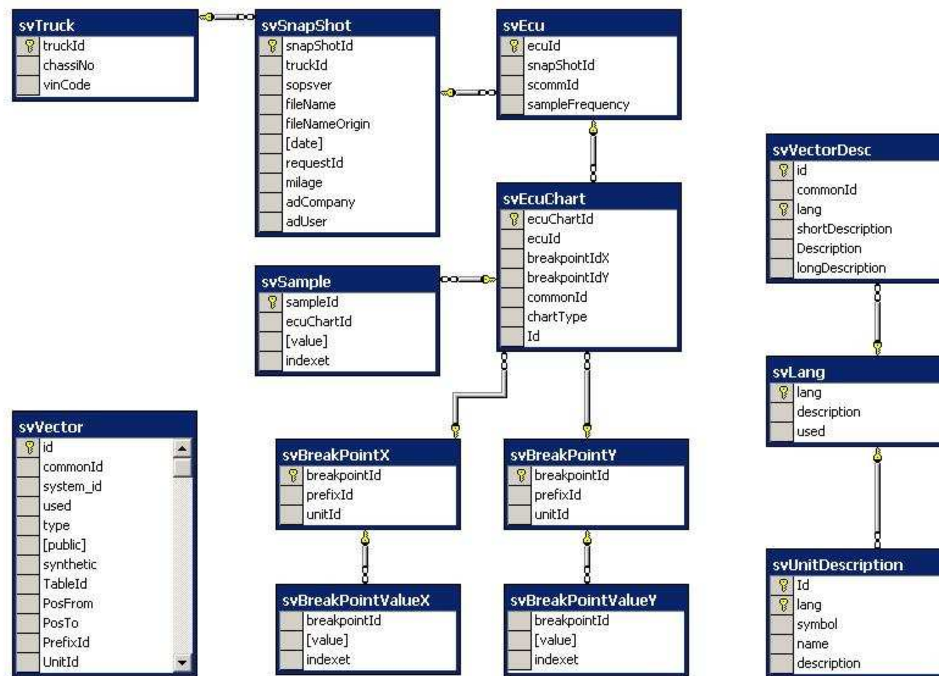
Tabellen svVectorDesc innehåller informationstexter för varje driftdatavariabel, både det interna variabelnamnet och ett mer beskrivande namn. Tabellen svVectorDesc är relaterad till svLang som listar de olika språktyperna som beskrivningarna kan ha.

Tabellen svUnitDescription innehåller enheter för axlarna hos matriser och vektorer. Tabellen länkar också till svLang.

5.1.3 Säkerhet och accesslösning

DCDA innehåller en hel del data som är hemlig och enbart ska finnas tillgänglig för vissa grupper på Scania. Rent allmänt ska programmet inte kunna användas av personer utanför utvecklingsenheten. Applikationen har därför kravet att den ska kunna garantera detta.

Alla driftdatavariabler har en flagga som visar ifall det är en publik variabel eller inte. Flaggan som heter public finns i tabellen svVector.



Figur 5.3: Vy över databastabellerna med driftdata

För att hantera accessäkerheten utnyttjas användarkontona i Windows NT som användare i DCDA. Detta sköts genom att NT-användarna kan tillhöra två roles, DCDAReader och DCDALimitedReader. DCDALimitedReader har tillgång till alla publika variabler och DCDAReader har tillgång till alla, både de publika och de hemliga variablerna. De berörda tabellerna i databasen är inte alls tillgängliga för någon användargrupp utan istället får de tillgå olika vyer som använder sig av gruppmedlemskapen för att välja ut rader ur tabellerna. I T-SQL[1] används kommandot `IS_MEMBER(Gruppnamn")` som returnerar 0 för falskt och 1 för sant.

Genom att låta användarna använda sina egna NT-konton för att logga in innebär det att inga användarnamn eller lösenord behöver skickas över nätverket och därmed finnas lagrade i programmet.

5.2 Applikation

Den färdiga applikationen är en Windowsapplikation vid namn DCDATool. Programmet hämtar data från DCDA genom SQLfrågor.

När programmet startar har användaren möjlighet att välja vilken typ av sökning som önskas. De två alternativen som finns är:

[1] T-SQL (*Transact-SQL*), är Microsofts vidarutveckling på SQL som används av bland annat Microsoft SQL Server

Sökning av fordon – Söker fram en population av fordon

Sökning på chassinummer – Söker fram ett fordon med hjälp av chassinummret.

5.2.1 Sökning av fordon

Vid valet Sökning av fordonpresenteras följande sökfönster (se figur 5.4):

Figur 5.4: Bild över sökfönstret

I rullgardinsmenyerna kan olika val göras för att förfina sökningen. Fälten är också editerbara och sökfunktionen klarar av att hantera enkla boolska uttryck. AND och OR går att använda var för sig och i kombination. Enkla reguljära uttryck med % som ”wildcard” fungerar också.

Om lastbil eller buss väljs som fordonstyp ges det möjlighet att använda två andra variabler. De variabler som finns att välja mellan är:

- Bakaxelutväxling
- Bakaxelväxeltyp
- Bussuppbyggnad (endast för Bussar)
- Chassityp
- Chassipåbyggnad (endast för lastbilar)
- Däckdimension
- Däcksfabrikat
- Däcksmönster
- Däcksomkrets
- Fordonstyp
- Hastighetsbegränsning
- Hjulkonfiguration

- Hyttyp (endast för lastbilar)
- Kraftutag
- Marknad
- Monteringsperiod
- Motorplacering (endast för bussar)
- Motortyp
- Produktionsfabrik
- Retarder
- Opticruise
- Växellådstyp

5.2.2 Sökresultat

När knappen ”Gör sökning” trycks utför programmet en sökning med de valda alternativen. Resultatet presenteras i en dialog innehållande de funna chassinummren, ett textfönster med de valda urvalsalternativen, antalet funna chassin och en trädmeny med driftdatavariabler (se figur 5.5 nedan).

I trädmenyn kan användaren välja ut en variabel att titta närmare på. Variablerna finns sorterade i olika flikar beroende på vilket styrsystem de tillhör. Användaren kan genom radioknappar välja om variablerna ska vara representerade med sina riktiga variabelnamn eller med en beskrivning på engelska. Framför varje variabel står ett värde vilket visar hur många fordon i populationen som har värden för denna variabel. Olika versioner av samma styrsystem kan lagra olika variabler och på olika sätt (mer om det senare).

För att visa vilken typ av driftdata variabeln representerar används olika ikoner.

För att välja en variabel kan dess namn dubbelklickas eller alternativt markeras och sedan aktiveras med ett klick på knappen Visa. Om det är en vektor eller en matris det handlar om och dessa råkar ha flera olika uppsättningar av brytpunkter för sina värden väljs en av dessa från en meny. Möjligheten finns också att ange ett eget intervall. Om det bara finns en uppsättning brytpunkter visas vektorn/matrisen direkt. En vektor och en matris representeras på följande sätt (se figur 5.6 och 5.7 nedan).

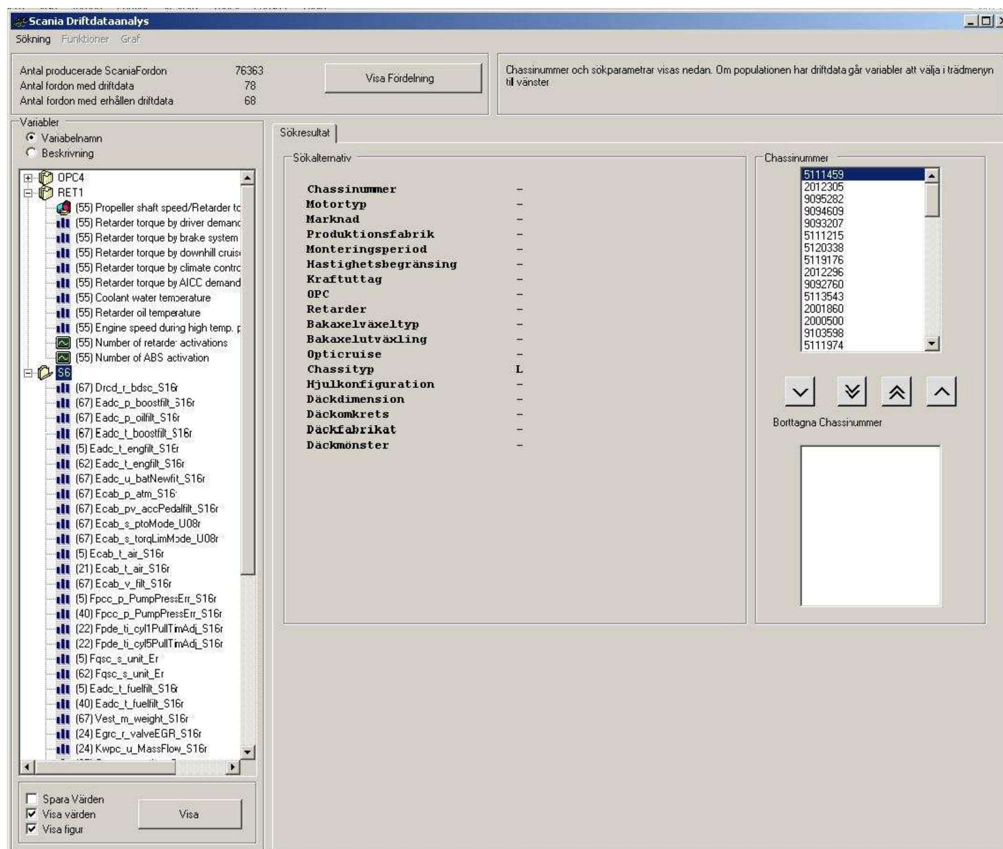
Vid flera brytpunkter kommer värdena approximeras till att alla bilars värden befinner sig inom samma intervaller, dvs. har samma brytpunkter. För att approximera detta antas det att värdena fördelar sig jämt över hela sina intervall och att de sedan delas upp och adderas till det valda intervallet.

När en variabel väljs för visning öppnas inte en ny dialog utan istället finns en tab-meny som låter användaren ha sökresultatstabben tillgänglig i samma fönster.

Staplarna representerar medelvärden uträknade från den sökta populationens värden. Under själva grafen skrivs matrisen/vektorns värden ut i textform. Dessa värden kan sparas undan för vidare bearbetning i ett annat program. Programmet klarar av att spara värdena i ett enkelt kommaseparerat format läsligt för matlab och ett format som är kompatibelt med Microsoft Excel.

5.2.3 Fördelning

Genom att trycka på knappen ”Visa Fördelning” öppnar sig en ny tab där möjligheten finns att se hur väl den framsökta populationen förhåller sig till Scanias produktion. Fördelningen kan visas i upp till tre led



Figur 5.5: Screenshot på sökresultatsfönstret

I figur 5.8 visas fördelningen över ChassiTyp och hyttyp (2:a ledet). Både hur alla fordon i CHIN fördelar sig över dessa variabler och hur populationen fördelar sig visas i procent.

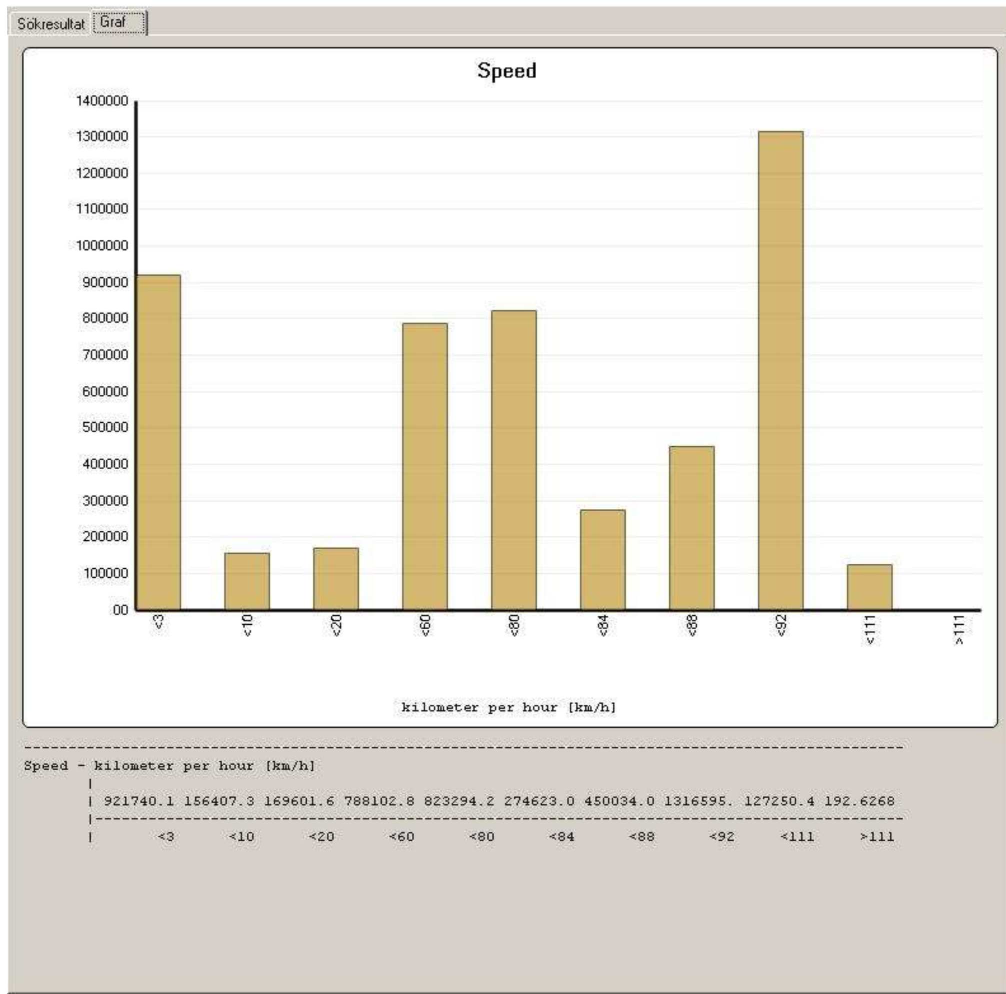
5.2.4 Enskilt fordon

När alternativet chassisökning väljs från menyn ”sökning” presenteras en dialogruta med möjligheten att skriva in ett chassinummer. Det går även att välja chassinumret från en rullgardinsmeny innehållande alla chassinummer från CHIN.

Efter att användaren har valt ett chassinummer presenteras ett sökresultatsfönster liknande det för populationsökning, se ovan. Det som skiljer sig från det tidigare nämnda sökresultatfönstret är att här presenteras det sökta fordonets konfiguration tillsammans med en lista på de olika utläsningarna som gjorts för fordonet. Populationsökningen arbetar bara med den senaste utläsningen av varje fordon men här ges möjligheten att titta på tidigare utläsningar.

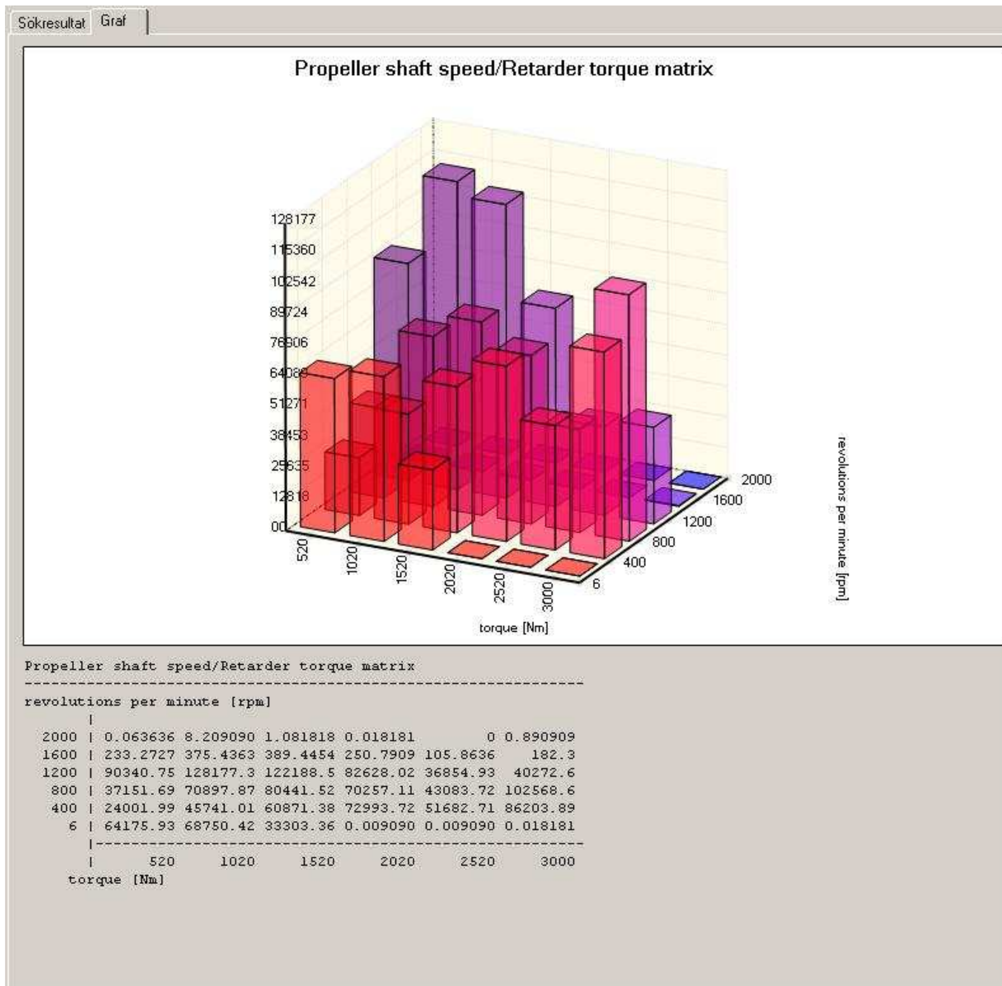
Utläsningarna är egentligen olika snapshotfiler och det datum som de har läst ut presenteras tillsammans med antalet mil fordonet totalt hade gått vid utläsningstillfället.

Till vänster presenteras driftdatavariablerna för de olika styrsystemen och dessa kan visas på samma sätt som för populationer, men värdena som visas är då naturligtvis inga medelvärden.



Figur 5.6: Bild på en driftdatavektor representerat som ett histogram

En screenshot över sökresultatfönstret för en chassisökning visas i figur 5.9.



Propeller shaft speed/Retarder torque matrix

revolutions per minute [rpm]

	520	1020	1520	2020	2520	3000
2000	0.063636	8.209090	1.081818	0.018181	0	0.890909
1600	233.2727	375.4363	389.4454	250.7909	105.8636	182.3
1200	90340.75	128177.3	122188.5	82628.02	36854.93	40272.6
800	37151.69	70897.87	80441.52	70257.11	43083.72	102568.6
400	24001.99	45741.01	60871.38	72993.72	51682.71	86203.89
6	64175.93	68750.42	33303.36	0.009090	0.009090	0.018181

torque [Nm]

Figur 5.7: Bild på matris-presentation

Sökresultat Fördelning

Fördelningsvariabler

Fördelning 1 led Chassityp Aktivera

Fördelning 2 led Hyttyp ☑

Fördelning 3 led Bakaxelutväxling ☐

Fördelning kan väljas i upp till tre led

Chassityp	Hyttyp	n dd	n prod	procent av Tot	procent av Pop
C	CP	4	3862	5,06 %	0,01 %
C	CR	2	3744	4,90 %	0,00 %
C	CT	0	101	0,13 %	0,00 %
D	CP	3	4001	5,24 %	0,00 %
D	CR	0	214	0,28 %	0,00 %
G	CP	0	932	1,22 %	0,00 %
G	CR	0	3295	4,31 %	0,00 %
G	CT	0	142	0,19 %	0,00 %
L	CP	4	4452	5,83 %	0,01 %
L	CR	64	54904	71,90 %	0,08 %
L	CT	0	637	0,83 %	0,00 %

Figur 5.8: Vy över fördelningstaben med två leds fördelning

Figur 5.9: Bild över sökresultatfönstret vid chassisökning

Kapitel 6

Sammanfattning

Den ursprungliga applikationen i matlab visade sig inte vara kraftfull nog för att klara att hantera både ett användargränssnitt och tunga databassökningar. Vidare krävde den att användaren hade tillgång till matlab med tillhörande licens.

Utifrån den specifikation och de önskemål som framkom under analysen och behovsinsamlandet implementerades och produktionsattes ett analysverktyg. Så långt har detta projekt uppfyllt de uppsatta målen.

Övriga typer av data (Felkoder, ecuID, freezeframes, etc.) hanteras inte av programmet. Orsaken till detta är att de inte finns i SVAP och att det inte heller finns någon färdig specifikation eller dokumentering för hur dessa kommer att lagras (i form av tabeller i databasen).

6.1 Möjliga förbättringar och optimeringar

6.1.1 DCDA

Databasen DCDA har kopierat alla relevanta tabeller från SVAP och uppdateringsscriptet för över all data från alla kolumner i dessa tabeller. DCDATool använder sig dock inte av alla kolumner och vissa tabeller skulle mycket troligt kunna slås ihop. Ifall tabellerna i DCDA skulle anpassas mer efter applikationens behov blir databasen mer översiktlig. En positiv bieffekt till en sådan anpassning är att datat inte upptar lika mycket diskutrymme vilket även innebär en kostnadsminskning.

För att genomföra en sådan anpassning krävs att tabellerna och programmet analyseras och att uppdateringsscriptet görs om så att det enbart kopierar den data som används.

6.1.2 Beräkningar

DCDATool hämtar data från databasen DCDA och utför beräkningar på denna data. Hur snabbt dessa beräkningar går beror på vilken dator som programmet exekveras på. Beräkningarna kan bli ganska tunga om det är driftdatamedelvärden för ett stort antal fordon och speciellt om dessa har olika brytpunkter på vektor eller matris.

En förbättring och optimering vore att utföra mer beräkningar i själva databasen så att datan som hämtas för de olika driftdatavariablerna är beräknade medelvärden.

6.2 Önskemål för framtida versioner

Genom användartester och presentationer av programmet har ett antal önskemål på kommande funktionalitet lagts fram.

6.2.1 Rapportfunktion

En funktion som gör det möjligt för en användare att skapa en rapport med bilder (grafer) och värden. Användaren ska då få välja ut vilka variabler som ska vara med i rapporten och hur de ska presenteras. En sådan rapport skulle då gärna gå att få ut i pdf eller doc-format.

6.2.2 Andra sökmöjligheter

Vissa efterfrågade möjligheten att kunna söka på vissa drifttider eller total körsträcka. Drifttiden går att räkna ut genom att addera alla kolumner i en matris (värdena uppdateras med 1Hz) för att få ut totala antalet sekunder som fordonet har varit i drift. Körsträckan finns sparad i databasen i tabellen svSnapshot.

Andra önskemål som framkom under framförallt behovsinsamlingen var möjligheten att söka en population på sedvanligt sätt samtidigt som extra sökkriterier rörande driftdatavärden lades in. Ex. Sök alla fordon som sålts i Spanien som har haft en växellådstemperatur över 100°C.

6.2.3 EcuID

Det ansågs viktigt att kunna titta på ecuID från enskilda fordon då den visar sådant som styrsystemsversioner och annat av intresse. Vid presentation av konfigurationen för enskilda fordon vid chassisökning finns utrymme avsatt för att presentera ecuId. Det som krävs för att denna information ska kunna presenteras är att datan tolkas och läggs in i SVAP och sedan att script och applikation ändras.

6.2.4 Grafer med populationer och enskilda chassin

För användare som vill använda DCDATool för att undersöka enskilda fordon skulle möjligheten att presentera grafer över populationers värden tillsammans med värdena från ett enskilt fordon vara användbart. På så sätt kan en utvecklare lätt se ifall det aktuella chassit avviker på något sätt från andra fordon med samma konfiguration.

6.2.5 Felkoder (DTC)

Redan under behovsinsamlingen framkom det att dessa data var av stort intresse för många av användarna. Ett antal olika förslag på funktioner rörande DTCer har lagts fram.

- Felkodsökning, möjligheten att söka på en viss felkod och titta på fordonsfördelningar.
- Presentera felkoder på liknande sätt som driftdata. Med det menas att felkoderna kommer att få en egen flik i trädmenyn i sökfönstret och för varje felkod kan sedan statistik (medelvärden) visas.
- Att för enskilda fordon kunna undersöka dess felkoder och titta på freezeframes, då gärna tolkad eller eventuellt ges möjligheten att spara undan freezeframen för analys i med ett annat verktyg.

6.2.6 Fältprovbilar

I SVAP lagras enbart data från kundbilar, dvs sålda fordon, men det finns också ett antal fordon som kallas för fältprovbilar. Dessa används för fälttester och loggar naturligtvis också driftdata. Deras driftdata skulle också kunna sparas i SVAP för att kunna analyseras med hjälp av DCDA-Tool. Dock skulle det vara nödvändigt att kunna skilja på kundbilar och fältprovbilar i databasen då analys ska göras. Fältprovbilarna loggar inte driftdata med samma frekvens som kundbilarna och får då helt andra värden på sina variabler. Om dessa skulle räknas in tillsammans med kundbilar skulle medelvärdena bli högre än de egentligen är. Möjligen skulle då loggfrekvensen för dessa fältprovbilar lagras och då kunna användas för att vikta driftdatan. Som exempel skulle ett fordon som loggar driftdata med frekvensen 10Hz ha 10 gånger så höga värden som en kundbil som loggar med 1Hz på samma drifttid.

6.3 Lösningens Begränsningar

En begränsning som gjorts jämfört med andra analysverktyg och också den tidigare versionen är att ingen ytterligare tolkning av datan har gjorts. All tolkning är upp till användaren. Som exempel på tolkning kan nämnas att i analysverktyget SD2 (*Scania Diagnos 2*), en föregångare till SDP3, visades driftdata för S6 där intervallen var färgade beroende på hur gynnsamma eller lämpliga de var för fordonet att befinna sig i.

Anledningen till att DCDATool inte hjälper till med tolkningen av data är för att DCDATool ska vara ett generellt verktyg som anpassar sig efter vilken driftdata som finns i SVAP. Om denna driftdata ändras eller nya styrsystem tillkommer ska DCDATool klara av att läsa in och presentera även denna data.

6.3.1 Angående omräkning av intervall för matriser/vektorer

Metoden att räkna om värdena för matriser/vektorer med andra brytpunkter kan ge missvisande resultat. Att antaga att värdena fördelar sig jämt över ett intervall kan vara helt fel. Exempelvis kan ett fordon som har stått still mycket ha höga värden i intervallet 0-20km/h i en hastighetsmatris. Om detta intervall sedan delas upp i 0-10 och 10-20 kommer omräkningen visa att bilen har körts mycket i låga hastigheter, krypkörning. Om slutsatser rörande bilens beteende dras utifrån detta kan det bli fel.

Kapitel 7

Tack

Jag skulle vilja börja med att tacka mina två handledare. Ola Ågren, min interna handledare, för tips, kommentarer och ovärdelig LaTeX-kunskap. Johan Gerhardson, min externa handledare, för feedback och hjälp att hantera Windows. I övrigt vill jag tacka Malin Pihl, Tomas Pihl och Björn-Egil Dahlberg för korrekturläsning av rapporten och support.

References

[1] Alex Buckley och Susan Eisenbach Anders Aaltonen. Flexible dynamic linking for .net. http://slurp.doc.ic.ac.uk/pubs/flexibledynamiclinkingfordotnet.pdf, (visited 2006-01-17).

[2] Alex Buckley. A model of dynamic binding in .net. http://www.cs.ru.nl/ftfjp/2005/Buckley.pdf, (visited 2006-02-09).

[3] Andrew Kennedy och Don Syme Dachuan Yu. Formalization of generics for the .net common language runtime. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 39–51, New York, NY, USA, 2004. ACM Press.

[4] DotGNU Project. Dotgnu project - gnu freedom for the net. http://www.dotgnu.org/, (visited 2006-01-22).

[5] ECMA. Standard ecma-335 common language infrastructure (CLI). Technical report, ECMA, 2001.

[6] Howard Gilbert. .net framework. http://pclt.cis.yale.edu/tp/framework.htm, (visited 2006-01-17).

[7] K. Gough. Stacking them up: A comparison of virtual machines. In *Australasian Computer Systems Architecture Conference*, pages 52–59. IEEE Computer Society Press, 2000.

[8] Jennifer Hamilton. Language integration in the common language runtime. *SIGPLAN Not.*, 38(2):19–28, 2003.

[9] Richard A. Kilgore. Open source initiatives for simulation software: multi-language, open-source modeling using the microsoft .net architecture. In *WSC '02: Proceedings of the 34th conference on Winter simulation*, pages 629–633. Winter Simulation Conference, 2002.

[10] Pattie Maes. Concepts and experiments in computational reflection. In *OOPSLA '87: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 147–155, New York, NY, USA, 1987. ACM Press.

[11] Microsoft Corporation. Ecma c# and common language infrastructure standards. http://msdn.microsoft.com/netframework/ecma/, (visited 2006-01-17).

[12] Mono Project. Mono project. http://www.mono-project.com/Main_Page, (visited 2006-01-22).

[13] Meg Murray. Move to component based architectures: introducing microsoft's .net platform into the college classroom. *J. Comput. Small Coll.*, 19(3):301–310, 2004.

[14] Kelly Heffner och Christian Collberg. The obfuscation executive. Technical Report TR04-03, Department of Computer Science, University of Arizona, 2004.

[15] Erik Meijer och Clemens Szyperski. What's in a name? .net as a component framework. In *Proceedings of the First OOPSLA Workshop on Language Mechanisms for Programming Software Components*, pages 22–28. Technical Report NU-CCS-01-06, College of Computer Science, Northeastern University, Boston, MA 02115, 2001.

[16] Nathanael Paul och David Evans. .net security: Lessons learned and missed from java. In *ACSAC '04: Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC'04)*, pages 272–281, Washington, DC, USA, 2004. IEEE Computer Society.

[17] Màrcio E. Delamaro och Gian Pietro Picco. Mobile code in .net: A porting experience. In *MA '02: Proceedings of the 6th International Conference on Mobile Agents*, pages 16–31, London, UK, 2002. Springer-Verlag.

[18] E. Meijer och J. Gough. Technical overview of the common language runtime. Technical report, Microsoft, 2000.

[19] Gregory Neverov och Paul Roe. Towards a fully-reflective meta-programming language. In *CRPIT '38: Proceedings of the Twenty-eighth Australasian conference on Computer Science*, pages 151–158, Darlinghurst, Australia, Australia, 2005. Australian Computer Society, Inc.

[20] Ramez Elmarsi och Shamkant B. Navathe. *Fundamentals of Database Systems*. Addison-Wesley, 2004.

[21] Walid Taha och Tim Sheard. Multi-stage programming with explicit annotations. In *PEPM '97: Proceedings of the 1997 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 203–217, New York, NY, USA, 1997. ACM Press.

[22] Jeffrey Richter. *Applied Microsoft .net framework programming*. Microsoft Press, Redmond, Washington, 2003.

[23] V. Jurisic och C. Sadler S. Eisenbach. Feeling the way through DLL Hell. `http://www.cs.uni-bonn.de/ gk/use/2002/sub/eisenbach.pdf`, (visited 2006-02-09).

[24] Jeremy Singer. Gcc .net—a feasibility study. In *Proceedings of the First International Workshop on C# and .NET Technologies*, pages 55–62. University of West Bohemia, 2003.

[25] Jeremy Singer. JVM versus CLR: a comparative study. In *PPPJ '03: Proceedings of the 2nd international conference on Principles and practice of programming in Java*, pages 167–169, New York, NY, USA, 2003. Computer Science Press, Inc.

[26] Don Syme. ILX: Extending the .NET common IL for functional language interoperability. *Electronic Notes in Theoretical Computer Science*, 59(1), 2001.

[27] Walid Mohamed Taha. *Multistage programming: its theory and applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, 1999. Supervisor-Tim Sheard.

[28] Stephen Walther. *ASP.NET Unleashed, 2nd edition*. SAMS Publishing, Indiana, USA, 2004.

[29] Karli Watson. *Börja med C#*. Pagina Förlags AB, Sundbyberg, Stockholm, 2002.

Bilaga A

Förkortningar

A.1 Scaniaspecifika

SDP3	Scania diagnos och programmering 3
SD2	Scania Diagnos 2
OPC	Opticruise
S6	En motorstyrenhet
RET	Retarder
VERA	Vehicle Electronic Reconfiguration Application
CHIN	Chasssis Information
SVAP	Scania Vehicle Analysis Portal
DCDA	Duty Cycle Data Analysis
DTC	Diagnostic Trouble Code

A.2 .net

CLR	Common language runtime
MSIL	Microsoft Intermidiate language
CTS	Common Type System
CLS	Common language specification
FCL	Framework Class Library
mscorlib	Multilanguage Standard Common Object Runtime Library
C#	C-sharp

A.3 Övriga

ECMA	European Computer Manufacturers Association
IDL	Interface definition library
JVM	Java Virtual Machine

COM	Common Object Model
SOM	System Object Model
ASP	Active Server Pages
T-SQL	Transact-SQL (Microsofts utökade variant av SQL)