Umeå Tekniska Högskola
Institutionen för Datavetenskap
16 mars 2004

Examensarbete 20 poäng
Handledare: Jerry Eriksson
HT 2003

Examensarbete
LrsPc testmiljö

. . . eller hur man med hjälp av statecharts utvecklar en modell av kraftförsörjningen till WCDMA-basstationer.

Författare:
Andreas Friis

Abstract

The work presented in this thesis consists of a description of a software model that simulates the power control in WCDMA-basestation's and an evaluation of different methods to develop statecharts. The software model is able to replace hardware tests of the basestations control software with tests against the model. The greatest achievement of the model is the ability to imitate the power controls function. The model is also built so that the tester can adjust the behavior so that it differs from normal behavior.

The evaluation of methods for developing statecharts includes four methods with different characteristics and design. The methods are: KEK, VT, WS and MAS. The result of the investigation was for example that two of the methods (KEK and MAS) cant be recommended for use because they lack the ability to use hierarchies, one of statecharts most important features. WS is a complete method for developing statecharts but has a big disadvantage in its complexity, it is both hard to use and to understand. The last method, VT, is easy to use but it doesn't give the designer much help, therefore it is best suited for small projects.

The thesis ends with a glance at the future, how statecharts will develop and what needs to be done to get more program designers to use statecharts.

Innehåll

Figurer

Kapitel 1

Inledning

Det här kapitlet beskriver varför en undersökning om metoder för att ta fram statecharts är intressant och varför en sådan undersökning behövs. Kapitlet inleds med en problembakgrund som beskriver de bakomliggande orsakerna till att undersökningen genomförs, därefter presenteras vilket problem som skall undersökas och vilket som är undersökningens syfte och mål. Kapitlet innehåller även en sammanställning av de förkunskaper läsaren förväntas ha för att kunna ta till sig uppsatsen. Avslutningsvis ges en sammanfattning av uppsatsens fortsatta disposition.

1.1 Problembakgrund

TietoEnator i Umeå utvecklar en programvara (*LrsPc*) som har hand om kraftförsörjningen till Ericssons WCDMA-radiobasstationer. Programvaran har bland annat som uppgift att övervaka kraftförsörjningen med avseende på gränsvärden för ström, spänning och temperatur, ladda eventuella backup-batterier samt att hantera strömavbrott. Under utvecklingen av programvaran krävs det regelbundna tester för att verifiera att den fungerar korrekt och att den följer specifikationen. Idag måste många av testerna ske direkt mot hårdvaran som består av batteri och växelströmsenheter. Det är tidsödande och dyrt att testa mot hårdvara, ett snabbare och billigare alternativ är att utveckla en *mjukvarumodell* som efterliknar det fysiska systemets beteende.

Examensarbetets mål är att utveckla en modell av kraftförsörjningsenheten

till en WCDMA-radiobasstation som beter sig på samma sätt som det fysiska systemet. Modellen skall reagera likadant på signaler som det fysiska systemet och den skall skicka mätvärden som är rimliga i förhållande till de reglersignaler som LrsPc skickar.

Applikationen är ett typiskt exempel på ett reaktivt system, alltså system som kontinuerligt måste reagera på extern och intern stimuli [Har87]. Andra exempel på reaktiva system är operativsystem, telefoner och trafikljus. Motsatsen till reaktiva system är *transformella system* som kännetecknas av att de går en förutbestämd väg under exekveringen. Exempel på detta är program för att komprimera musik eller en funktion som beräknar roten ur ett tal. För att utveckla reaktiva system använder TietoEnator utvecklingsmiljön *Rational Rose RealTime* härefter kallad Rose RT.

När ett programvarusystem utvecklas är det vanligt att börja med högnivådesignen, alltså en övergripande beskrivning av systemet som grovt beskriver dess komponenter. Vid modellering av reaktiva system används ofta en UML-notation som kallas för *statecharts*, statecharts introducerades av David Harel 1987 och är tillståndsmaskiner (finite state machines) som utökats med hierarkier och parallellism [VT01]. Mer information om statecharts presenteras i kapitel 3. Att ta fram statecharts är ingen enkel uppgift, det krävs en rutinerad mjukvarudesigner för att utveckla statecharts som täcker ett systems samtliga krav. För att lyckas måste systemets samtliga funktioner vara specificerade och designern måste veta hur de skall implementeras. Om designen inte är korrekt måste systemet byggas om vilket kan bli mycket dyrt om felet finns i systemets grundstruktur. För att förenkla framtagandet av statecharts har ett antal metoder utvecklats, bland annat [WS00], [VT01], [KM93] och [KEK98]. Metoderna ger hjälp till systemdesignern för att utveckla statecharts som fungerar korrekt och tar upp systemets samtliga krav.

Att det är svårt att utveckla statecharts visade sig tidigt när modellen av kraftförsörjningen skulle implementeras. Vid undersökningar om vilken metod för att utveckla statecharts som skulle passa det här projektet visade det sig att det inte finns några oberoende jämförelser mellan olika metoder. De jämförelser som finns tillgängliga är gjorda av forskare som själva har utvecklat sin egen metod och som har egenintresse i att presentera sitt eget arbete i så gott ljus som möjligt. För att råda bot på det här problemet innehåller den här uppsatsen förutom en presentation av implementeringen av mjukvarumodellen även beskrivningar av fyra olika metoder för att utveck-

la statecharts. Metoderna jämförs och deras för- och nackdelar presenteras, jämförelsen kan användas av andra som vill komma igång med statecharts och som vill veta vilken metod som passar deras projekt.

1.2 Problemformulering

Den här rapporten består av två delar; dels en teoretisk undersökning där ett antal metoder för att utveckla statecharts jämförs och dels en implementering av en modell för att kunna mjukvarutesta programvaran som sköter kraftförsörjningen till WCDMA-basstationer. Implementeringen sker utifrån en statechart som togs fram med hjälp av en av metoderna som undersökts.

Det här leder fram till undersökningens problemformulering, alltså den fråga som undersökningen skall besvara:

> *Vilka olika metoder finns det för att ta fram statecharts och vilka för- och nackdelar har respektive metod?*

1.3 Syften

Målet för det här examensarbetet är att:

> *- utveckla en modell av det fysiska systemet för kraftförsörjningen till WCDMA-basstationer som möjliggör mjukvarutest av regler-programvaran.*

1.4 Några ord om rapportens språk

Engelska facktermer inom datavetenskap saknar ofta översättning till svenska vilket gör att språket i den här rapporten ibland blir en blandning mellan engelska och svenska. Vissa uttryck får en underlig grammatik när de införs i svenska meningar, jag har dock valt att inte göra några egna översättningar av termer eftersom jag anser att det är bättre att använda vedertagna uttryck än att alltid ha korrekt grammatik. Därför ber jag läsaren ha överseende med felaktiga böjningsformer som finns hos vissa engelska

ord.

1.5 Läsarens förkunskaper

Läsaren förutsätts ha grundläggande kunskaper inom Unified Modelling Language (UML) och ha kännedom om dess elementära byggstenar. De flesta element förklaras i samband med att de används men det är en fördel att tidigare ha studerat UML för att kunna följa med i resonemanget.

Det är även en fördel om läsaren har grundläggande programmeringskunskaper och känner till de olika faserna i programvaruutveckling.

1.6 Rapportens fortsatta disposition

För att läsaren skall få en uppfattning om vad rapporten innehåller ges här en sammanfattning av de kommande kapitlens innehåll.

Kapitel 2 - Metod
Kapitlet beskriver undersökningens olika delar och hur de har genomförts.

Kapitel 3 - Statecharts
Den centrala komponenten i arbetet är statecharts; i kapitel tre finns en introduktion till konceptet för läsare som inte har kunskap om det.

Kapitel 4 - Metoder för att utveckla statecharts
Det här kapitlet beskriver ett antal metoder programvarudesigners kan använda för att utveckla statecharts.

Kapitel 5 - Implementering
En metod för att utveckla statecharts har valts ut och i det här kapitlet beskrivs implementeringen av mjukvarumodellen och vilket resultatet blev.

Kapitel 6 - Resultat
I det här kapitlet beskrivs vilket resultatet av examensarbetet är, dessutom besvaras undersökningens problemformulering och syfte.

Bilaga A - Beteckningar
I den första bilagan finns en sammanställning på de beteckningar som används

i beräkningar i rapporten.

Bilaga B - Förkortningar
Om det är någon förkortning som läsaren är obekant med finns de förkortningar som används i rapporten presenterade i den här bilagan.

Kapitel 2

Metod

Metodkapitlets syfte är att beskriva hur undersökningen och implementeringen har genomförts, genom att presentera de metodologiska val som gjorts är tanken att läsaren skall få en uppfattning om hur undersökningens utförande samt dess eventuella svagheter avseende utformning, källkritik och noggrannhet.

2.1 Övergripande angreppssätt

Det här arbetet består av tre delar; inledningsvis genomfördes en teoretisk undersökning om olika metoder för att ta fram statecharts där fyra metoder redovisades och jämfördes. Därefter används en av metoderna för att ta fram en statechart till testmodellen som implementerades. Avslutningsvis gjordes en analys av de data som tagits fram i undersökningen och resultatet presenterades.

2.1.1 Teoretisk undersökning av metoder

Den här delen presenteras i kapitel 4 och består av genomgång av olika metoder för att ta fram statecharts. Metoderna varierar i grad av användarvänlighet och möjlighet till automatisering, vissa av metoderna klarar av att automatgenerera statecharts från specifikationer medan andra kräver att utvecklaren hjälper till.

Metoderna är hämtade från forskningsrapporter som presenterats på olika seminarier och konferenser. Beskrivningarna av metoderna saknar i regel självkritik och samtliga författare hävdar att deras metod är den bästa och att det är den som bäst löser problemet med att utveckla statecharts. Bristen på självkritik har gjort det nödvändigt att granska metoderna kritiskt för att upptäcka brister i dem. Stor ansträngning har lagts på att hitta metodernas svagheter.

Metoder har valts efter hur väl de kompletterar varandra, tanken är att de skall täcka upp ett brett spektra gällande möjlighet till automatisering, krav på indata och användarvänlighet. Resultatet är en sammanställning som tar upp metoder som passar för ett flertal olika typer av programvaruprojekt. Förutom en beskrivning av metoden innehåller varje utvärdering en sammanställning av metodens för- och nackdelar samt till vilken sorts projekt som den passar till. Utvärderingen har framarbetats dels från kritik i andra rapporter och dels från egna åsikter. Läsaren bör därför vara medveten om att författarens egna åsikter och värderingar är en del i rapporten och kan inverka på resultatet.

Källkritik

Det är viktigt att källorna till den information som används granskas, eftersom olika typer av källor har olika hög trovärdighetsgrad. En artikel i en vetenskaplig tidskrift har granskats av personer med stor kunskap inom ämnesområdet innan den publicerats och dessutom har artikeln utsatts för en kritisk granskning. De flesta böcker och tidskrifter har däremot inte genomgått denna vetenskapliga granskning.

Samtliga rapporter som använts i denna undersökning är skrivna av forskare och även om det inte på något sätt är en garanti för hög trovärdighet så ökar det källans trovärdighet. Vissa av källorna har även använts som en del författarens egen forskning vilket betyder att den har eller kommer att bli granskad vid en disputation eller liknande.

Den litteratur som använts är mestadels engelsk, det finns en risk i att översätta källor till svenska då den ursprungliga betydelsen kan försvinna. Detta har beaktats under arbetet och översättningen har skett med stor noggrannhet vid användandet av engelsk litteratur. Ett alternativ skulle ha varit att använda engelska citat direkt i uppsatsen, men för att öka läsbarheten av rapporten gjordes valet att översätta engelska källor.

2.1.2 Implementering

Även om rapporten till stor del koncentreras på utvecklingen av statecharts så har den större delen av arbetet bestått av att implementera testmodellen. I avsnittet om implementeringen beskrivs hur utvecklingen av mjukvarumodellen har gått till och hur det resulterande systemet fungerar.

Implementeringen har skett på plats hos TietoEnator med utvecklingsverktyget Rational Rose RealTime. Kontinuerligt under arbetet har det dokumenterats men det är endast det färdiga resultatet som redovisas i rapporten. Tonvikten läggs på högnivådesign samt på de algoritmer som beskriver hur mätvärden beräknas.

Implementeringen är gjord med hjälp av en statechart som togs fram med hjälp av en av metoderna som undersökts. Utifrån utvärderingen av metoderna valdes den som bäst passade för det här projektet och som inte innehöll något element som Rational Rose inte klarade av att modellera[1].

2.1.3 Analys

Analys består av att sammanföra teoretiska fakta med egna erfarenheter som framkommit under arbetet, i den här rapporten presenteras analysen i kapitel 4 och 6. I kapitel 4 presenteras analysen i samband med att metoderna för att ta fram statecharts utvärderas. Men det är framförallt i resultatkapitlet som analysen finns, resultatkapitlet är uppdelat dels i ett avsnitt om utveckling av statecharts och dels i ett avsnitt om implementeringen av testmodellen.

I det första avsnittet redogörs för de lärdomar och slutsatser som framkommit under arbetet med att utvärdera de fyra olika metoderna. Här beskrivs vilka förbättringar och förändringar som författaren anser bör göras för att metoderna skall bli effektivare och bättre fylla sitt syfte. Det är viktigt att utvecklingen av metoderna fortsätter, samtliga metoder har sina brister och kan förbättras.

I det andra avsnittet är det implementeringen av testmodellen som är i fokus, avsnittet redogör för vilka förbättringar som har åstadkommits och

[1] Rational Rose RT har vissa begränsningar, bland annat klarar den inte av alla element som ingår i specifikationen för statecharts.

vilka ytterligare förbättringar som är möjliga.

Analysen bygger i mycket hög utsträckning på författarens egna åsikter och erfarenheter vilket kan ses som en svaghet. Rapportens författare har inga formella kunskaper inom området, har inte gått någon kurs inom det eller forskat inom det. Däremot har författaren kontinuerligt under arbetet samlat in information och erfarenheter vilket har medfört att en kompetens inom området har utvecklats. Den här kompetensen har kombinerats med åsikter från andra forskare och det är detta som presenteras i analysavsnitten, en kombination av författarens och ett flertal andra forskares kunskap och åsikter inom området.

Kapitel 3

Statecharts

Det här kapitlet är en snabbintroduktion till UML-notationen statecharts, kapitlet redogör för dess olika byggstenar och går steg för steg igenom de element som används för att bygga upp en statechart.

3.1 Inledning

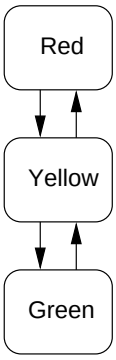
Statecharts introduceras 1987 av David Harel i ett försök att skapa en notation som bättre än befintliga metoder kunde beskriva reaktiva system. Namnet är taget från den enda oanvända kombinationen av *flow* och *state* med *diagram* och *chart*. Mycket förenklat kan man säga att statecharts är finita tillståndsmaskiner som har utökats med hierarkier, samtidighet och kommunikation [Har87].

I det här avsnittet används mestadels ett exempel med trafikljus för att visa hur statecharts är uppbyggda. Statechart är en *visuell metod* för att modellera ett system varför stor tonvikt läggs på illustrationer.

3.1.1 Tillstånd - states

Tillstånd är grundelementet i statecharts. Ett tillstånd är ett villkor under ett objekts livstid under vilken objektet utför en uppgift eller väntar på en händelse [OMG03]. Ett tillstånd innehåller delar av systemets historia, till

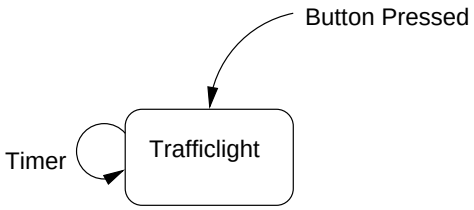
exempel har ett trafikljus tillståndet `Red` som talar om att just nu visas att rött sken. Efter att en tid passerat kommer trafikljuset att övergå till tillståndet `Yellow` och därefter till `Green`, mer om övergångar mellan tillstånd i avsnitt 3.1.3.



Figur 3.1: Exempel på enkla tillstånd.

3.1.2 Events

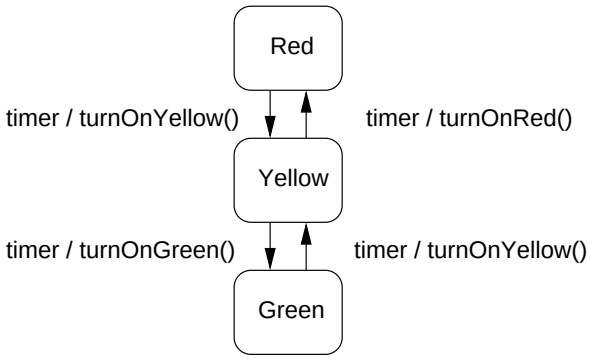
Ett event är en händelse som sker vid en specifik tidpunkt [Rum91], till exempel *användaren trycker ner en knapp på tangentbordet* eller *X2000 lämnar Stockholm Centralstation*. Det är event som förändrar systemets tillstånd och det är events som styr hur systemet skall bete sig. När man pratar om att ett reaktivt system reagerar på *intern och extern stimuli* är det events som menas med stimuli. I fallet med trafikljuset kan man tänka sig att ett externt event är att en fotgängare trycker på knappen som meddelar att han vill att ljuset skall slå om, en intern händelse är att en timer slår om och indikerar att ljuset skall ändras. Händelser kan även innehålla argument, till exempel kan eventet `buttonPressed` innehålla ett argument som talar om på vilken sida av gatan som fotgängaren står.



Figur 3.2: Interna och externa event

3.1.3 Transitioner

Transitioner är övergångar mellan tillstånd. En transition triggas alltid av att ett event har skickats till systemet. Namnet på transitionen är samma som beteckningen för det event som orsakar transitionen [OMG03]. Med varje transition finns det en funktion associerad som exekveras när transitionen sker. I exemplet med trafikljuset kan eventet `timer`[1] leda till att transitionen från tillståndet `Red` till `Yellow` utförs, funktionen som är associerad med tillståndet ser till att den gula lampan tänds och att alla andra lampor är släckta.



Figur 3.3: Exempel på transition

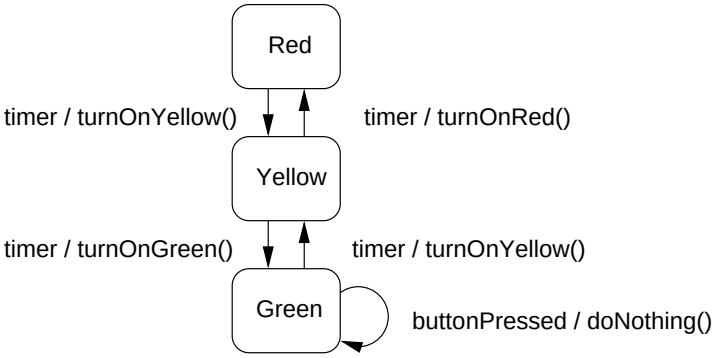
Självtransitioner

Ett specialfall av transition är *självtransition* vilket innebär att ett event har tagits emot men att systemet inte gör någon övergång till ett annat tillstånd. Om en fotgängare har tryck in knappen på stolpen skall inte trafikljusets färg ändras omedelbart utan styrsystemet skall bara ta hänsyn till att en fotgängare vill komma över gatan. Detta illustreras i figur 3.4.

3.1.4 Start- och sluttransitioner

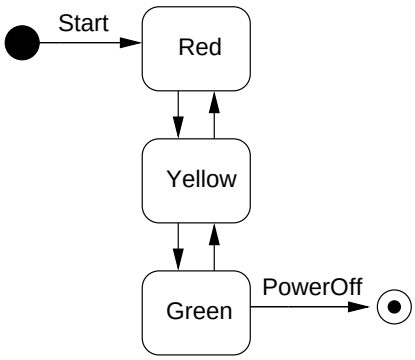
Varje statechart måste ha en *starttransition* som systemet kan starta från. Starttransitionen går från ett tillstånd som illustreras av en ifylld svart cirkel, tillståndet har inget namn och systemet gör övergången direkt när det startas. Starttransitionen innehåller vanligtvis funktionalitet för att initialisera variabler och tillstånd.

[1] Timer är ett event som indikerar att trafikljuset skall ändra färg.



Figur 3.4: Exempel på självtransition

En annan specialttransition är *sluttransitionen*, att ha en sluttransition är valbart till skillnad från starttransitionen [Sam02]. När systemet utför sluttransitionen avslutas exekveringen vilket medför att det inte får gå några transitioner från ett sluttillstånd. Ett sluttillstånd visas som en tom cirkel med en svart punkt inuti.[2]



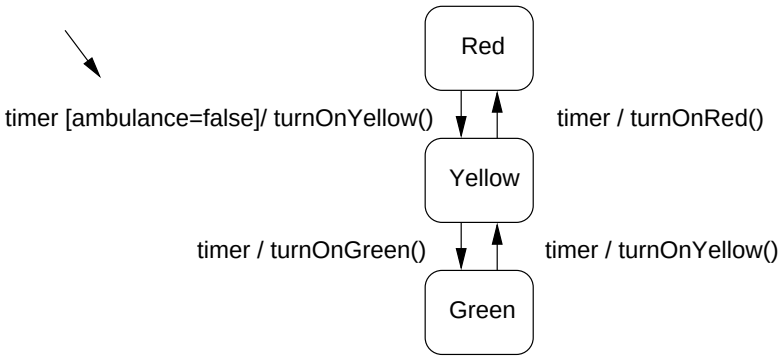
Figur 3.5: Start- och sluttransitioner

3.1.5 Guards/Vakter

En transition kan ha ett boolskt uttryck associerat till sig, om uttrycket är sant genomförs transitionen, annars inte [Sam02]. I exemplet med trafikljuset kan man tänka sig att SOS vid en ambulansuttryckning kan ställa in att trafikljuset visar rött tills dess att ambulansen har passerat, oavsett om

[2]I figur 3.5 går det bara att avsluta trafikljuset när det är i tillståndet `green`, det är endast för att förenkla exemplet. I ett riktigt system skulle hierarkier ha använts, mer om dem i avsnitt 3.1.8.

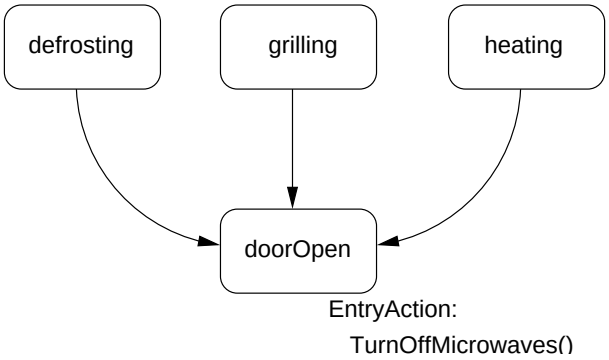
timern har gått ut eller inte.



Figur 3.6: Vakter

3.1.6 Entry- och exitactions

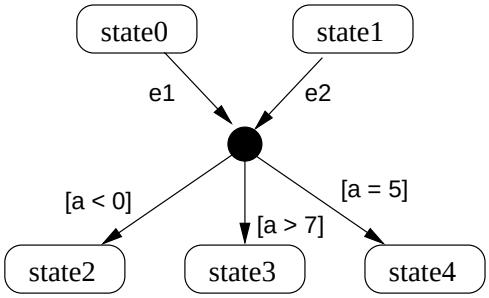
En stor programmeringsteknisk hjälp är entry- och exitactions. Det är funktioner som alltid exekveras när ett tillstånd nås eller lämnas. Funktionerna kan användas för att initiera tillståndet eller städa upp efter att tillståndet lämnats [Sam02]. Framförallt är detta användbart när man har variabler som måste initieras och är oberoende av från vilket tillstånd som transitionen sker. Här frångås exemplet med trafikljuset och istället används ett exempel med en mikrovågsugn som har ett tillstånd *doorOpen* som innebär att luckan till ugnen har öppnats. Oavsett från vilket tillstånd som övergången sker till `doorOpen` skall mikrovågorna stängas av när luckan öppnas. Med hjälp av en entryaction behövs den funktionaliteten endast implementeras en gång oavsett hur många tillstånd som är möjliga.



Figur 3.7: Entry- och exitactions

3.1.7 Choicepoints

I vanliga fall är det förutbestämt till vilket tillstånd ett system skall gå när ett event inträffar men om choicepoints används avgörs destinationstillståndet dynamiskt. En choicepoint är en knutpunkt dit flera transitioner kan sammanföras och där nästa tillstånd bestäms med hjälp av booleska uttryck för transitionerna [OMG03].



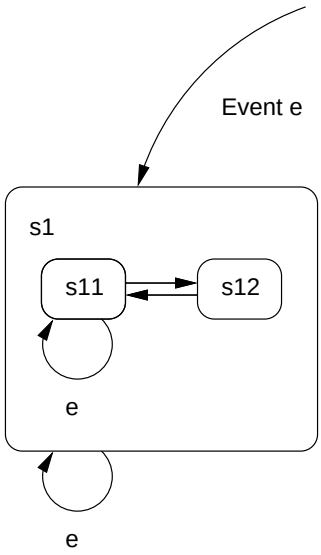
Figur 3.8: Exempel på choicepoint

I exemplet i figur 3.8 går två transitioner till en choicepoint och exekveringens fortsättning beror på variabeln `a`. Noterbart är även att om `a` är lika med 6 kommer ingen transition att göras utan exekveringen fortsätter vid föregående tillstånd.

3.1.8 Hierarkiska statecharts

Den viktigaste skillnaden med statecharts jämfört med de klassiska tillståndsmaskinerna är introduktionen av *hierarkiska nästlade tillstånd* [Sam02].

Tillstånd som innehåller andra tillstånd kallas för *sammansatta tillstånd*, omvänt kallas tillstånd utan intern struktur för *enkelt tillstånd* [Sam02]. Tanken är enligt figur 3.9 att om ett system är i tillstånd s1, är det även i tillstånd s11. Fördelen med hierarkiska tillstånd är att endast *skillnaden* mellan ett tillstånd och dess omgivande tillstånd behöver implementeras. Finns det inga skillnader kan subtillståndet låta nästa steg i hierarkin ta hand om ett event. När ett event inträffar kommer systemet först att försöka hantera det i det innersta tillståndet, finns det ingen matchande transition försöker den ett steg utåt tills alla nivåer är provade. Om Event e skickas till systemet i figur 3.9 som är i tillstånd s11 kommer det att tas om hand av s11. Däremot om systemet är i tillstånd s12 som inte har någon mat-



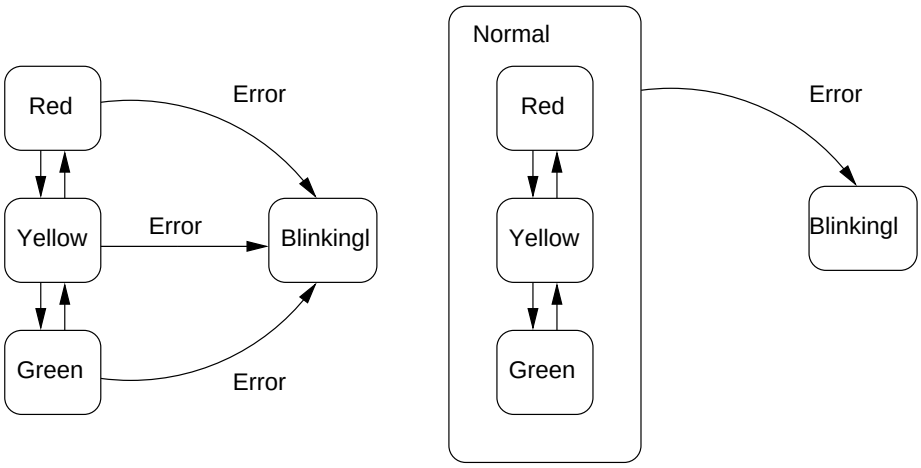
Figur 3.9: Hierarkiska statecharts

chande transition så kommer tillståndet i nivån utanför, s1, att ta hand om transitionen.

Hierarkiska statecharts kan även minska antalet transitioner. Återigen används exemplet med trafikljuset, man tänka sig en transition som sker när det blir något fel på trafikljuset som gör att trafikljuset övergår till tillståndet `Blinking` oavsett tidigare tillstånd. Utan hierarkier krävs det tre transitioner för att visa detta, medan det med hierarkier endast krävs en (se figur 3.10). Fördelen blir än tydligare om man tänker sig att man har 100 tillstånd som alla kräver en transition till `Blinking`, med hierarkier räcker det *alltid* med en enda transition.

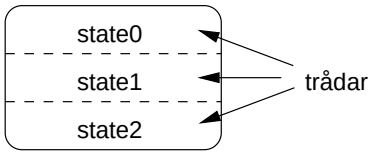
3.1.9 Samtidighet

En annan stor skillnad mellan statecharts och tillståndsmaskiner är möjligheten att modellera för samtidig exekvering (concurrency). Ofta har ett system flera subsystem som måste kunna fungera samtidigt oberoende av varandra, en bil har till exempel växellåda, motor och bromsar vilka alla måste arbeta parallellt. I statecharts modelleras samtidighet med en streckad linje mellan de exekverande regionerna [Rum91]. Tillstånd som kan exekveras samtidigt brukar ibland även kallas för *trådar*.



Figur 3.10: Exempel på hierarkier

Samtidighet benämns ibland som ett *AND-förhållande* mellan tillstånd, två tillstånd som inte kan exekveras samtidigt har på samma sätt ett *OR-förhållande*.



Figur 3.11: Exempel på samtidighet

Kapitel 4

Metoder för att utveckla statecharts

I det här kapitlet kommer fyra metoder för att skapa statecharts beskrivas och diskuteras. Metoderna har olika krav på indata och olika möjligheter till automatisering, för någon metod finns det redan utvecklingsverktyg medan andra är tänkta att användas manuellt av mjukvarudesigners. Metoderna är hämtade från olika forskningsrapporter där författarna har beskrivit sina egna metoder.

Beskrivningarna avslutas med en diskussion kring metodens styrkor och svagheter samt till vilka användningsområden som de passar.

4.1 Metod 1: KEK

Den första metoden som tas upp är *KEK*, namnet är uppbyggt efter utvecklarnas förstabokstav i efternamnet: Khriss, Elkoutbi och Keller. Metoden är en blandning av redan kända algoritmer och nya som författarna har utvecklat. Enligt författarna är tanken att metoden en dag skall vara automatiserad, det skall vara möjligt att utifrån indata automatiskt datorgenerera korrekta statecharts. Som metoden ser ut idag är det en bit kvar till fullständig automatisering men vissa delar av algoritmen är implementerad och fungerar. Metoden introducerades 1998 i [KEK98].

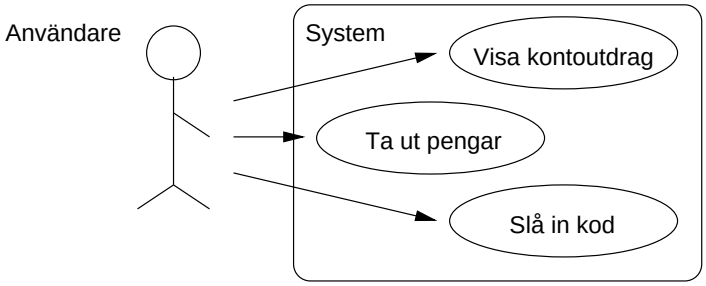
KEK startar med kravanalysen för systemet, kravanalysen är i stort sett en

kopia av Rumbaughs metodik [Rum91] för att utveckla kravspecifikationer. I den här jämförelsen är KEK den enda metod som tar upp utveckling av kravspecifikation, övriga metoder utgår från att en specifikation existerar sedan tidigare.

KEK är uppbyggd i fyra steg som i sin tur är indelade i ett antal delsteg. De fyra huvudstegen är: *kravanalys*, *generering av objektspecifikationer från scenarier*, *analys av objektspecifikationer* samt *integrering av objektspecifikationer*, nedan förklaras respektive steg närmare.

Kravanalys

I det första steget av kravanalysen utvecklar designern use-casediagram för systemet. Ett use-casediagram visar sambanden mellan systemet och objekt som interagerar med det [KEK01]. Figur 4.1 visar interaktionen mellan en uttagsautomat och en användare.



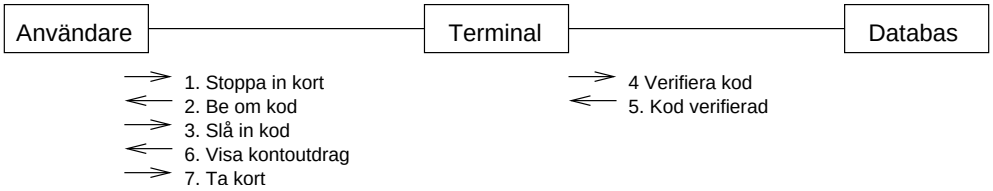
Figur 4.1: Use-case för en uttagsautomat och en användare.

Därefter tas klasser fram för systemet, för varje klass utarbetas vilka dess attribut och metoder skall vara. Avslutningsvis skapar designern ett *collaborationdiagram* för varje use-case, ett collaborationdiagram visar anropsstrukturen mellan klasser och i vilken ordning som anropen sker [KEK01]. Collaborationsdiagrammet för att göra ett kontoutdrag skulle kunna se ut som i figur 4.2, det här use-caset innehåller tre klasser, en användare som vill ha ett kontoutdrag, terminalen som visar kontoutdraget samt en databas som verifierar användarnas kod.

Kravanalysen är helt manuell och går inte att överlåta till automatiseringsverktyg. Om det finns flera use-case kommer en kravanalys att göras för varje use-case.

Generering av objektspecifikationer från scenarier

I metodens andra steg används de collaborationdiagram som togs fram i



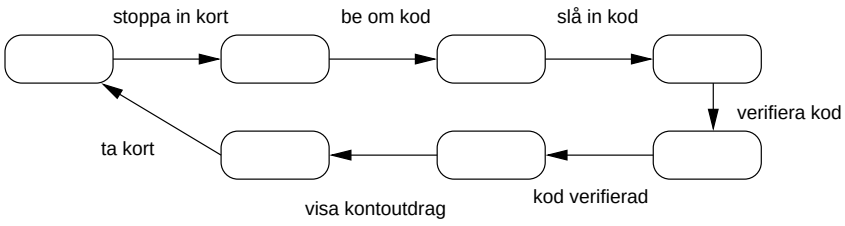
Figur 4.2: Collaborationsdiagram för use-case visa kontoutdrag.

föregående punkt för att skapa statecharts. Algoritmen för att göra det är följande:

1. Skapa en tom statechart för varje klass som beskrivits i collaboration-diagrammen.
2. Gör om de variabler som inte är attribut i något objekt till tillstånds-variabler, alltså variabler som beskriver vilket tillstånd systemet är i.
3. Skapa transitioner för de klasser som skickar och tar emot meddelanden,
4. Skapa tillstånd som kopplar samman transitionerna.

Den här delen av algoritmen kallas för *Generation of Partial Statediagrams* (GPS) och det finns fungerande implementeringar som automatiskt omvandlar collaborationdiagram till statecharts [SKK01].

För collaborationsdiagrammet för terminalen i figur 4.2 kommer statecharten att få följande utseende:



Figur 4.3: Collaborationsdiagram för use-case visa kontoutdrag.

Analys av objektspecifikationer

Tillstånden i de statecharts som utvecklats i punkten ovan har ännu inte givits några namn, något som kommer att krävas i det sista steget av algoritmen när statecharten slås ihop till ett enda statechart.

För att kunna ge tillstånden namn krävs förutom de tillståndsmaskiner som skapades i föregående steg även en *OCL-specifikation* för systemet. OCL-specifikationen beskriver vilka villkor som måste vara givna före och efter en transition. Till exempel har en bankomat ett krav på att ett giltigt bankomatkort är instoppat för att ett uttag skall tillåtas.

Algoritmen som används heter *Analysis of Partial Statediagrams* börjar med att ett antal *potentiella tillstånd* skapas, de potentiella tillstånden utvecklas från de tillståndsvariabler som togs fram i kravspecifikationen. Ett system kan till exempel ha tillståndsvariablerna a_1 och a_2 vilka antingen kan vara större än noll eller mindre än eller lika med noll. De potentiella tillstånden blir alla kombinationer av variablerna:

$S_1 = a_1 \leq 0, a_2 \leq 0$
$S_2 = a_1 \leq 0, a_2 > 0$
$S_3 = a_1 > 0, a_2 \leq 0$
$S_4 = a_1 > 0, a_2 > 0$

Därefter provas tillstånden mot systemet i en djupet-först-sökning. Algoritmen provar de potentiella tillstånden mot OCL-specifikationen, om det är något villkor som tillståndet inte uppfyller backar algoritmen ett steg och försöker en annan väg. Det här upprepas tills systemet har fått en uppbyggnad som alla stämmer med OCL-specifikationen.

Syftet med den här delen av metoden är att ge likadana tillstånd i olika statecharts samma namn, detta krävs när flera statecharts skall slås ihop till den slutgiltiga statecharten.

Integrering av objektspecifikationer

Det här är det sista steget i KEK och det består av att slå samman alla statecharts. Algoritmen sammanför statecharts två och två tills det bara finns ett statechart kvar. Under sammanslagningen görs kontroller av att resultatet följer systemets specifikation.

Algoritmen för integreringen består av följande fem steg:

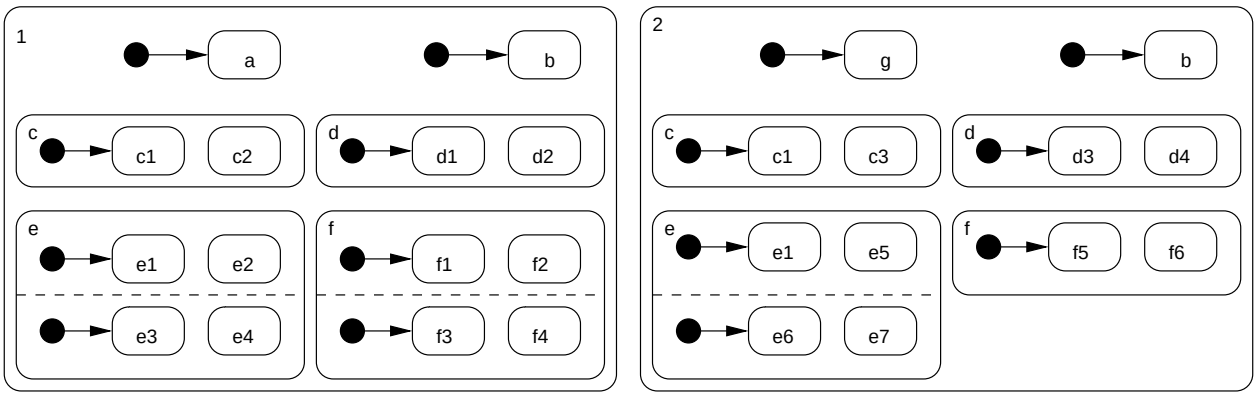
1. Kontroll av tillstånd. Om det finns tillstånd med samma namn på olika nivåer i två statecharts måste designern rätta till detta. Om det enligt specifikationen är tillåtet att flytta ett av tillstånden till det andra tillståndets nivå görs det, annars döps ett av tillstånden om så att tillstånden anses vara skilda.

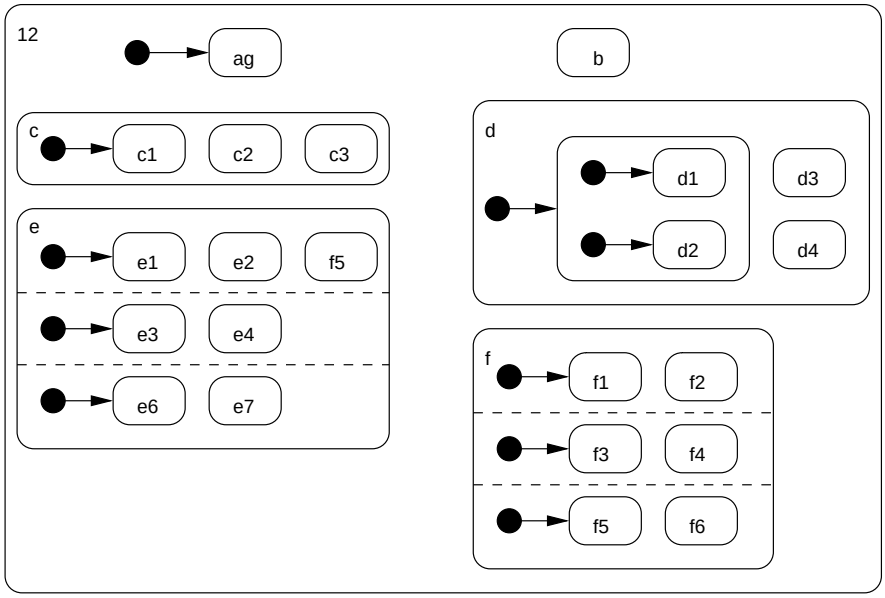
2. Sammanföra tillstånden. När det inte längre finns några kollisioner mellan tillstånden skall de sammanföras. Det finns tre fall att ta hänsyn till:

   (a) De två tillstånden har ett OR-förhållande. Om deras starttillstånd är samma tas unionen mellan tillstånden, detta är fallet för nivå `c` i figur 4.4 (den sammanslagna figuren finns på nästa sida). Är starttillstånden inte samma konstrueras ett nytt tillstånd som innehåller båda de tidigare tillståndens egenskaper. Det här är fallet för den yttersta nivån där `a` och `g` kolliderar.

   (b) Tillstånden har ett AND-förhållande. Om de har samma initialtillstånd tas unionen mellan tillstånden, detta visas när nivå `e` slås samman och e1 är starttillstånd. I annat fall adderas en ny tråd till tillståndet vilket är fallet med de övriga trådar i `e`.

   (c) Ett av tillstånden har ett OR-förhållande, det andra har ett AND-förhållande. Lösningen är att se tillståndet med OR-förhållande som ett AND-förhållande med bara en tråd. Då kan man slå ihop tillstånden på samma sätt som i punkten före, vilket visas när `f` slås samman.



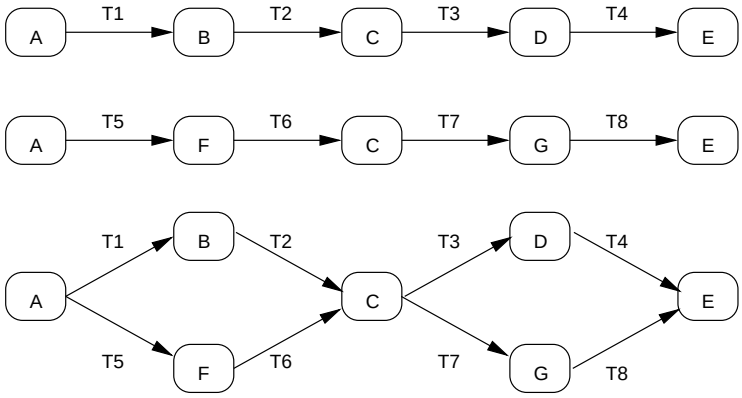


Figur 4.4: Exempel på sammanslagning av tillstånd.

3. Sammanför transitionerna. I det föregående steget sammanfördes tillstånd, i det här steget skall transitionerna sammanföras vilket är betydligt enklare. Två transitioner anses vara lika om de startar från samma tillstånd, går till samma tillstånd och har samma event. Om de inte är identiska skapas två stycken skilda transitioner.

4. Ta bort ogiltiga exekveringsordningar. Ett problem som uppstår när flera scenarier skall integreras är *övergeneralisering* vilket innebär att resultatet tillåter mer än vad den ursprungliga specifikationen gjorde. Ser man på illustrationen i figur 4.5 så tillåter resultatet av sammanslagningen exekveringsordningen $T_1 \rightarrow T_2 \rightarrow T_7 \rightarrow T_8$, något som inte var tillåtet enligt de ursprungliga exekveringsordningarna. KEKs lösning på det här problemet är att skapa en lista med alla tillåtna exekveringsordningar (enligt de ursprungliga scenariospecifikationerna), för att en transition skall vara tillåten måste den ingå i listan. Listan kan ses som en extra guard för varje transition.

5. I det sista steget sker en avslutande verifiering av att den resulterande statecharten stämmer med de premisser som sattes upp i kravspecifikationen. Bland annat verifieras att varje tillstånd har en transition för varje möjligt indata och att de klasser som skapats stämmer överens med vad som angivits i specifikationen. Om det är någonting som inte stämmer rättar designern till felet manuellt.



Figur 4.5: Exempel på övergeneralisering.

4.1.1 Utvärdering

Författarna hävdar i [KEK01] att deras metod är helt automatiserbar och att det är en av metodens största fördelar. Det här stämmer dock inte, som nämnts i texten ovan så är till exempel framtagandet av kravspecifikation helt manuellt och integreringen delvis manuell. Även om metoden har flera moment som är automatiserade är den en lång väg från fullständig automatisering.

Den största nackdelen med metoden är att den inte inför hierarkier, den klarar av att slå samman hierarkiska statecharts men metoden beskriver inte hur hierarkier skall skapas. Troligtvis är tanken att hierarkierna skall skapas i steg två när collaborationdiagrammen görs om till statecharts men det finns ingen beskrivning på hur detta skall gå till. Hierarki är kanske det viktigaste elementet i statecharts vilket gör att den här metoden har en mycket stor nackdel. Det som är mest anmärkningsvärt är att författarna använder en algoritm som klarar av att hantera sammanslagning av hierarkiska statecharts tillsammans med en algoritm som inte klarar av att introducera hierarkier.

De två största fördelarna med KEK är att metoden förenklar utveckling av kravspecifikationen och att den använder flera välkända algoritmer vars korrekthet har verifierats. Genom att använda kända algoritmer som är granskade ökar säkerheten kring metoden.

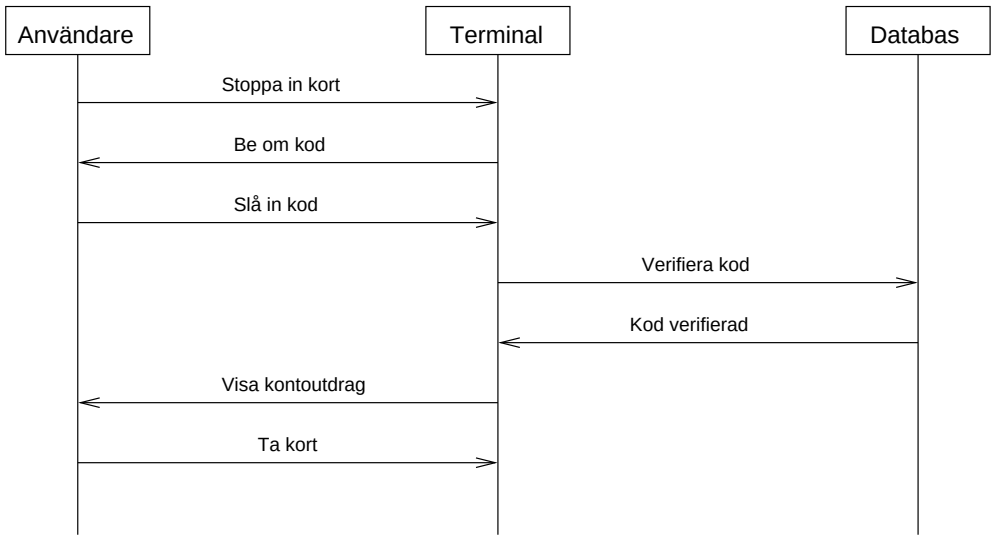
4.1.2 Rekommendation

Jag skulle inte rekommendera någon att använda den här metoden. Så länge problemet med hierarkier inte är löst finns det ingen anledning att använda metoden. Hierarkier är en allt för viktig del av statecharts för att utelämnas. Det är synd att utvecklarna har ignorerat problemet eftersom övriga delar av metoden är väl utvecklade. Till dess att hierarkier har lagts till metoden har jag svårt att se något användningsområde för den.

4.2 Metod 2: VT

En metod som däremot innehåller alla viktiga komponenter hos statecharts men som saknar möjlighet till automatisering är *VT* utvecklad av Vasilache och Tanaka [VT01]. Metoden är betydligt mer intuitiv än KEK vilket främst beror på att den är utvecklad för att användas manuellt av programvarudesigners.

Att ta fram krav för modellen ligger utanför metodens räckvidd, det antas redan vara gjort. Modellen utgår från *scenarios*, ett scenario representerar ett möjligt exekveringsfall av modellen. En bankomat kan till exempel ha scenarierna *uttag av pengar* och *felaktig kod inslagen*. I figur 4.6 visas ett exempel med scenariot `visa kontoutdrag` som visualiseras med ett sekvensdiagram.



Figur 4.6: Exempel på scenario.

Skillnaden mellan sekvensdiagram och collaborationdiagram som presenterades i KEK är mycket liten, de innehåller i princip samma element och de kan åskådliggöra samma funktioner. Den främsta skillnaden ligger i att sekvensdiagram har en tidslinje medan collaborationdiagram anger exekveringsordningen med hjälp av numrerade meddelanden. Anledningen till att VT använder sekvensdiagram är att utvecklarna anser att den finns bättre formellt beskriven.

Ett scenario är endast en delbeskrivning av ett system, för att kunna beskriva hela systemet krävs ett flertal scenarier. Men det är inte enbart scenariernas uppbyggnad som är viktig, utan även i *vilken ordning* som de exekveras, ett scenario kan till exempel behöva utdata från ett annat scenario. VTs största skillnad mot andra metoder är att den lägger stor vikt på i vilken ordning scenarierna får exekveras. Exekveringsordningen av scenarier faller inom någon av följande kategorier: *följd*, *disjunktion*, *konjunktion* eller *upprepning*. Följd innebär att ett scenario måste följa efter ett annat, disjunktion att endast ett av två eller flera scenarier får exekveras vid en tidpunkt, konjunktion att flera scenarier exekveras samtidigt och upprepning att ett scenario exekveras flera gånger i en följd. Närmare beskrivning av kategorierna finns i algoritmbeskrivningen nedan.

Algoritmen för att utveckla statecharts består av sex steg, i det första steget skapas en statechart för varje scenario, i steg två till fem slås scenarierna samman och i det sjätte och sista steget införs hierarkier. Nedan följer en mer detaljerad presentation av varje steg.

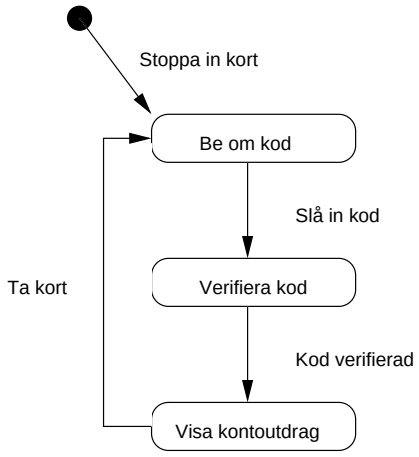
Skapa en statechart för varje scenario

I det första steget skapas ett statechart för varje scenario. Designern undersöker sekvensdiagrammet för det objekt som skall modelleras, inkommande pilar representerar events och de blir transitioner, utgående pilar är funktioner som exekveras när transitionen sker. Intervallen mellan events blir tillstånd. Tillstånden ges namn efter de utgående pilarna. Ett exempel på hur ett statechart kan skapas visas i figur 4.7, exemplet beskriver scenariot *visa kontoutdrag* hos en bankomat, objektet som skall modelleras är `terminal`.

Till skillnad från KEK där flera statecharts skapades för varje use-case skapar VT bara en statechart för varje scenario.

Införa följdförhållanden

Efter att ha utvecklat ett statechart för varje scenario skall de slås sam-



Figur 4.7: Statechart för scenariot visa kontoutdrag.

man till ett slutgiltigt statechart, sammanslagningen sker i de fyra steg som beskrivs i de kommande avsnitten.

Det första steget är att identifiera vilka scenarier som har ett följdförhållande, alltså ett förhållande där ett scenario måste föregå ett annat. Lösningen är att det senare scenariot ses som en fortsättning på det första scenariot och placeras efter det.

Om de två scenarierna innehåller likadana transitioner skall transitionerna slås ihop. Det är upp till designern att lösa problemet, metoden ger inga riktlinjer på hur det skall gå till.

Införa disjunktioner

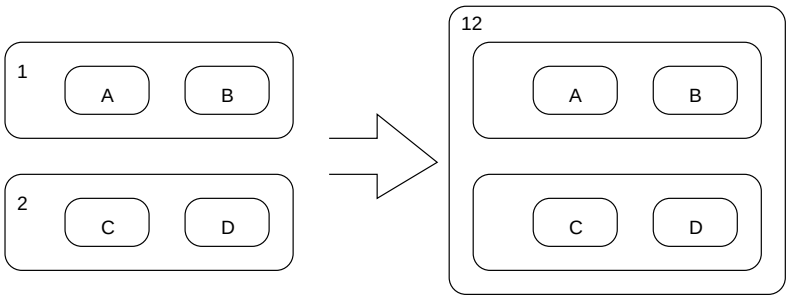
Disjunktioner innebär att endast ett av flera scenarier får exekveras vid en tidpunkt. Om två scenarier är disjunkta införs ett sammansatt tillstånd som består av de två scenarierna enligt figuren nedan. De två statecharten får ett OR-förhållande vilket visas i figur 4.8.

Införa konjunktioner

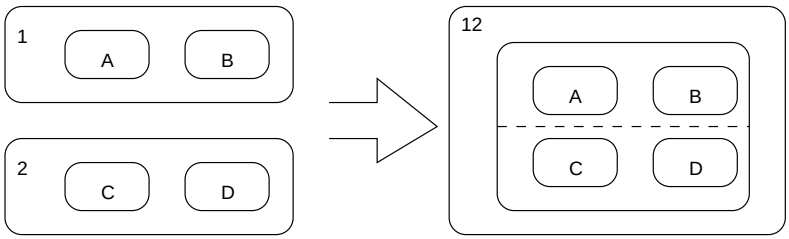
Konjunktioner innebär att flera scenarier exekveras samtidigt, konjunktioner behandlas på ett liknande sätt som disjunktioner. Det skapas ett sammansatt tillstånd med AND-förhållande med två trådar, en för vardera scenario som i figur 4.9.

Införa upprepningar

Upprepningar av ett scenario modelleras med en självtransition till starttill-



Figur 4.8: Sammanslagning med disjunktioner.



Figur 4.9: Sammanslagning med konjunktioner.

ståndet. För att ställa hur många gånger som scenariot skall iterera införs en variabel som räknas upp för varje varv. Variabeln fungerar som en guard för transitionen som ser till att scenariot inte kan startas efter sista iterationen.

Införa hierarkier

Vid det här steget har samtliga statecharts slagits ihop till ett statechart, nu skall hierarkier införas. Designern granskar statecharten och letar efter tillstånd och transitioner som förekommer på flera ställen. Om det är möjligt försöker han med hjälp av hierarkier att gruppera dem så att bara en instans av tillståndet eller transitionen finns.

När designern anser att det inte finns några förbättringar att göra är VT avslutat.

4.2.1 Utvärdering

Den största fördelen med VT är dess enkelhet, metoden är både enkel att förstå och enkel att använda. De enda indata som krävs för att använda algoritmen är de scenarier som systemet skall hantera. Dock krävs mycket arbete från designern, metoden ger inte mycket hjälp vid utformningen av

statecharten.

Utvecklarna av metoden skriver i slutet på sin beskrivning av algoritmen att de planerar att implementera en programvara som möjliggör automatisering av metoden. Dock är det en lång väg innan metoden är automatiserad, metoden har ingen formell beskrivning utan finns bara som löpande text. Det är betydligt svårare att automatisera en sådan metod jämfört med en som använder en mer matematisk notation.

En nackdel med den här metoden är att man som designer får väldigt lite hjälp med utvecklingen. VT fungerar mer som en metodik för att ta fram statecharts än en algoritm. Det krävs stor kunskap om statecharts för att designern skall kunna använda metoden.

Ytterligare en nackdel är att VT inte innehåller någon resultatkontroll. Som nämndes i beskrivningen av KEK kan det uppstå problem när flera scenarier slås ihop. VT tar inte hänsyn till problemet och nämner inte på något ställe hur verifiering av resultatet skall ske.

4.2.2 Rekommendation

Jag skulle rekommendera den här metoden till designers som har goda kunskaper inom statecharts och som bara behöver vägledning vid utvecklingen. Metoden passar inte för stora komplexa projekt, det blir allt för svårt att till exempel införa hierarkier med den här metoden.

4.3 Metod 3: WS

En metod som redan bevisat att den fungerar är *WS*, utvecklad av Whittle och Schumann år 2000. Whittle och Schumann har utifrån sin metod även implementerat ett verktyg som automatiskt kan omvandla scenarier till statecharts. Verktyget har till exempel använts för verifiering av statecharts vid utvecklingen av ett trafikkontrollsystem för flygplatser.

WS kräver på samma sätt som KEK att utvecklaren tillhandahåller sekvensdiagram och OCL-specifikationer för systemet. Utöver detta måste fyra parametrar ställas in när introduceringen av hierarkier skall ske, mer om dessa parametrar i det sista avsnittet.

Den här algoritmen är utvecklad för automatisk omvandling mellan sekvensdiagram och statecharts vilket gör den något komplicerad, därför kommer endast huvuddragen av metoden tas upp i den här rapporten, den kompletta metoden redogörs i [WS00] och [WS02].

Metoden är uppbyggd i fyra steg: *upptäcka konflikter*, *omvandling till tillståndsmaskin*, *integrering* samt *introduktion av hierarkier*.

Upptäcka konflikter

Det första steget i omvandlingen från sekvensdiagram till statecharts är att upptäcka eventuella konflikter mellan sekvensdiagrammen och OCL-specifikationen. OCL-specifikationen för att visa kontoutdrag hos en bankomat kan se ut på följande sätt:

```
Display main screen:
   pre: cardHalfway = false, cardIn = false
   post:
Insert card:
   pre: cardIn = false
   post: cardIn = true, card = c
Request password:
   pre: passwordgiven = false
   post:
Enter password:
   pre: passwordGiven = false
   post: passwordGiven = true, password = p
Eject card:
   pre: cardIn = true
   post: cardHalfway = true, cardIn = false
         card = null, password = p
Request take card:
   pre: cardHalfway = true
   post:
Take card:
   pre: cardHalfw:Way = true
   post: cardHalfway = false, cardIn = false
```

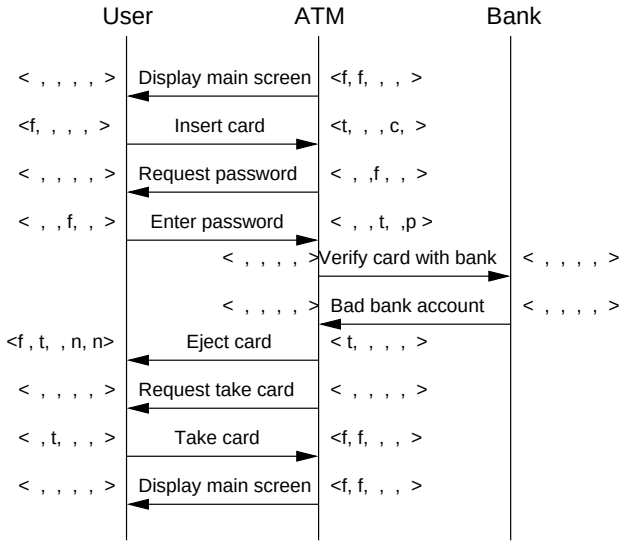
Figur 4.10: Exempel på OCL-specifikation.

Systemet har fem variabler: `cardIn` avgör om ett bankomatkort är instoppat, `cardHalfway` om kortet är halvvägs in, `passwordGiven` om lösenord har angetts, `card` är ett objekt för kortet som är instoppat samt `password` som är det angivna lösenordet. Det är dessa variabler som styr vilket tillstånd som systemet är i. Konfliktdetekteringen startar med att samtliga pre- och postvillkor förs in i sekvensdiagrammet för varje meddelande. Variablerna

presenteras på följande form:

```
<cardIn, cardHalfway, passwordGiven, card, password>
```

Platser som är tomma i diagrammet anger att det inte spelar någon roll vilket värde variabeln har, n står för NULL.



Figur 4.11: Konflikthantering, bild 1.

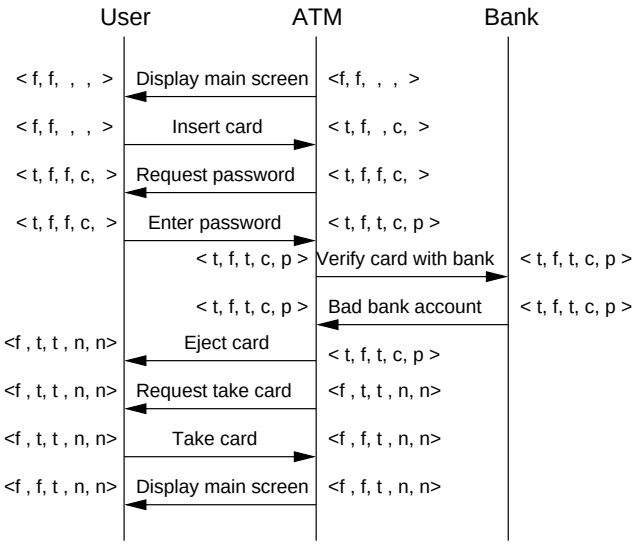
Därefter vidarebefordras tillståndsvektorn framåt i sekvensdiagrammet, platser i tillståndsvektorn som ännu inte är ifyllda fylls i med variabler från tillståndsvektorn som ligger före i sekvensdiagrammet enligt figur 4.12:

När slutet på sekvensdiagrammet nåtts vidarebefordras tillståndsvektorn till början och arbetet börjar om. Det här pågår tills det inte finns några fler tilldelningar att göra vilket illustreras i figur 4.13.

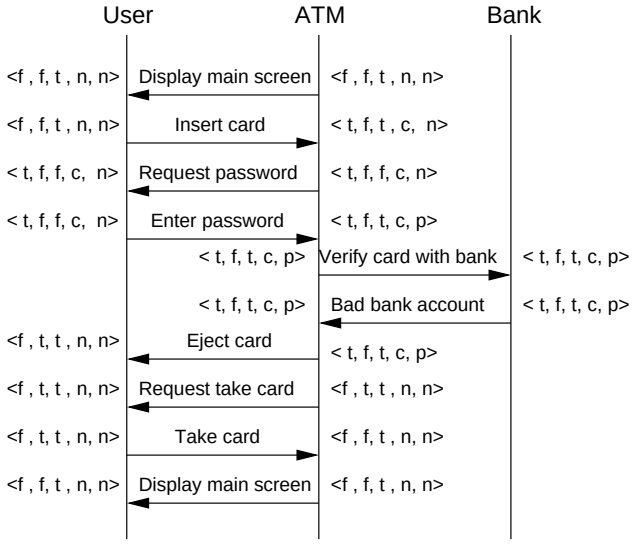
I det sista steget görs konfliktkontrollen. En konflikt upptäcks om en tillståndsvektor som följer direkt efter ett meddelande tagits emot och tillståndsvektorn omedelbart före nästa meddelande skickas inte är lika. Konflikter måste lösas av designern som därefter får starta från början med algoritmen. I figur 4.14 visas upptäckten av en konflikt.

Det är upp till designern att på något sätt lösa konflikten, WS ger ingen vägledning till hur det skall göras.

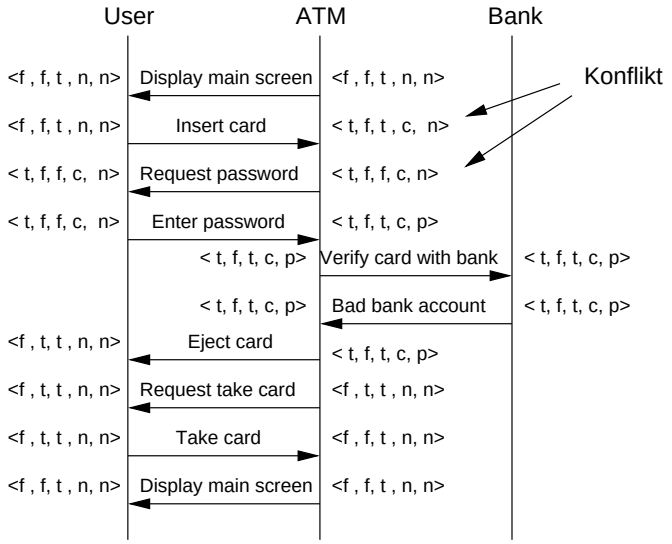
Omvandling till tillståndsmaskin



Figur 4.12: Konflikthantering, bild 2.



Figur 4.13: Konflikthantering, bild 3.



Figur 4.14: Konflikthantering, bild 4.

I det här steget överförs varje sekvensdiagram till en tillståndsmaskin, en tillståndsmaskin är i stort sätt en statechart utan hierarkier och samtidighet.

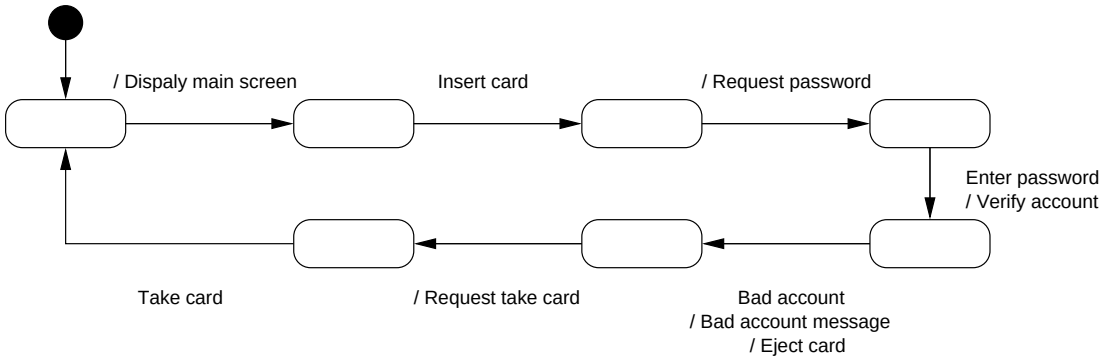
Vid omvandlingen undersöks ett av objekten i sekvensdiagrammen enligt följande två regler:

1. Pilar som är riktade mot objektets tidslinje omvandlas till events.
2. Pilar som är riktade från objektets tidslinje omvandlas till transitioner.

Algoritmen inleds med att skapa ett starttillstånd, därefter följer algoritmen exekveringsordningen hos sekvensdiagrammet och skapar actions och events enligt reglerna ovan.

Tillståndsmaskinen för ATM i sekvensdiagrammet i figur 4.14 visas i figur 4.15.

Algoritmen skapar en tillståndsmaskin för varje sekvensdiagram, meddelanden som har ett '/'-tecken framför sig betyder att det är en funktion som skall utföra meddelandets uppgift, övriga meddelanden är namn på event.



Figur 4.15: Tillståndsmaskin.

Integrering

Efter föregående steg har ett antal tillståndsmaskiner skapats, nästa steg är att slå samman de till en enda tillståndsmaskin. Sammanslagningen baseras på identifikation av *liknande tillstånd* i tillståndsmaskinerna. Två tillstånd anses vara lika om:

- De har samma tillståndsvektor, tillståndsvektorn togs fram från OCL-specifikationen i det första steget av algoritmen.
- De har minst en inkommande transition med samma namn.

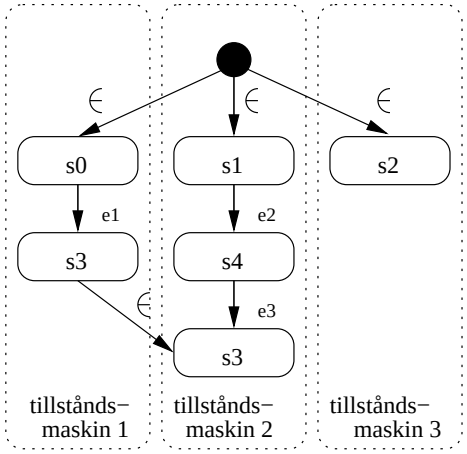
Algoritmen för integreringen slår ihop alla tillståndsmaskiner på en gång. Först skapas ett tomt nytt statechart med bara ett starttillstånd, därefter kopplas samtliga tillståndsmaskiner ihop med starttransitionen med tomma ϵ-transitioner vilket visas i figur 4.16.

Alla tillstånd som anses vara lika enligt definitionen ovan kopplas därefter ihop med ϵ-transitioner. Resultatet är en icke-deterministisk tillståndsmaskin. Med hjälp av en variant av standardalgoritmen för att omvandla en icke-deterministisk tillståndsmaskin till en deterministisk tillståndsmaskin tas *de flesta* av ϵ-transitionerna bort. Det är upp till designern att designa om systemet så att eventuella kvarvarande ϵ-transitionerna försvinner.

Introduktion av hierarkier

Vid införandet av hierarkier får designern ställa in följande fyra parametrar:

1. **Maximalt djup** - För många nästlade hierarkier leder till en svårläst statechart, en alltför platt statechart blir rörig och svåröverskådlig.



Figur 4.16: Exempel på integrering av flera tillståndsmaskiner.

Det är upp till designern att ställa in vilken nivå som passar deras projekt.

2. **Maximalt antal tillstånd per nivå** - Den här restriktionen kan krocka med den som beskrevs i föregående punkt, den beskriver hur många tillstånd som maximalt får finnas på varje nivå.

3. **Maximalt antal transitioner mellan olika nivåer** - Många övergångar mellan olika nivåer begränsar modulariteten hos systemet, med den här parametern kan designern ställa in hur många procent av systemets transitioner som maximalt får ske mellan olika nivåer.

4. **Prioriteten hos tillståndsvariabler** - Den här parametern bestämmer i vilken ordning som tillståndsvariabler skall undersökas. Parametern anger hur algoritmen skall försöka dela upp statecharten.

Utifrån dessa parametrar har Whittle och Schumann skapat en algoritm som lägger till hierarkier till tillståndsmaskiner. Algoritmen provar sig fram och försöker dela upp tillståndsdiagrammet på olika sätt tills den anser att den inte kan åstadkomma en bättre lösning avseende de parametrar som är givna. I vissa fall kan det hända att något av de kriterier som ställts upp inte kan uppfyllas och då kommer en kompromiss att göras där den lösning som bäst stämmer med parametrarna väljs.

4.3.1 Utvärdering

En stor fördel med metoden är att den innehåller utförliga felkontroller. När konflikter upptäcks får designern lösa dem genom att antingen ändra i sekvensdiagrammen eller OCL-specifikationen. Tyvärr finns det väldigt lite hjälp i metoden till hur konflikterna skall lösas, det är upp till designern att själv hitta en lösning.

En annan fördel är att metoden bevisligen fungerar. Författarna har bland annat gjort ett test på ett flygkontrollsystem som skulle användas på flera amerikanska storflygplatser. Testet utfördes efter att systemet redan var designat varför det mest blev ett test på om metoden kunde utföra ett liknande arbete som utvecklarna. Enligt författarna var försöket lyckat och de statecharts som genererades med deras programvara fungerande lika bra som de som designarna hade utvecklat [WS02+]. Det är dock klokt att betrakta sådana argument med en viss skepsis, det är inte ovanligt att forskare försöker sätta sin egen metod i allt för god dager.

En nackdel med metoden är att den är komplex, den består av många delsteg som var för sig är komplicerade. Allt tillsammans gör WS till en svår metod att använda, det krävs stort arbete för att sätta sig in i metoden och ännu mer för att kunna använda den.

4.3.2 Rekommendation

Vid kontakter med författarna har de tyvärr inte velat dela med sig av det verktyg som de utvecklat utifrån sin metod. Om verktyget fungerar som de uppger tror jag att det kan vara till en stor hjälp vid utveckling av stora programvarusystem. Jag tror däremot inte på användning av deras metod utan automatisering, det finns andra metoder som är mer lättförståliga och enklare att ta till sig. Den här metoden är utvecklad för automatisering och det är på det sättet som den bör användas.

Jag rekommenderar användning av WS vid stora projekt där man antingen kan få tag på författarnas prototyp eller där man har möjlighet att själv utveckla en liknande programvara. I stort sett samtliga av deras algoritmer finns tillgängliga för att på egen hand göra en implementation av metoden. Med tanke på metodens komplexitet kommer det dock att krävas en hel del tid och resurser för att lyckas med det. Därför tror jag endast att det är

större företag som kan göra en implementering. Ett stort företag kan ta del av resultaten i flera projekt och på så sätt tjäna in sin investering.

4.4 Metod 4: Minimally Adequate Synthesizer

Den sista metoden i den här jämförelsen är Minimally Adequate Synthesizer (MAS) utvecklad år 2000 av Erkki Mäkinen och Tarja Systä. Metodens huvudtanke är att generera statecharts utan övergeneraliseringar. Övergeneralisering introducerades i avsnittet om KEK och innebär att oönskad funktionalitet införs när flera scenarier slås samman. MAS undviker problemet genom att fråga designern om råd under processen. [MS00].

Ett viktigt koncept i metoden är *läraren*, en lärare är en person som kan besvara följande två frågor om systemet:

1. Är ett givet beteende hos systemet giltigt?
2. Skall den genererade statecharten förkastas eller accepteras? Om det skall förkastas måste ett motexempel presenteras som visar varför den inte kan accepteras.

MAS använder sekvensdiagram för att generera statecharts och det är det enda indata som metoden behöver. Metoden består av följande tre steg: *skapa ett exekveringsspår*, *generera statecharts* samt *felkontroll*.

Skapa ett exekveringsspår

Det första steget i metoden är att skapa ett exekveringsspår från sekvensdiagrammet genom att undersöka tidslinjen för det objekt som skall modelleras. En post i exekveringspåret består av två intilliggande meddelanden som benämns e_i och e_j. e_i är ett meddelande som objektet har skickat och e_j ett meddelande som har tagits emot. För varje sådant par läggs (e_i, e_j) till exekveringsspåret. Skulle antingen e_i eller e_j saknas ersätts den med NULL, detta uppstår när två meddelanden i följd skickas eller tas emot. När det sista meddelandet i sekvensdiagrammet nåtts skall e_j ersättas med VOID.

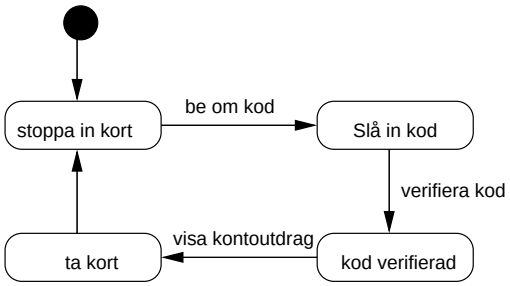
Nedan följer ett exempel på hur ett exekveringsspår genereras från sekvensdiagrammet i figur 4.6.

e_i,	e_j
(stoppa in kort,	be om kod)
(slå in kod,	verifiera kod)
(kod verifierad,	visa kontoutdrag)
(ta kort,	VOID)

Generera statecharts

En post (e_i, e_j), i exekveringsspåret kan tolkas som att vid en tidpunkt i exekveringen skickar objektet (i exemplet ovan terminalen) ett meddelande e_i och reagerar därefter på meddelande i_j som skickats av ett annat objekt. Till exempel, posten (slå in kod, verifiera kod) betyder att det finns ett tillstånd med funktionen `do: slå in kod` och en utgående transition med namnet `verifiera kod`, *do: slå in kod* kan ses som en entryaction för tillståndet.

Vid genereringen av statecharts traverseras exekveringsspåret och i_j blir transitioner och varje unikt i_i blir ett entryaction i ett tillstånd. Om det finns flera likadana i_j i exekveringsspåret kommer endast ett tillstånd att skapas. Nedan visas den resulterande statecharten från exekveringsspåret som utvecklades ovan.



Figur 4.17: Den resulterande statecharten.

Felkontroll

Tidigare nämndes att utvecklaren kallas för lärare och att en lärare måste kunna besvara på två frågor om sitt system. Den första frågan som berörde om en exekveringssekvens var giltig frågas då en transition skall kopplas samman med ett existerande tillstånd. Om algoritmen upptäcker en övergeneralisering frågas läraren om generaliseringen skall vara tillåten, om den inte är tillåten skapas ett nytt tillstånd.

Den andra typen av felkontroll sker efter att hela statecharten har gene-

rerats, då skall läraren ta ställning till om resultatet är godtagbart eller inte. Om läraren accepterar lösningen är algoritmen slutförd, annars måste ett motexempel presenteras som visar varför lösningen inte är korrekt. Ett motexempel är antingen positivt eller negativt. Om läraren presenterar ett positivt motexempel betyder det att algoritmen inte klarar av ett önskat scenario som då måste läggas till. Vid negativa motexempel tillåter systemet scenarier som inte skall få exekveras och läraren måste ange hur lösningen strider mot specifikationen. När motexempel angetts till algoritmen startas den om från början igen, det här förfarandet upprepas tills läraren är nöjd med lösningen.

4.4.1 Utvärdering

MAS ingår i *TED*, ett verktyg som används för att modellera olika UML-diagram. Verktyget är implementerat av Nokia Research och används enligt författarna vid programvaruutveckling hos Nokia. Att metoden används är ett tecken på att metoden fungerar och ger önskvärda resultat. En stor fördel med MAS är som för WS att den fungerar, dessutom har den en väldigt enkel uppbyggnad och det krävs lite arbete för att sätta sig in i den.

Utvecklarna har lagt ner mycket arbete på att varken för mycket eller för lite funktionalitet skall få finnas vid genering av statecharts vilket gör metoden med stor sannolikhet levererar ett korrekt resultat.

En stor nackdel med metoden är att den varken kan hantera hierarkier eller samtidighet. Resultatet blir därför endast en tillståndsmaskin och inte en statechart. En annan nackdel är ingenstans i beskrivningen av metoden nämns det hur hopslagningen av flera exekveringssekvenser skall ske. Metodbeskrivningen ger endast riktlinjer för hur sekvensdiagrammet skall överföras till en tillståndsmaskin men inte hur tillståndsmaskinerna skall slås samman. Troligtvis går det att slå samman flera exekveringssekvenser i steg två, men det är ingenting som beskrivs i algoritmen.

4.4.2 Rekommendation

På grund av att MAS inte innehåller hierarkier går det inte att rekommendera någon att använda metoden. Om metoden kombinerades med en av

de andre metodernas algoritmer för att införa hierarkier skulle MAS bli en intressant metod.

En annan sak som måste läggas till för att metoden skall bli användbar är hur olika statecharts sammanfogas. Troligtvis är det enkelt att göra i steg två men det måste finnas en beskrivning på hur det skall ske.

4.5 Sammanfattning

Ovan har fyra metoder för att utveckla statecharts beskrivits och utvärderats, samtliga metoder har sina för- och nackdelar och de passar olika bra för olika projekt.

Som en sammanfattning på avsnittet presenteras här en tabell indelad efter ett antal olika variabler där metoderna skiljer sig åt.

	KEK	**VS**	**WS**	**MAS**
automatik	delvis	nej	ja	nej
Hierarkier	nej	ja	ja	nej
över-generalisering	ja	nej	nej	ja
fördelar	innehåller utveckling av krav-specifikation	enkel att använda	felkontroller	används redan
nackdelar	inga hierarkier	lite hjälp	komplex	inga hierarkier
projekttyp	inga	små projekt	stora projekt	inga

Kapitel 5

Implementering

I det här kapitlet redovisas den praktiska delen av examensarbetet. Kapitlet inleds med en beskrivning av uppgiften och vilket dess syfte var, därefter redogörs för hur statecharten har utvecklats och motiven bakom valet av utvecklingsmetod.

Redogörelsen för programmeringsarbetet är inriktad på de viktigaste delarna av implementationen som till exempel hur mätvärdesberäkningar har tagits fram. Kapitlet avslutas med en beskrivning av modellens funktion och hur resultatet av arbetet används.

5.1 Uppgift

Uppgiften var att implementera en mjukvarumodell av det fysiska system som har hand om kraftförsörjningen till Ericssons WCDMA-basstationer. Modellen skulle fungera på samma sätt som det fysiska systemet och skicka rimliga mätvärden och signaler till styrprogramvaran. Programvarans beteende skulle även kunna förändras för att möjliggöra test som simulerade att hårdvaran inte fungerade korrekt.

Syftet var att ersätta test av styrprogramvaran mot hårdvara med test mot mjukvarumodellen. Det här skulle medföra en betydligt kortare testfas vilket i sin tur leder till en billigare slutprodukt.

5.1.1 Översikt över modellens funktion

Det fysiska systemet består av tre till sex stycken växelströmsenheter (även kallade *PSUer*) samt eventuellt även ett batteri som används som backup-system. PSUerna och batteriet försörjer radiobasstationen med ström och spänning. Systemets beteende styrs av en mjukvara som kallas *LrsPc* vars uppgift är att:

1. Övervaka kraftförsörjningen med avseende på gränsvärden för ström, spänning samt temperatur.

2. Ladda batterierna och motionera dem regelbundet.

3. Reglera PSUerna så att de ger optimal spänning beroende på temperatur, batteriets status, lastdelning med mera.

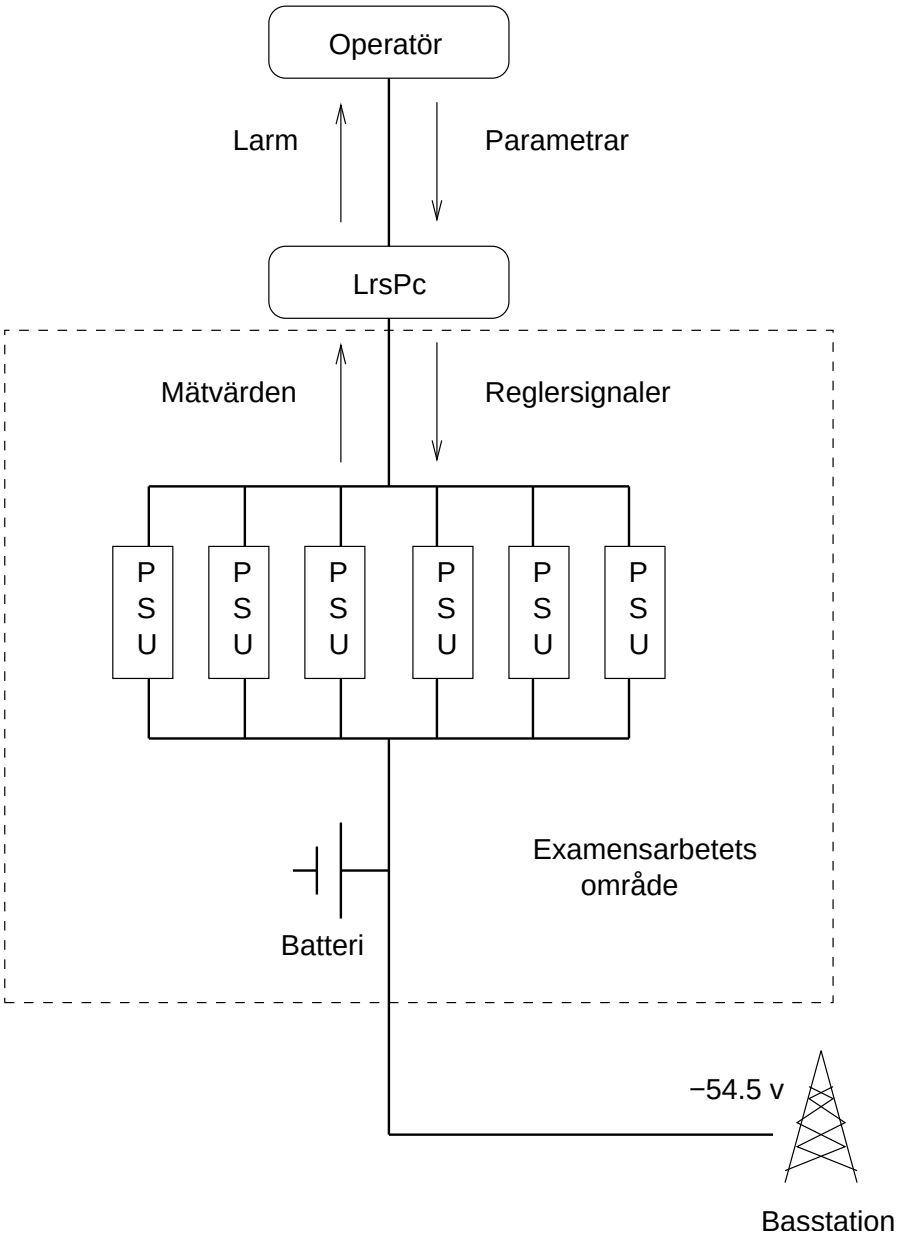
4. Hantera strömavbrott.

LrsPc får parametrar från teleoperatören som styr hur strömförsörjningen till basstationen skall fungera. Operatören kan till exempel initiera tester av strömförsörjningen och sätta igång funktioner som konditionerar batteriet.

LrsPc skickar i sin tur reglersignaler till PSUerna och batteriet, signalerna styr respektive enhets beteende. När systemet fungerar korrekt skall reglersignalerna vara sådana att den totala spänningen från enheterna och batteriet ger en spänning på -54,5 volt till radiobasstationen samt en ström som är tillräcklig för att driva den.

PSUerna och batteriet skickar regelbundet (var tionde sekund) mätvärden till LrsPc som med hjälp av dem tar beslut om hur reglersignalerna skall förändras för att systemets kapacitet skall användas på bästa sätt. Varje PSU skickar information om sin spänning, ström samt hur många procent av dess kapacitet som används. Batteriet skickar information om sin spänning, ström och temperatur. Om LrsPc upptäcker större problem som till exempel ett strömavbrott skickas larm till operatören om detta.

I figur 5.1 visas en översikt av systemet, det är området inom den streckade fyrkanten som skall ersättas med en mjukvarumodell.



Figur 5.1: Översikt av systemet, det är området inom den streckade fyrkanten som skall ersättas med en mjukvarumodell.

5.1.2 Scenarior

Modellen måste klara fem olika *scenarier*, ett scenario beskriver en situation som kan uppstå hos det fysiska systemet. De scenarier som modellen skulle klara av var *normal användning*, *strömavbrott*, *batteritest*, *återuppladdning av batteri* samt *DC-kabel utdragen*.

Scenarierna beskriver de vanligaste situationer som kan uppstå hos en basstationen och det är LrsPc:s uppgift att kunna hantera dem. Scenarierna startas i de flesta fall genom att testaren skickar en signal till mjukvarumodellen som då förändrar sitt beteende så att den imiterar den verkliga hårdvarans beteende vid en sådan situation.

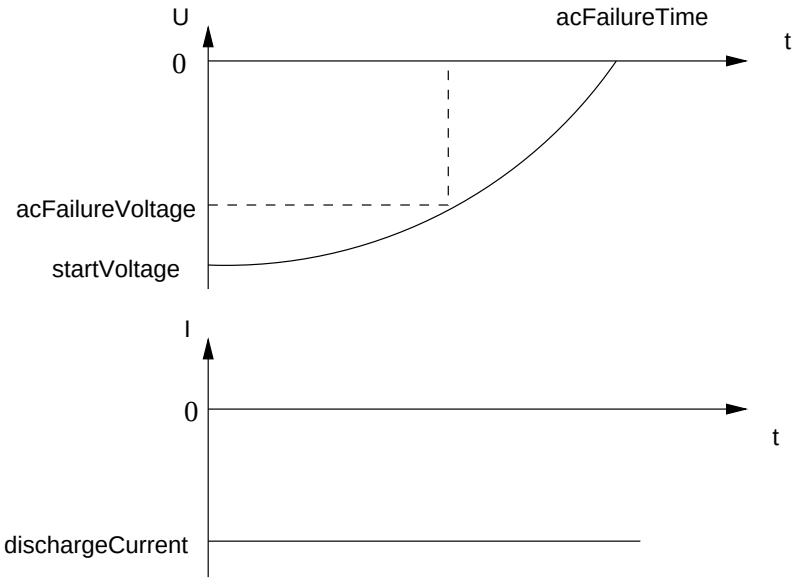
Scenario 1: Normalt beteende
Det första scenariot simulerar hur systemet fungerar normalt då systemet inte utsätts för några störningar.

Modellen skall skicka mätvärden var tionde sekund (det här gäller för samtliga scenarier), mätvärdena beräknas enligt följande riktlinjer vid normalt beteende:

- PSUernas spänning beror på hur mycket spänning som LrsPc begär att PSUn skall leverera, samt på PSUns konfiguration.
- PSUernas ström beror på dess spänning samt hur mycket ström som basstationen kräver.
- PSUernas last är proportionell mot dess spänning och ström.
- Batteriets spänning beror på PSUernas spänning
- Batteriets ström skall vara i det närmaste noll.
- Batteriets temperatur varierar långsamt mellan värden omkring 20 grader.

Scenario 2: Strömavbrott
En av de viktigaste uppgifterna för kraftförsörjningssystemet är att hantera strömavbrott och det här scenariot simulerar att basstationens strömförsörjning upphör vilket innebär att den måste gå över till batteridrift.

Under simuleringen kommer batteriets spänning långsamt att sjunka samtidigt som dess ström är konstant enligt figur 5.2:



Figur 5.2: Batteriets spänning och ström under strömavbrott.

Samtliga värden för PSUerna kommer att vara noll.

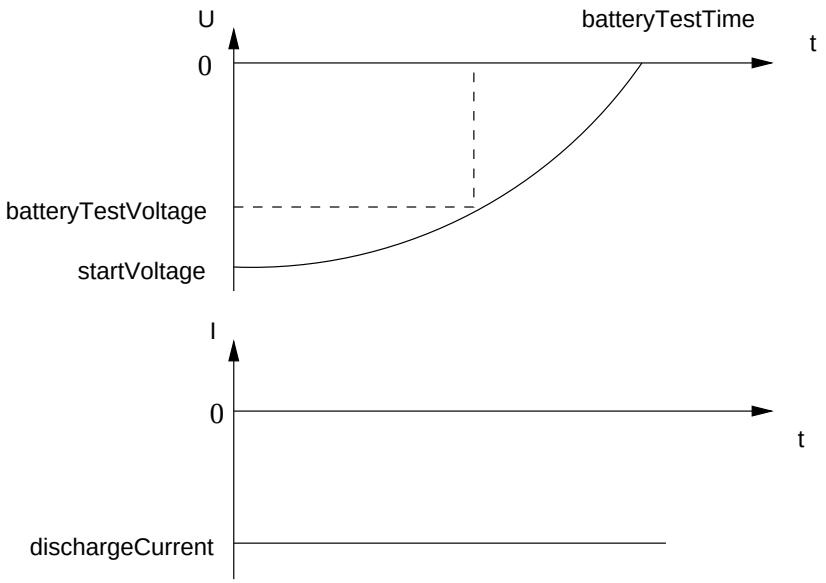
Scenario 3: Batteritest
Batterier åldras och dess kondition försämras över tiden. För att operatören inte skall överraskas av att batterierna är dåliga när ett strömavbrott inträffar kan ett batteritest göras. Genom att undersöka hur snabbt batterierna laddas ur kan operatören avgöra om det behöver bytas ut eller inte.

Under batteritestet sänker LrsPc spänningen på PSUerna till -44.0 volt, en spänning som är tillräcklig för att driva basstationen om batterierna inte skulle fungera. Batteritestet fungerar på samma sätt som strömavbrottet med skillnaden att PSUerna levererar en låg ström och spänning under hela testet enligt figur 5.3.

PSUernas spänning är -44.0 volt under hela testet, övriga parametrar beräknas på samma sätt som i `normal`.

Scenario 4: Återuppladdning av batteriet
Efter att batteriet har laddats ur vid strömavbrott eller batteritest måste det laddas upp igen. Återuppladdningen av batteriet sker i tre faser:

1. floatcharging – I den första fasen ökar batteriets spänning långsamt tills den når den spänning som LrsPc begär.



Figur 5.3: Batteriets spänning och ström under batteritest.

2. tre timmars väntan – När floatcharge har avslutats kommer systemet att vänta i tre timmar.

3. boostcharging – När tre timmar har passerat startar boostcharging. Boostcharging ser till att batterierna är helt fulladdade, den här fasen pågår tills batteriet inte längre kan ta emot mer ström.

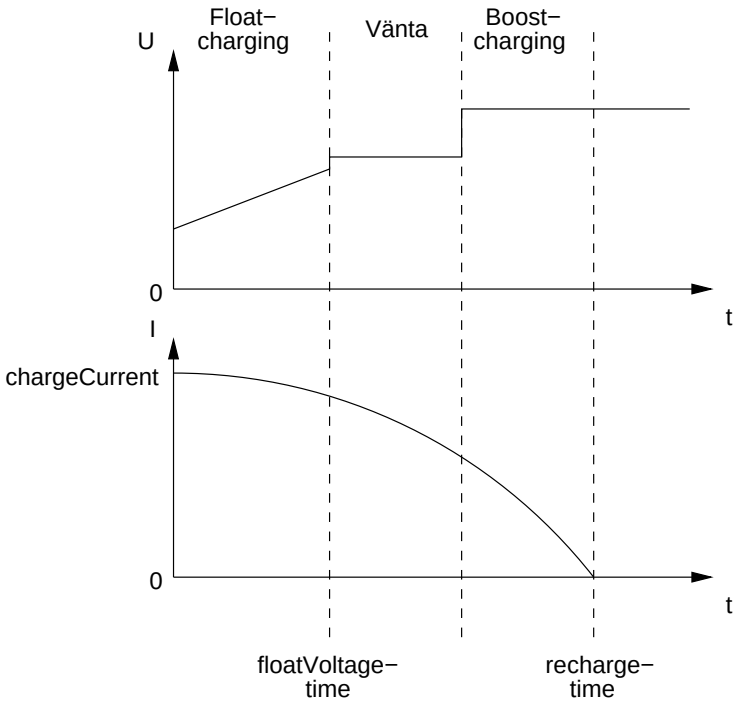
Batteriets ström och spänning kommer att följa den kurva som visas i figur 5.4, strömmen är positiv under uppladdningen av batteriet.

PSUernas mätvärden beräknas på samma sätt som i scenariot *normalt beteende*.

Scenario 5: Strömkabel utdragen

Det här scenariot innebär att en PSUs kabel har blivit utdragen. Detta får dess ström att sjunka till en mycket låg nivå samtidigt som spänningen sjunker långsamt. Övriga PSUers ström kommer att öka för att kompensera strömbortfallet.

Det här scenariot inträffar parallellt med något av de tidigare beskrivna vilket innebär att PSUernas mätvärden beräknas utifrån vilket scenario som sker samtidigt.



Figur 5.4: Batteriets spänning och ström under uppladdning.

5.2 Utveckling av statechart

Utifrån dessa scenarier skulle en statechart som beskrev modellen utvecklas. Utvecklingsmiljöns tillsammans med projektets egenskaper och krav ledde till ett val av metod för att utveckla statechart, detta beskrivs nedan.

5.2.1 Urval

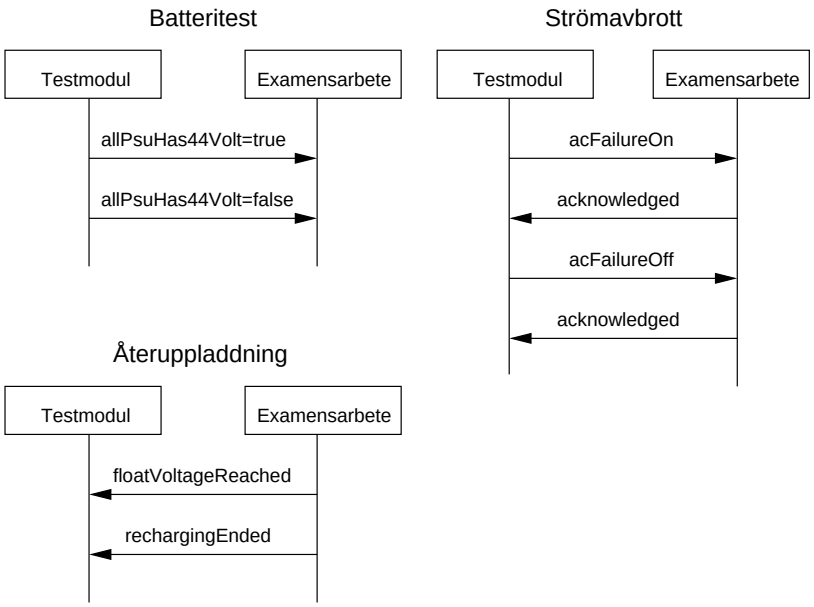
De metoder som undersöktes i föregående kapitel har alla sina för- och nackdelar och de passar olika bra för olika typer av projekt. Examensarbetets egenskaper gav att metoden skulle ha följande egenskaper:

- den skulle vara anpassad för små projekt.
- den skulle klara av att modellera statecharten manuellt.
- den skulle kunna hantera hierarkier.
- den skulle klara av att sammanfoga flera scenarier.

Eftersom hierarkier kan behövas gick varken KEK eller MAS att använda, MAS har dessutom ingen möjlighet att sammanfoga flera scenarier vilket var ytterligare en anledning till att den valdes bort. Kvar var VT och WS där WS är en metod som är utformad för automatisering och stora projekt medan VT är enkel att använda manuellt och passar bra för små projekt. Tack vare sin enkelhet föll valet på VT, om utvecklarna bakom WS hade tillhandahållit sitt utvecklingsverktyg hade troligtvis den metoden använts. Nackdelen med VT är att den endast ger en *vägledning* vid utvecklingsarbetet vilket gör att det mesta av arbetet hamnar hos designern som får väldigt lite hjälp från metoden.

Genomförande

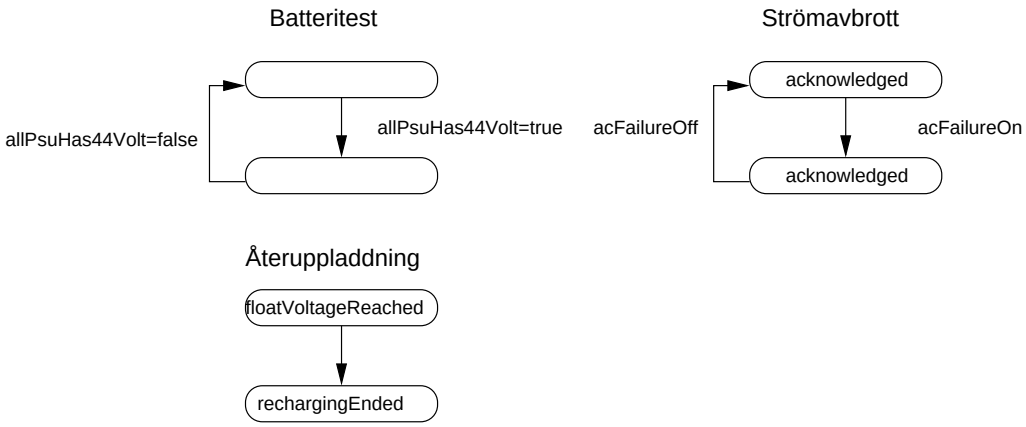
Av de fem scenarier som skulle utvecklas var det endast batteritest, strömavbrott samt återuppladdning av batterier som behövde sekvensdiagram, i övriga fall var de för enkla för att det skulle vara meningsfullt att utveckla dem. Sekvensdiagrammen visas i figur 5.5.



Figur 5.5: Examensarbetets sekvensdiagram.

Därefter utvecklades tillståndsmaskiner för varje sekvensdiagram enligt VT:s regler, tillståndsmaskinerna visas i figur 5.6.

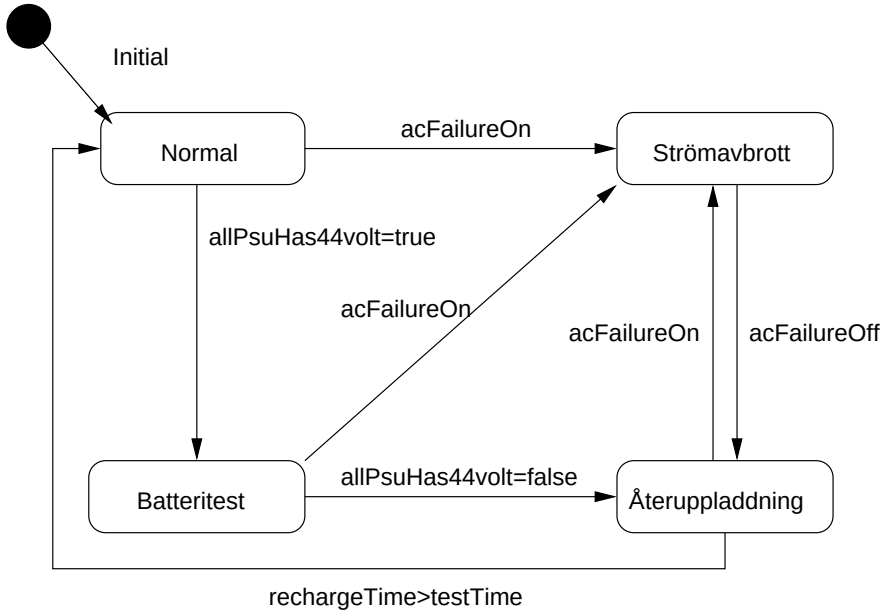
Nästa steg var att identifiera hur scenarierna förhåller sig till varandra. Det visade sig att normalt beteende, strömavbrott, batteritest och återuppladdning av batteritest hade OR-förhållande, alltså endast en av dem fick ex-



Figur 5.6: Examensarbetets tillståndsmaskiner.

ekveras åt gången medan DC-kabel utdragen hade ett AND-förhållande till övriga scenarier. VT ger mycket frihet till designern när tillståndsmaskinerna skall slås samman vilket innebär att den resulterande statecharten inte är *den enda* statechart som kunnat modelleras utifrån indatatat, en annan designer skulle möjligtvis ha kommit fram till en annan lösning.

Resultatet blev statecharten i figur 5.7



Figur 5.7: Översikt av modellens tillstånd och transitioner.

Noterbart är att acknowledged-tillstånden har ändrat namn till ett namn som bättre beskriver tillståndens uppgift. Dessutom har hela scenariot DC-kabel utdragen införlivats i de övriga tillstånden. Detta val gjordes på grund av att det underlättade programmeringsarbetet väsentligt.

5.3 Implementering av modellen

Efter att statecharten för modellen tagits fram gjordes implementeringen av den. Programmeringen gjordes i utvecklingsmiljön Rose RT som är en miljö som är speciellt utvecklad för att hantera realtidsapplikationer. Miljön är uppbyggd kring statecharts och mycket av arbetet består av att designa statecharts i en grafisk editor. Programmeringen sker i C++ och vid kompileringen görs samtliga statecharts om till objekt och metoder, ett tillstånd blir ett objekt och en signal blir en metod.

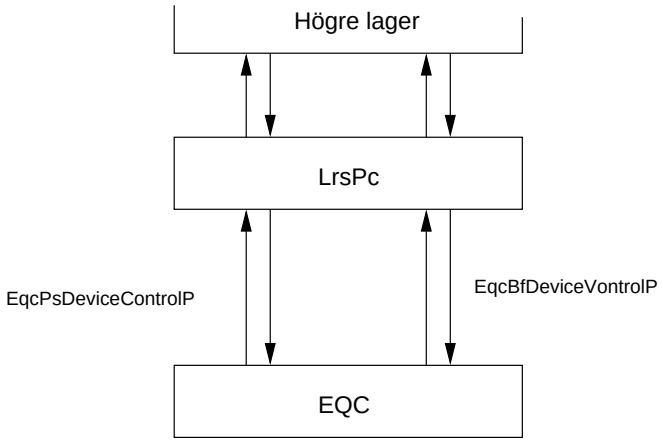
5.3.1 Integrering i befintlig modell

I ett så stort system som en radiobasstationen skickas det en otroligt stor mängd signaler hela tiden, för att inte alla tillstånd i ett system skall nås av samtliga signaler innehåller Rose RT något som de kallar för *protokoll*. Varje protokoll innehåller ett antal in- och utsignaler med en gemensam nämnare, ett protokoll kan till exempel innehålla de signaler som hanterar uppsättning av en ny telefonförbindelse eller de signaler som initialiserar basstationen. För att ett statechart skall nås av signaler måste det först kopplas samman med ett eller flera protokoll.

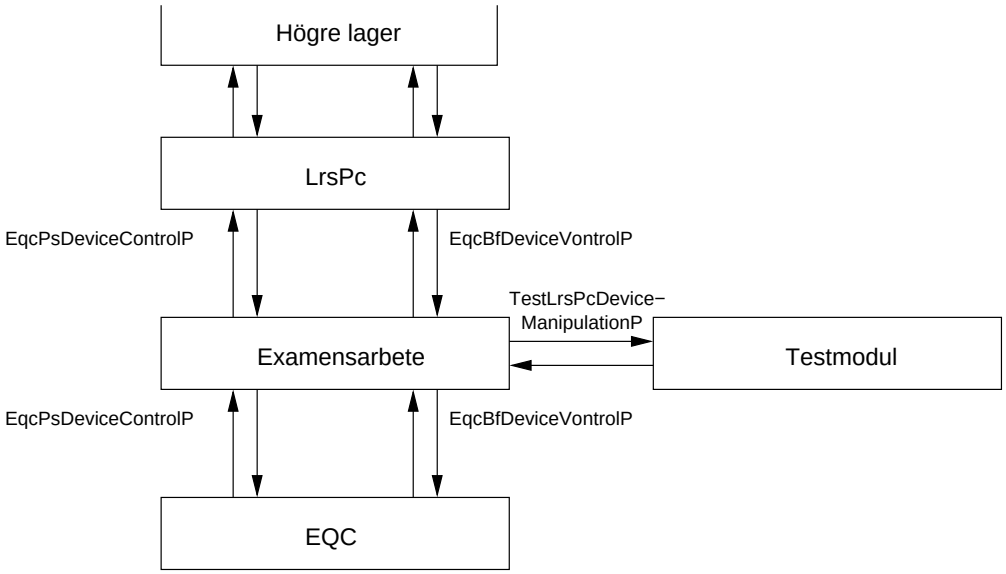
I det riktiga systemet ser LrsPCs kommunikation ut som i figur 5.8.

EQC står för EQuipment Control och är ett subsystem som kontrollerar hårdvaran (bland annat PSUer och batteri). Modellen som skulle implementeras lades in mellan de två komponenterna och implementerade protokollen åt båda hållen som i figur 5.8.

EqcBfDeviceControlP och EqcPsDeviceControlP är protokoll som har hand om kommunikationen mellan batteriet och LrsPc respektive mellan PSUerna och LrsPc. Eftersom systemet kan ha upp till sex stycken PSUer implementerades sex instanser av detta protokoll.



Figur 5.8: Kommunikationen för LrsPc.



Figur 5.9: Kommunikationen för LrsPc och testmodellen.

Genom att implementera protokollen var det möjligt för modellen att efterlikna beteendet hos hårdvaran utan att LrsPc märkte någon skillnad. Modellen lades in som ett mellanlager mellan EQC och LrsPC och avgjorde vilka signaler som skulle få passera igenom. Men eftersom modellens beteende även skulle kunna förändras så att det avvek från normalt beteende infördes protokollet TestLrsPcDeviceManipulationP. TestLrsPcDeviceManipulationP innehåller signaler för att ställa in parametrar som styr modellens beteende samt för att starta scenarier. Protokollet är ihopkopplat med en modul som används vid testning av systemet. Detta kommer att beskrivas närmare i avsnitt 5.3.5.

5.3.2 Utveckling av mätvärden

När modellen var infogad i systemets befintliga struktur startade undersökningen kring de mätvärden hårdvaran skickade. Utifrån loggar från testkörningar gjordes en analys av mätvärden, analysen användes som underlag för att utveckla algoritmer som gav mätvärden liknade de som det fysiska systemet skickar.

Från början var förhoppningen att med hjälp av teoretiska samband kunna bestämma mätvärden, tyvärr visade det sig att det i endast ett fall fanns ett sådant samband. I övriga fall gjordes analyser av mätvärden för att härleda mätvärdenas samband.

Utgångspunkten för beräkningarna var hur mycket ström som basstationen behövde och vilken spänning som LrsPc ville att varje PSU skulle leverera, dessa två värden var alltid givna och utifrån dem skulle övriga värden härledas. Basstationens ström ställde testaren in med hjälp av testprotokollet medan PSUernas begärda spänning gavs i en reglersignal från LrsPc.

De sex mätvärden som skulle beräknas var som tidigare nämnts: PSUernas ström, spänning samt last och batteriets ström, spänning och temperatur. Utifrån specifikationen för examensarbetet och analys av mätvärden från testkörningar av hårdvaran togs följande beräkningar fram:

Mätvärde 1: PSUernas spänning
Spänningen beräknas utifrån det värde som LrsPc begär. Testaren kan med en signal ställa in hur mycket spänningen skall skilja från det begärda värdet.

Mätvärde 2: PSUernas ström

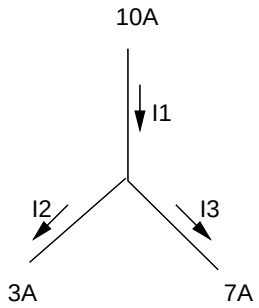
I alla tillstånd förutom strömavbrott beräknas PSUernas ström enligt den algoritm som presenteras nedan. Algoritmen är uppdelad i två delar där den ena delen tar hänsyn den spänning som PSUn har och den andra delen till inställningar som testaren kan göra.

Algoritmen går ut på att fördela den totala strömmen på respektive PSU, PSUernas totala ström ges av hur mycket ström som basstationen behöver samt hur mycket ström som batteriet ger eller tar.

Den totala ström som PSUerna skall levereras beräknas med följande formel:

$I_{PSU} = I_{RBS} + I_{batteri}$ Där RBS står för radiobasstation.

Formeln är härledd från Kirchhoffs strömlag som säger att summan av strömmen som går till en punkt måste vara lika med summan av strömmen som lämnar punkten vilket illustreras i följande bild:



Figur 5.10: Exempel på Kirchhoffs strömlag.

Steg 1: Spänningsdifferens

I det första steget av algoritmen beräknas PSUernas ström med avseende på den spänning de har. En PSU med hög spänning kommer att få hög ström, sambandet är dock inte linjärt utan en liten ökning i spänning ger en stor ökning i ström. Algoritmen utgår från den totala ström som skall levereras och fördelar strömmen mellan PSUerna.

Algoritmen tar även hänsyn till hur stor differens det är mellan PSUernas spänning; en uppställning med stor spänningsspridning skall ha stor spridning på strömmen. Spridningen beräknas utifrån en *idealandel*, om det är sex PSUer i uppställningen är idealandelen $\frac{1}{6}$. Spridningsgraden tas fram genom att beräkna den största differensen mellan idealandelen och varje PSUs spänningsandel.

$spread_{max} = max(abs(\frac{1}{n} - \frac{U_i}{\sum_{j=0}^{n}(U_j)}))$. Där n är antal PSUer.

Nästa del av algoritmen tar hänsyn till hur stor spridningen är för respektive PSU. Spridningen beräknas enligt:

$$spread_i = \frac{1}{n} - \frac{U_i}{\sum_{j=0}^{n}(U_j)}$$

Det värde som tagits fram är ett mått på hur stor andel av den totala strömmen som PSUn skall ge, för att göra skillnaden större multipliceras det med ett stort tal, initialt är det 800 000[1]. Det här värdet multipliceras därefter med $spread_{max}$ för att ta hänsyn till uppställningens spridning.

Resultatet är ett mått på hur långt från idealandelen som strömmen för den här PSUn skall vara.

Sammantaget beräknas strömmen för en PSU enligt:

$$I1_i = I_{tot} * (\frac{1}{n} - 800000 * spread_{max} * spread_i)$$

Om det skulle visa sig att 800 000 är ett för stort värde (i vissa fall kan det leda till att strömmen får fel tecken eller att lasten för en PSU överstiger 100 procent) kommer den att gradvis sänkas med 20 procent tills samtliga PSUers ström- och lastvärden är giltiga.

Steg 2: Individuell inställning
Alla PSUer har olika egenskaper eftersom de slits olika mycket, för att kunna simulera detta kan testaren ställa in individuella egenskaper för varje PSU. Det här ger även testaren möjlighet att förändra respektive PSUs strömvärde. Den här delen av algoritmen påverkas inte av att spänningen förändras utan är oberoende av dess variation.

Den individuella inställningen görs med en signal som ställer in *förhållandet* mellan PSUernas ström. Om man har en uppställning med tre PSUer som skall leverera 60 Ampere och har inställningarna 5 för PSU_0, 10 för PSU_1 och 15 för PSU_2 kommer PSU_1 att ge dubbelt så mycket ström som PSU_0 och PSU_2 tre gånger så mycket ström som PSU_0. Strömmen för respektive PSU kommer att vare $I_0 = 10, I_1 = 20, I_2 = 30$. Allmänt kan det här uttryckas enligt följande formel:

[1]Detta värde togs fram genom att undersöka testkörningar med både stor och liten spridning och det visade sig att 800 000 gav det bästa resultatet

$$I2_i = RBScurrent * \frac{installning_i}{\sum_{j=0}^{n}(installning_j)}$$

Steg 3: Medelvärdesberäkning
Det sista steget i beräkningen består av att ta medelvärdet mellan de två beräkningarna, resultatet är PSUns ström.

$$I_i = \frac{I1_i + I2_i}{2}$$

Den här algoritmen ger testaren möjlighet att antingen låta modellen efterlikna den riktiga hårdvaran eller låta beteendet avvika från normalt beteende.

Mätvärde 3: PSUernas last
Efter analys av mätdata visade det sig att lasten alltid är direkt proportionellt mot PSUns ström och spänning enligt följande formel:

$$P_i = \frac{U_i * I_i}{-12}$$

Det här sambandet visar att mätvärdet är överflödigt, om LrsPc känner till ström och spänning går det alltid att beräkna lasten.

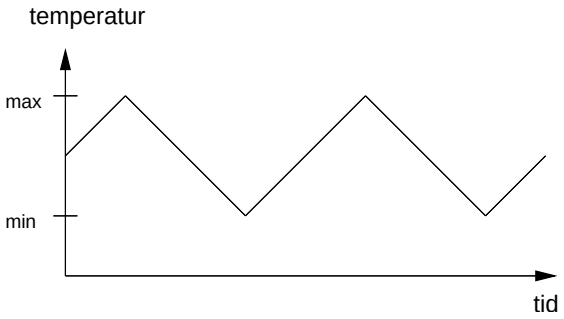
Mätvärde 4: Batteriets spänning
Batteriets spänning är alltid något högre än PSUernas, hur mycket högre kan testaren bestämma med hjälp av en signal i testprotokollet.

Mätvärde 5: Batteriets ström
I specifikationen för Ericssons basstationer står det att batteriet skall ha en negativ ström när den laddas ur och en positiv ström när den laddas upp. Hur stor strömmen är beror på vilket scenario som pågår för tillfället, strömmen kommer att följa de kurvor som presenterades under "Scenarier". I normalfallet kommer batteriet att ha en liten negativ ström som indikerar att batteriet mycket långsamt laddas ur.

Mätvärde 6: Batteriets temperatur
Batteriets temperatur är relativt konstant och det var inget krav från TietoEnator att temperaturen skulle kunna variera hos testmodellen. Men eftersom en del i LrsPc algoritm tar hänsyn till batteriets temperatur var det en fördel om även detta kunde testas.

Lösningen blev en funktion där testaren kan ställa in inom vilket intervall som temperaturen skall variera och hur lång tid det skall ta för temperaturen att gå mellan extremvärdena. Temperaturen kommer att variera mellan

intervallets gränser och förändringstakten är konstant. Detta ger att temperaturkurvan kommer att ha följande utseende.



Figur 5.11: Temperaturvariationen.

5.3.3 Scenarior

När analysen av mätvärdena var klar implementerades scenarierna och de signaler som behövdes för att styra modellens beteende. I det här avsnittet beskrivs först hur scenariorna implementerades och därefter vilka funktioner som implementerades i testprotokollet.

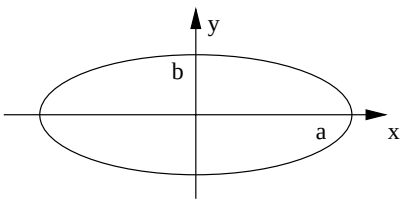
Normal

Det här scenariot implementerades efter de värden som beskrivits ovan.

Strömavbrott

Det största problemet vid implementeringen av strömavbrott var hur en mjuk nedgång i batteriets spänning skulle åstadkommas. Till en början laddas batteriet ur mycket långsamt för att senare öka urladdningshastigheten allt mer. För att åstadkomma detta användes ekvationen för en ellips:

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1$$



Figur 5.12: Ellips.

Med hjälp av den här formeln var det möjligt att åstadkomma precis det som

eftersöktes, en funktion där värdet sjönk långsamt initialt för att därefter sjunka allt fortare.

Batteritest

Batteritestet fungerar ungefär på samma sätt som strömavbrott och dess funktion ser i stort sätt likadan ut. Den enda svårigheten var att simulera när det blev strömavbrott samtidigt som ett batteritest utfördes. Det här löstes genom att fortsätta ladda ur batteriet på samma kurva som vid batteritestet.

Återuppladdning

Återuppladdningen är implementerad så att den fungerar precis som beskrivet i specifikationen. Testaren kan specificera hur länge systemet skall vara i respektive fas och hur snabbt batteriets ström skulle laddas upp.

DC-kabel utdragen

LrsPc upptäcker att en kabel är utdragen om PSUn levererar mellan noll och tre procent av sin möjliga last. I vanliga fall beräknas PSU-lasten med:

$$P_i = \frac{U_i * I_i}{-12}$$

Omskriven med avseende på ström:

$$I_i = \frac{P_i * -12}{U_i}$$

Med lasten satt till 1.5 procent ges följande formel:

$$I_i = \frac{1.5 * -12}{U_i} = \frac{-18}{U_i}$$

Detta kommer att indikera en mycket liten spänning och LrsPc kommer att anta att den aktuella PSUns kabel är utdragen och utesluta den från sina beräkningar.

5.3.4 Signaler

De signaler som styr beteendet hos modellen finns beskrivna i protokollet `TestLrsPcDeviceManipulationP`. Det är ett protokoll som endast används för testning av systemet och ingår inte i den kod som finns i basstationerna.

För att göra modellen så generell som möjligt har ett stort antal signaler utvecklats som låter testaren förändra modellens beteende. Här kommer signalerna att beskrivas utifrån sin funktion, varje funktion beskrivs och

presenteras tillsammans med sina associerade signaler.

Hårdvaran svarar inte

För att kunna testa hur LrsPc beter sig om hårdvaran slutar fungerar har två signaler utvecklats som startar och avslutar simulering av att hårdvaran inte fungerar. Det görs genom att modellen slutar svara på signaler från LrsPc.

`deviceOK`
`deviceNotOK`

Skicka vidare signal till hårdvarulagret

I vanliga fall skickas inte signalen med begärd spänning för PSUerna vidare till hårdvarulagret (EQC) utan den stannar vid testmodellen. Om testaren vill att signalen skall skickas vidare kan den här signalen användas.

`configVoltOffset`

Starta mätningar

Testaren kan även ställa in så att antingen PSUerna eller batteriet slutar att skicka mätvärden. Om det inte kommer några mätvärden under en tid skall LrsPc först försöka interpolera med gamla värden, om det dröjer en längre tid skall enheterna anses vara trasiga.

`startPsMeasurementSeq`
`startBfMeasurementSeq`
`stopPsMeasurementSeq`
`stopBfMeasurementSeq`

Antal enheter i simuleringen

Med hjälp av den här signalen kan testaren ställa in hur många PSUer det skall ingå i simuleringen samt om det skall finnas något batteri.

`possAllocPreset`

Ström hos basstationen

Som nämnts tidigare är ett av utgångsvärdena vid beräkning av mätvärden vilken ström som basstationen behöver. Med den här signalen ställer testaren in hur stor den strömmen skall vara.

`setRBSCurrent`

Spänning hos batteriet

Den här signalen används för att ange hur mycket som batteriets spänning skall skilja från den PSU som har högst spänning.

`setBfVoltOffset`

Spänning hos PSUer

PSUerna levererar aldrig exakt den spänning som LrsPc begär, med hjälp av den här inställningen kan testaren ställa in hur mycket varje PSU skall skilja från det begärda värdet individuellt.

`setPsVoltOffset`

Ström hos PSUer

En individuell inställning för hur mycket ström respektive PSU skall ge. För en närmare beskrivning av strömberäkningen se avsnitt 5.3.2.

`setPsCurrentOffset`

Batteriets ström i normalt tillstånd

Även när batteriet inte används kommer det att dra lite ström, hur mycket ställs in med den här signalen.

`setBfCurrentInNormal`

Batteriets temperatur

Temperaturen kommer att variera mellan två värden med konstant steglängd. Starttemperaturen ligger mittemellan gränsvärdena, temperaturen kommer att börja gå mot det högre av värdena.

`setBfTempIntervall`

Strömavbrott - start

Vid start av strömavbrott kan testaren ställa in fyra parametrar, den första är hur stor spänning batteriet skall ha vid avbrottets start och den andra vilken ström som batteriet skall ge under urladdningen. De två sista parametrarna bestämmer vilken spänning batteriet skall ha vid en viss tidpunkt. Det ger en punkt på den urladdningskurva som beskrevs i scenariot för strömavbrott. Med hjälp av den punkten går det att beräkna hur snabbt urladdningen skall ske och när batteriets spänning skall nå noll.

`acFailureOn`

Strömavbrott - stopp

När strömavbrottssimuleringen avbryts startas uppladdningen av batteriet automatiskt. Samtidigt som testaren avbryter urladdningen bestämmer han parametrarna för återuppladdningen. Testaren kan med den här inställningen bestämma vilken ström batteriet skall ha vid start av återuppladdningen, hur länge återuppladdningen skall vara i fasen floatcharging och hur lång tid det skall ta innan batteriet skall vara fulladdat.

```
acFailureOff
```

Batteritest

Batteritestet startas inte med en signal utan genom att LrsPc begär att alla PSUers spänning skall vara -44.0 volt, testet avslutas när någon av PSUerna får en spänning som inte är -44.0 volt. Detta gör att parametrar för både urladdning av batteriet och återuppladdning måste ställas in före batteritestets start. Med hjälp av den här signalen ställs parametrarna för ur- och uppladdning in, parametrarna är samma som för strömavbrottssimuleringen.

```
batteryTest
```

DC-kabel i- eller utdragen

Det här är signalen som har hand om scenariot för DC-kabel urkopplad, signalen har ett argument som indikerar vilken PSU som avses.

```
dcCableConnected
dcCableDisconnected
```

Feedback från återuppladdningen

För att testaren skall kunna se hur lång tid det har tagit tills en fas i återuppladdningen är klar skickar modellen signaler till testmodulen när floatcharging är färdig och när batteriet är fulladdat.

```
floatVoltageReached
rechargingEnded
```

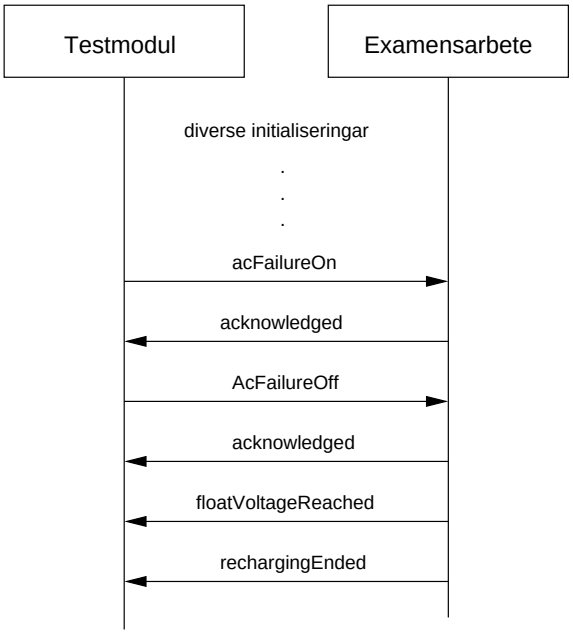
Allmän feedback

För att testmodulen skall veta att examensarbetet har tagit emot en signal skickas en bekräftelse vid varje mottagen signal.

```
acknowledged
```

5.3.5 Testning

Vid testning använder Rose RT en testmodul som genererar de signaler som skickas till den modul som skall testas. Som nämnts tidigare är examensjobbet ihopkopplat med testmodulen genom protokollet TestLrsPcDeviceManipulationP. En testsekvens för att testa strömavbrott och återuppladdning kan se ut som i figur 5.13.



Figur 5.13: Exempel på testsekvens.

Testmodulen skickar inte sin nästa signal förrän den har fått de signaler som den förväntar sig. I fallet ovan skickas inte `acFailureOff` förrän `acknowledged` har tagits emot från examensarbetet.

Testningen av modellen visade att examensarbetet klarade av att bete sig som det fysiska systemet, vid jämförelser med tester av det riktiga systemet verifierades att de mätvärden som modellen levererade var lika det riktiga systemets mätvärden.

5.3.6 Framtida användning

Även om modellen klarar av att imitera det riktiga systemet så är det inte

dess främsta styrka, styrkan ligger i att testaren kan få modellen att *avvika* från normalt beteende. Modellen har många olika konfigureringsmöjligheter i och med testprotokollet som utvecklats, testaren kan ställa in de parametrar som systemet skall utgå från och därefter observera beteendet hos LrsPc. Det här ger testaren möjlighet att testa funktionen hos LrsPc både när hårdvaran fungerar korrekt och när den har problem.

Det kan verka som att examensjobbet är för sent ute eftersom LrsPc redan är implementerat. Utvecklingen av LrsPc är just nu inne på sin tredje generation i utvecklingsprocessen och man förutspår att ytterligare två generationer kommer att utvecklas. Modellen hade självklart gjort mest nytta om den funnits redan vid den första generationen men den kommer åtminstone att användas vid de avslutande två cyklerna. Dessutom kommer modellen att användas vid det test som görs när en större förändring har skett i någon del av basstationens programvara. För att undvika att förändringar som härrör från en annan del av systemet påverkar övriga funktioner görs ett test som verifierar samtliga funktioners korrekthet. I det här testet kommer examensarbetet att ingå för att verifiera att LrsPc fungerar som det skall.

Kapitel 6

Resultat

I resultatkapitlet visas de nya kunskaper som undersökningen har lett till, här besvaras även den problemformulering och det syfte som presenterades i inledningskapitlet.

Kapitlet och rapporten avslutas med en diskussion om hur programmering med statecharts kan komma att utvecklas framöver.

6.1 Utveckling av statecharts

Problemformuleringen för det här uppdraget löd:

> *Vilka olika metoder finns det för att ta fram statecharts och vilka för- och nackdelar har respektive metod?*

Även om den här rapporten inte har tagit upp alla metoder som finns tillgängliga så har den tagit upp olika typer av metoder och redogjort för dem som har blivit mest uppmärksammade inom området. Det skulle naturligtvis varit en starkare rapport om fler metoder undersökts men det har visat sig att övriga för författaren kända metoder inte tillförde någon ny kunskap, de var samtliga varianter av de metoder som tagits upp i den här rapporten.

När det gäller metodernas för- och nackdelar har de presenterats i avsnitt

4.5 i kapitel ”Metoder för att utveckla statecharts”. Kapitlet avslutades med en tabell som visar samtliga metoders för- och nackdelar och vilka projekt som de passar för.

Tack vare detta kan undersökningens problem anses vara besvarat, under undersökningens gång har även andra reflektioner uppkommit vilka presenteras nedan.

6.1.1 Metodernas grundstruktur

Som läsaren kanske upptäckte under redogörelsen för metoderna har de en liknande grundstruktur, även om de har många olikheter gällande indata, formalisering och annat så har samtliga följande struktur:

1. Skapa ett antal tillståndsmaskiner från någon UML-notation, till exempel sekvensdiagram eller collaborationsdiagram.
2. Slå samman tillståndsmaskinerna.
3. Inför hierarkier (om metoden innehåller det).

Förutom dessa punkter har de flesta metoder felkontroller antingen kontinuerligt eller i slutet när en statechart har utvecklats.

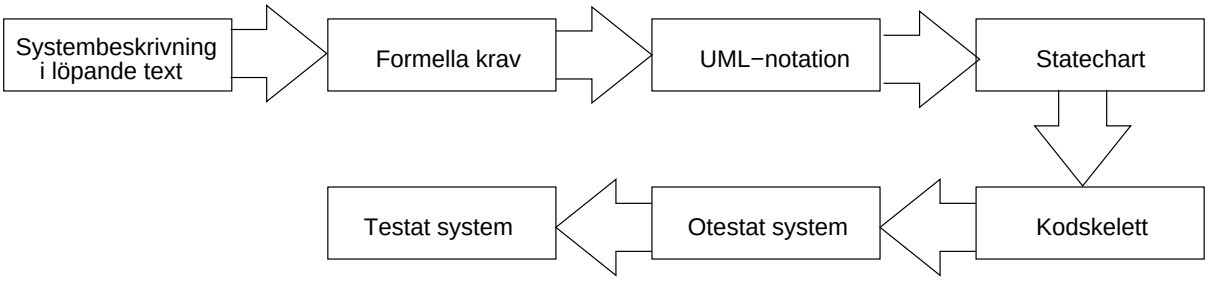
I samtliga fall är faserna stegen klart åtskiljda, varje steg beror av utdata från föregående steg och det är endast ett steg åt gången som utförs. Forskarna har i nästan samtliga fall utvecklat egna algoritmer för de olika faserna. Det här är onödigt anser jag, det skulle vara möjligt att definiera ett standardiserat gränssnitt mellan stegen vilket skulle medföra att forskare skulle kunna koncentrera sig på att förbättra endast en del av algoritmen. På så sätt skulle utvecklingen kunna ske mycket snabbare än idag. Tyvärr är konkurrensen mellan olika forskargrupper för stor för att de skall kunna samarbeta och enas om ett gemensamt gränssnitt.

6.1.2 Framtiden...

Idag går trenden mot en allt mer automatiserad programvaruutvecklingsprocess och tack vare utvecklingsmiljöer som till exempel Rational Rose

har man sedan ett antal år tillbaka kunnat generera kodskelett från UML-diagram. Den här trenden finns även när man läser metodbeskrivningarna för utveckling av statecharts, det slutgiltiga målet för samtliga undersökta metoder är att en dag ha en programvara som från givna indata automatiskt skapar korrekta statecharts. För två av metoderna är det här mer eller mindre en realitet idag och de kan idag användas för att automatiskt generera statecharts.

Men det slutar inte här, automatiseringsförsök finns både i stegen före och efter utveckling statecharts och programmering, försök till automatisering finns till exempel för framtagning av kravspecifikationer och för testning. Det finns forskningsprojekt som försöker skapa formella krav utifrån löpande text och som utifrån kraven kan generera UML-diagram. På samma sätt finns det försök att till viss del automatisera testningsfasen, man gör program som själva utvecklar testfall och kör testsekvenser, allt det här skulle kunna göra att programvaruutveckling i framtiden har följande faser:



Figur 6.1: Tänkbar framtida mjukvaruutvecklingssprocess.

Av stegen i illustrationen ovan är det endast steget mellan kodskelett och otestat system som behöver göras av designern. Även om vi är långt därifrån idag så är det tydligt att det finns en vilja att gå i den här riktningen.

Om man arbetar med reaktiva system bör man vara medveten om att utvecklingen går mot automatisering. Men hur ser det ut idag? Var någonstans i utvecklingen är vi nu? På TietoEnator till exempel har man en lång väg kvar till automatisering, idag används inte någon metod överhuvudtaget för att skapa statecharts utan det är upp till den enskilda designern att på något sätt ta fram en modell som klarar specifikationerna för systemet. Det är en farlig väg att inte ha några klara riktlinjer för hur utvecklingen av en av systemets grundstenar skall gå till. Om något allvarligt fel sker vid utvecklingen av statecharts kan det bli mycket kostsamt att rätta till det senare i processen. Om situationen är likadan på andra företag får vara osagt, men

om den är det måste det förändras, det är ett mycket stort risktagande att utveckla statecharts utan några hjälpmedel.

6.2 Testmodellen

Syftet för examensarbetet var följande:

> *Att utveckla en modell av det fysiska systemet för kraftförsörjningen till WCDMA-basstationer som möjliggör mjukvarutest av regler-programvaran.*

Det här är tveklöst uppfyllt, programvaran kan idag användas av Tieto-Enator för att testa sin mjukvara och snabbar på det här upp utvecklingsprocessen.

Under utvecklingsarbetet väcktes en tanke på att utveckla en mer generell applikation som kunde användas för att skapa testmodeller som kunde anpassas för olika projekt, den programvara som har utvecklats i examensarbetet är högst specialiserad.

6.2.1 Generell testmodell

Den generella testmodellen skulle vara uppbyggd av moduler med olika egenskaper, en modul skulle till exempel kunna svara på en viss signal med en acknowledgement eller skicka mätvärden inom ett visst intervall. Mätvärden skulle beräknas med hjälp av funktioner som utvecklaren fick tillhandahålla, modellens uppgift skulle bara vara att underlätta utvecklingen av testmodeller, inte att klara av efterlikna alla fall som kan uppstå. Med hjälp av den generella testmodellen skulle utvecklare på ett enkelt sätt kunna specificera vad som skulle hända när en viss signal togs emot, en sådan modell skulle betydligt förenkla och göra det billigare att implementera och testa reaktiva system.

Även om det är svårt att tänka ut alla möjliga testfall som ett reaktivt system kan tänkas behöva tror jag att det är möjligt att göra en meningsfull implementation. Den testmodell som byggt i det här examensarbetet hade inte krav på sig att bete sig *exakt* som det fysiska systemet, det viktiga var att

dess beteende låg nära det riktiga systemets och att det gick att avvika från det. Genom att utveckla ett relativt litet antal olika moduler är det möjligt att täcka upp de allra flesta testfall som kan behövas. Framförallt skulle en sådan applikation inbjuda till att tidigt i utvecklingsprocessen utveckla testmodeller.

6.3 Avslutning

UML-notationen statecharts får inte någon stor uppmärksamhet i metodlitteratur idag, än mindre uppmärksammas metoder för att utveckla statecharts. I och med att programvarusystem måste utvecklas allt snabbare och att de blir mer och mer komplicerade är det här något som förhoppningsvis kommer att ändras. Statecharts är en mycket intuitiv notation att arbeta med och det är enkelt att förstå hur ett system fungerar bara genom att titta på dess statechart. Dessutom gör den snabbare utvecklingstakten att det är ännu viktigare att grunden blir rätt, ett tidigt fel i utvecklingen kan straffa sig mångfalt senare. Den största anledningen till att statecharts inte har slagit igenom ännu är att modellering med signaler är ett annorlunda tankesätt mot vad man är van vid. I en radiobasstation till exempel är det naturligt att prata om signaler medan det i de flesta andra sorters programvaror är helt främmande. Det här förstärks även med att vi är vana med att tänka i form av *funktionsanrop* i stället för signaler, men med hjälp av utvecklingsmiljöer som Rose RT går det att förändra. Dessa miljöer fungerar lika bra att programmera i som de traditionella samtidigt som de möjliggör modellering av statecharts på ett intuitivt sätt. Jag är övertygad om att statecharts kommer att få en allt större uppmärksamhet dels tack vare de nya miljöer som utvecklas och dels tack vare att metoderna för att utveckla statecharts blir allt bättre.

Litteraturförteckning

[Har87] Harel D. *Statecharts: A visual formalism for complex systems.* Science of Computer Programming 8 (1987) 231-274.

[KEK01] Khriss I., Elkoutbi M och Keller R. *Automatic synthesis of behavioral object specifications from scenarios.* Transactions of the SDPS, September 2001, Vol 5, No 3, 53-77.

[KEK98] Khriss I., Elkoutbi M och Keller R. *Automating the Synthesis of UML Statechart Diagrams from Multiple Collaboration Diagrams.* Departemant dínformatique et de recherche operationnelle, Univerisite du Montreal.

[KM93] Koskimies K. och Mäkinen E. *Inferring state machines from trace diagrams.* Department of computer science, University of Tampere, Report A-1993-3.

[MS00] Mäkinen E., Systä T. *An Interactive Approach for Synthesizing UML Statechart Diagrams from Sequence Diagrams.* Proceedings of OOPSLA 2000 Workshop: Scenario-based round-trip engineering. 7-12.

[Rum91] Rumbaugh J., Blaha M., Premerlani W., Eddy F., Lorensen W. *Object Oriented Modeling and Design.* Englewood Cliffs : Prentice Hall, cop. 1991

[Sam02] Samek M. *Practical statecharts in C/C++ – quantum programming for embedded systems.* Lawrence, Kanada : CMP Books, c2002.

[SKK01] Schönberger S., Keller R. och Khriss I. *Algorithmic support for model transformation in object-oriented software development.* Concurrency and Computation: Practice and Experience 13(5): 351-383 (2001).

[OMG03] The object management Group. *OMG Unified Modeling Language Specification, version 1.5.* http://www.omg.org/ november 03.

[VT01] Vasilache S., Tanaka J. *Synthesizing statecharts from multiple interrelated scenarios.* Institute of Information Sciences and Electronics, University of Tsukuba. (2001)

[WS00] WHITTLE J., SCHUMANN J. *Generating Statechart Design From Scenarios.* Proceedings of International Conference on Software Engineerings (ICSE 2000), Limerick, Ireland, 2000.

[WS02] WHITTLE J., SCHUMANN J. *Scenario-based Engineering of Multiagent systems* In Formal Approaches to Agent-Based Systems, Chris Rouff(ed.), Kluwer, 2002

[WS02+] WHITTLE J., SCHUMANN J. *Statechart Synthesis From Scenarios: an Air Traffic Control Case Study.* In Proc. of Scenarios and State-Machines: models, algorithms and toolsåorkshop at the 24th Int. Conf. on Software Engineering (ICSE 2002), Orlando, FL, May, 20th 2002.

Bilaga A

Beteckningar

n	Antal PSUer.
P_i	Last för PSU i.
U_i	Spänningen hos PSU i.
I_i	Strömmen för PSU i.
$I1_i$	Strömmen för PSU i enligt beräkningsmetod 1.
$I2_i$	Strömmen för PSU i enligt beräkningsmetod 2.

Bilaga B

Förkortningar

KEK	Khriss, Elkoutbi, Keller. En metod för att utveckla statecharts.
MAS	Minimally Adequate Synthesizer. En metod för att utveckla statecharts.
OCL	Object Constraint Language, ett språk utan sidoeffekter som är speciellt framtaget för att beskriva restriktioner.
PSU	Power Source Unit, en modul som omvandlar trefas växelström till likström.
RBS	RadioBasStation. En sändare och mottagare för mobilsamtal.
UML	Unified Modelling Language, en samling med visuella beskrivningar av programvarusystem.
VT	Vasilache, Tanaka. En metod för att utveckla statecharts.
WCDMA	Wireless Code Division Multiple Access, tredje generationens mobilsystem.
WS	Whittle, Schumann. En metod för att utveckla statecharts.