CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Logikprogrammering för styrning av autonoma system

Kandidatarbete i data- och informationsteknik

VIKTOR ERIKSSON

ERIK FOGELSTRAND

EVELINA FRÄNNHAG

HAMPUS SIIK

Institutionen för data- och informationsteknik
CHALMERS TEKNISKA HÖGSKOLA
GÖTEBORGS UNIVERSITET
Göteborg, Sverige 2026

Kandidatarbete 2026

Logikprogrammering för styrning av autonoma system

VIKTOR ERIKSSON
ERIK FOGELSTRAND
EVELINA FRÄNNHAG
HAMPUS SIIK

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Institutionen för data- och informationsteknik
Chalmers tekniska högskola
Göteborgs universitet
Göteborg, Sverige 2026

Logikprogrammering för styrning av autonoma system
VIKTOR ERIKSSON ERIK FOGELSTRAND EVELINA FRÄNNHAG HAMPUS SIIK

Handledare: Jonas Almström Duregård, Institutionen för data- och informationsteknik
Examinator: Patrik Jansson, Institutionen för data- och informationsteknik
Medrättande lärare: Sandro Stucki, Institutionen för data- och informationsteknik

Kandidatarbete 2026
Institutionen för data- och informationsteknik
Chalmers tekniska högskola och Göteborgs universitet
SE-412 96 Göteborg
Telefon +46 31 772 1000

Typsatt i LaTeX
Göteborg, Sverige 2026

Logikprogrammering för styrning av autonoma system
VIKTOR ERIKSSON, ERIK FOGELSTRAND, EVELINA FRÄNNHAG,
HAMPUS SIIK

Institutionen för data- och informationsteknik
Chalmers tekniska högskola och Göteborgs universitet

Abstract

This thesis presents the development of a logic programming language for control of autonomous systems. A conceptual model is introduced in which an autonomous system is described as a set of input channels, output channels, and rules defining how data flows between them. Based on this model, the language Channelog is developed, inspired by Datalog and extended with support for channels and time-ordered data. The work results in a prototype of a Channelog runtime and a conceptual model. The prototype primarily consists of a parser generated using BNFC and a translator that converts the abstract syntax into SQL queries. With this prototype one can realize a control system.

Sammandrag

Detta arbete behandlar utvecklandet av ett logikprogrammeringsspråk för att styra autonoma system. En konceptuell modell presenteras där ett autonomt system beskrivs som en uppsättning inkanaler, utkanaler och regler som definierar hur data flödar mellan dessa. Utifrån modellen utvecklas språket Channelog, inspirerat av Datalog med stöd för kanaler och tidsordnade datamängder. Arbetet resulterar i en prototyp av exekveringsmiljö för Channelog samt en konceptuell modell. Prototypen består huvudsakligen av en parser genererad av BNFC och en översättare som översätter från den abstrakta syntaxen till SQL. Med denna prototyp kan man realisera en styrmodell.

Nyckelord: databaser, logikprogrammering, autonoma system

Tillkännagivanden

Denna rapport är skriven med mall skapad av Wolfgang Ahrendt baserad på mall av David Frisk. Vi vill även tacka Fabian Schlemmer som bidrog till planeringen av projektet.

Viktor Eriksson, Erik Fogelstrand, Evelina Frännhag, Hampus Siik,
Göteborg, Maj 2026

Innehåll

Figurer

1
Introduktion

Logikprogrammering är ett paradigm med sina rötter i bland annat forskning om AI och datorstödd bevisföring. Inom dessa områden har det länge använts, och används fortfarande [1]. Ytterligare en tidig tillämpning för logikspråk var databaser, genom utveckling av deduktiva databaser [2]. På senare år har logikspråk dessutom fått flera nya tillämpningsområden, bland annat informationsutvinning, nätverksövervakning och molntjänster [3]. Det faktum att paradigmen fortfarande används, och dessutom får nya tillämpningar, visar att den förblir relevant, och att vissa typer av program uttrycks väl i logikspråk. Denna fortsatta – och inom vissa områden nya – relevans ligger till grund för detta projekt, som utforskar möjligheten att utveckla ett logikspråk för programmering av autonoma system. Att vissa autonoma system gynnas av att lagra sin data i en databas kan ses i systemet som beskrivs i kapitel 1.1. Av denna anledning byggs språket ovan på en databas.

1.1 Syfte

Detta projekt ämnar utveckla ett språk, i vilket man kan beskriva ett reaktivt system med hjälp av logikprogrammering. Språket baseras på Datalog – ett logikprogrammeringsspråk – med vissa begränsningar och utökningar för att bättre anpassa språket till ändamålet. En utökning är att man kan definiera kanaler för kommunikation till och från systemet. Målet åstadkoms genom att skapa en konceptuell modell för generella reaktiva system, en språkbeskrivning och en delvis färdig prototyp. Prototypens huvudsakliga del är en översättare från det konstruerade språket till SQL-frågor men består också av en kanalkomponent som kan hantera kommunikation. För att mäta språkets och prototypens uttrycksförmåga skapas en beskrivning av ett hypotetiskt bokningssystem, enligt följande beskrivning:

Systemets upplägg: Systemet består av en klockinput, en bokningsinput, en krockoutput och en output för aktiva bokningar.

Önskat beteende:

- Information om bokningar skickas till bokningsinput.
- Nuvarande tid skickas till klockinput.

- Flera bokningar ska kunna tas emot och lagras.
- Ogiltiga bokningar ska filtreras bort.
- Tidskrockar mellan bokningar ska skickas till krockoutput.
- Aktiva bokningar ska skickas till output för aktiva bokningar.

Då språket kan uttrycka ett system med detta upplägg och önskade beteende, och prototypen kan köra det anses målet vara uppfyllt. Att detta har åstadkommits – och att målet därmed anses uppfyllt – visas i kapitel 4.5.

1.2 Avgränsningar

Projektet inkluderar inte användartester eller empirisk utvärdering av systemet. Fokus ligger istället på konstruktion och utvärdering av språket och dess översättning till SQL. Vidare genomförs inga prestandatester eller optimeringsstudier av den genererade SQL-koden eller den underliggande exekveringen i databashanteringssystemet.

Projektet behandlar heller inte fullständig implementation av ett produktionsklart system, utan begränsas till en prototyp av en översättare från det definierade språket till SQL, samt en enkel kanalkomponent för kommunikation.

2

Teori

2.1 Logikprogrammering

Logikprogrammeringsspråk är deklarativa språk uppbyggda kring hornklausuler, en formulering av logiska påståenden introducerade av och är döpta efter Alfred Horn [4]. En hornklausul är ett logiskt uttryck som uttryckt på disjunktiv form har högst *en* atom som inte är negerad [5]. Inom logikprogrammering används främst tre former av hornklausuler: strikta hornklausuler, faktum och målklausuler. Alla tre kan skrivas på implikationsform och disjunktionsform.

Strikta hornklausuler på implikationsform:

$$p \leftarrow q_1 \wedge q_2 \wedge \cdots \wedge q_n$$

och på disjunktionsform:

$$p \vee \neg q_1 \vee \neg q_2 \vee \cdots \vee \neg q_n$$

och tolkas som att p gäller om $q_1, q_2, \ldots, q_n$ gäller. Att de två formuleringarna är ekvivalenta följer direkt från definitionen av implikation, tillsammans med de Morgans lagar.

Fakta är specialfall av strikta klausuler med $q_1 = q_2 = \cdots = \top$, vilket på implikationsform blir:

$$p \leftarrow \top$$

och på disjunktionsform:

$$p$$

och tolkas som ”p är sant”.

Målklausuler är specialfallet då p snarare än $q_1, q_2, \ldots$ fixeras, och då till $\bot$. På implikationsform:

$$\bot \leftarrow q_1 \wedge q_2 \wedge \cdots \wedge q_n$$

och på disjunktionsform:

$$\neg q_1 \vee \neg q_2 \vee \cdots \vee \neg q_n$$

vilket tolkas som ”bevisa att alla q är sanna genom att anta motsatsen” [6].

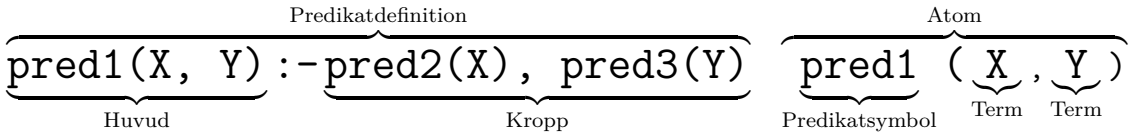
2.1.1 Prolog och Datalog

Prolog, från **pro**grammering i **log**ik, är ett deklarativt programmeringsspråk som i grunden används för att bevisa eller motbevisa logiska påståenden, men kan även användas som ett allmänt programmeringsspråk. Logiken uttrycks i hornklausuler på implikationsform, som i exemplet i 2.1.

```
human(socrates).          % Sokrates är en människa.
mortal(X) :- human(X).    % Alla människor är dödliga.
?- mortal(X).             % Vilka är dödliga?
```

Figur 2.1: Kort Prologprogram.

I detta exempel syns de semantiska strukturer som finns i Prolog: fakta (`human(socrates).`), regler eller *predikatdefinitioner* (`mortal(X) :- human(X).`) och frågor (`?- mortal(X).`). Dessa motsvarar fakta, strikta hornklausuler respektive målklausuler. Den exakta terminologin denna rapport kommer att använda för predikatdefinitioners och atomers ingående delar finns demonstrerad i figur 2.2.



Figur 2.2: Predikatdefinitionens och atomens anatomi.

Körning av ett Prologprogram innebär besvarande av alla frågor; i exemplet i figur 2.1 igen är svaret alla de `X` som tillfredsställer predikatet `mortal`. `X` tolkas här som en variabel då dess namn inleds med versal. En fråga utan variabler, alltså av typen `?-mortal(socrates).` – lägg märke till att `socrates` enbart är gemener – är också tillåten; svaret på denna blir ett sanningsvärde, i detta fall sann.

Eftersom körande av ett Prologprogram innebär besvarande av frågor finns tydliga paralleller till SQL och interaktion med databaser. I grunden strukturerar relationsdatabaser information i relationer, vilka överlappar med hur Prolog hanterar information. Det är därför relativt naturligt att använda Prolog som ett frågespråk, likt SQL, för en relationsdatabas; figur 2.3 nedan visar en SQL-fråga som ger samma resultat som Prologprogrammet i 2.1.

```
INSERT INTO Humans (name) VALUES ('Socrates');
CREATE VIEW Mortals AS SELECT name FROM Humans;
SELECT name FROM Mortals;
```

Figur 2.3: Samma exempel i SQL.

Observera att predikatet ”Alla människor är dödliga.” är en vy i relationsdatabasen. Svaret på frågan ”Vilka är dödliga?” är alltså härlett, snarare än att explicit lagrat [7].

Insikten att Prolog kan användas som frågespråk har legat till grund för utveckling av språket Datalog, ett logikspråk med fokus på databaser. Även om syntaxen mellan de två språken är samma finns det stora semantiska skillnader. Datalog hanterar regler och fakta som mängder, vilket gör att ordningen av dessa irrelevant, till skillnad från Prolog, vars semantik är beroende av delmålens inbördes ordning. Konkret kan detta leda till av vissa program i Prolog fastnar i oändlig rekursion om påståenden skrivs i fel ordning, medan detta inte är ett problem i Datalog [8].

2.2 Kommunikation mellan processer

Det finns flera arkitekturer och kommunikationsprotokoll för kommunikation mellan processer och i distribuerade system. Ett av dessa är Publisher-Subscriber-arkitekturen (Pub/Sub) vilken består av ett antal klienter som prenumererar på olika ämnen och ett antal publicister som publicerar till dessa ämnen. Det finns också Request-Reply-arkitekturen (Req/Rep) vilken bygger på att en enhet skickar en förfrågan till någon annan enhet, som skickar ett svar tillbaka.

2.2.1 Pub/Sub-arkitektur

I många Pub/Sub-system används en meddelandehanterare som tar emot meddelanden från publicerande noder och vidarebefordrar dem till prenumeranter av motsvarande ämne. Ett vanligt protokoll för denna arkitektur är MQTT, vilket kräver en meddelandehanterare [9]. Fördelen med denna modell är att kontakt endast behöver etableras med en enhet (meddelandehanterare) och till denna annonseras vad som vill prenumereras på. För denna arkitektur finns minst två bibliotek att använda, MQTT och ZeroMQ [9], [10]. Pub/Sub-arkitekturen möjliggör reaktiva system där komponenter kan reagera asynkront på inkommande händelser utan direkt koppling mellan avsändare och mottagare.

2.2.2 Req/Rep-arkitektur

I Req/Rep-arkitekturen påbörjas kommunikationen genom att en klient skickar en förfrågan till en server, vilken hanteras av mottagaren som sedan skickar ett svar. Ett exempel är HTTP där en klient begär en resurs från en server som svarar med den efterfrågade resursen, samt metadata om transaktionen [11].

3

Metod

Genomförandet av projektet förstås enklast som bestående av två delar; en teoretisk del med huvudfokus på utformning av språket och den konceptuella modellen, samt en praktisk del med huvudfokus på programmering av en exekveringsmiljö. Dessa två processer har pågått parallellt under projektets gång, och insikter gjorda i den ena kunde därför förstärka den andra.

3.1 Språk och konceptuell modell

Som nämnts i kapitel 1.1 ovan är ett av projektets huvudsakliga mål utveckling av ett logikprogrammeringsspråk. Av denna anledning utgick språkets utveckling från Datalog så som det beskrivits i kapitel 2.1.1. Utifrån denna språkdefinition skapades en ursprunglig beskrivning av språkets syntax i BNF. Denna modifierades successivt under arbetets gång, i takt med att syntax krävdes för att kunna beskriva nya semantiska strukturer.

Parallellt med specificeringen av den ursprungliga syntaxen skapades exempel av vad som skulle kunna uttryckas. Olika exempel utvecklades separat av gruppmedlemmarna, för att sedan presenteras för gruppen för diskussion. Därmed gjordes de olika gruppmedlemmarnas visioner kring språket tydliga, och diskussion kunde sedan föras tills en gemensam vision – med tillhörande modifierade exempel – var uppnådd. Dessa modifierade exempel låg sedan till grund för besluten kring vilka semantiska strukturer som skulle implementeras i språket. Nedan följer beskrivning av ett sådant exempel, och hur det påverkade språkdesignen.

Upplägg: Systemet består av en ljussensor och en lampa.

Önskat beteende:

- Om det senaste värdet från ljussensorn underskrider ett givet tröskelvärde ska lampan vara tänd.

- Om det senaste värdet från ljussensorn överskrider tröskelvärdet ska lampan

vara släckt.

Detta exempelsystem kräver att följande kan uttryckas i språket:

1. Indataenheter
2. Utdataenheter
3. Numeriska värden, och jämförelse av dessa
4. Mätvärdens kronologiska ordning

Vidare insågs att det önskade tillståndet för utenheten endast beror av datan från inenheten, alltså kan styrningen betraktas som en specifikation av hur ett flöde av indata översätts till ett flöde av utdata. Denna insikt låg till grund för introduktionen av *kanaler* som koncept; dessa beskrivs i större detalj i kapitel 4.1.

3.2 Implementation

Istället för att implementera en egen inferensmotor utvecklas ett system där Datalog-liknande regler och frågor översätts till SQL och exekveras av en relationsdatabas. Diskussion fördes i gruppen om det vore bättre att försöka bygga på en Datalogimplementation. Att SQL:s har bättre stöd och är mer välbekant för många programmerare övervägde dock, och systemet designas därför som en översättare från Datalog till SQL med mål att bevara semantiska egenskaper från Datalog. Det måste också finnas ett sätt att specificera vilken typ av data som systemet kan ta emot och vilka frågor det ska skicka ut svar på. Några komponenter är centrala:

- En språkspecifikation i LBNF [12] som kan tolkas av verktyget BNFC [13].
- En översättare från den abstrakta syntaxen till SQL-frågor som kan ställas till en relationsdatabas.
- En kanalhanterare som kan hantera kommunikation mellan systemet och givare och ställdon.

Designen begränsades till en delmängd av Datalog – skillnader mellan Channelog och Datalog beskrivs närmare i 4.2.2 – med utökningar för att kunna specificera kanaler och dessas egenskaper.

Implementationsarbetet bestod huvudsakligen av programmering i Haskell; baserat på tidigare erfarenhet gjordes bedömningen att Haskell var väl lämpat för språkutveckling. Programmeringen genomfördes sedan genom ett agilt arbetsflöde, med ett

möte varje tisdag. På dessa diskuterades vad som hade gjorts sedan förra, vad som skulle göras innan nästa, och ifall problem hade dykt upp som gruppen kunde lösa tillsammans.

3.2.1 Utveckling av översättaren

Första steget i utvecklingen av översättaren var undersökning av verktyg som kunde generera parser automatiskt. För detta ändamål valdes BNFC[13], då detta verktyg har stöd för att generera parser och lexer som kan användas i Haskell och bedömdes vara väldokumenterat.

För att kunna interagera med SQL-databasen påbörjades sedan en undersökning av Haskellbibliotek som möjliggör detta. Tre kandidater utvärderades:

1. `Hasql`
2. `sqlite-simple`
3. `HDBC`

Både `Hasql` och `sqlite-simple` riktar in sig på specifika SQL-implementationer – PostgreSQL respektive SQLite – till skillnad från `HDBC` som är implementationsagnostiskt. Den – i författarnas mening – största nackdelen med `HDBC` är att det inte uppdaterats på flera år; senaste uppdatering var i februari 2022. Det underhålls dock fortfarande[1] och bedömningen gjordes att möjligheten att vara implementationsoberoende övervägde potentiella nackdelar. Därav valdes `HDBC`.

Översättningen i sig handlade om att avbilda de semantiska strukturer som finns i Channelog på SQL:s semantiska strukturer. Den slutgiltiga avbildningen som utvecklades presenteras i kapitel 4.3.

3.2.2 Kanalhantering

Som nämnts i 3.1 ovan grundades konceptet kanaler i viljan att kunna beskriva dataflöde – alltså kommunikation – från inenheter, via styrenhet till utenhet; kommunikationen i en kanal är således alltid enkelriktad. Detta faktum, tillsammans med att ett godtyckligt antal fysiska enheter ska kunna vara anslutna till en och samma kanal ledde till beslutet att bygga kanalerna på en Pub/Sub-arkitektur.

Efter beslut kring arkitektur hade tagits fortsatte arbetet med undersökning av existerande kommunikationsbibliotek. Dessa undersökningar resulterade i två kandidater,

[1] I skrivande stund, 6 maj 2026.

1. MQTT [9]

2. ZeroMQ [14]

Vid en första anblick kan MQTT framstå som det bättre alternativet då det uteslutande fokuserar på Pub/Sub-kommunikation [9]. Detta är till skillnad från ZeroMQ, som beskrivs av utvecklarna som ett universellt meddelandebibliotek [10]. Denna universalitet bedömdes dock vara önskvärd, och ZeroMQs integration med Haskell bedömdes vara bättre för detta projekt. Vidare kräver MQTT en separat meddelandehanterare, vilket ZeroMQ inte gör. Dessa anledningar ledde till beslutet att ZeroMQ skulle användas. Därför implementerades kanaler via ZeroMQs ”publisher”- och ”subscriber”-sockets [10, s. 12–14], via Haskellbiblioteket `zeromq4-haskell` [15].

4

Resultat

Projektet har givit flera resultat. Det första är en konceptuell modell för autonoma system. Denna modell underbygger designen för nästa resultat: språket Channelog, ett logikspråk inspirerat av Datalog. Sist, men inte minst presenterar detta projekt en implementation av ett system – en exekveringsmiljö – byggd ovanpå en SQL-databas i vilken användaren kan specificera önskat beteende för ett autonomt system i Channelog.

4.1 Konceptuell modell

Ett autonomt system består av en *styrenhet* med regler och ett godtyckligt antal *kanaler*; in- och utenheter – så som givare, ställdon, motorer, knappar med flera – modelleras endast indirekt via kanaler.

4.1.1 Detaljbeskrivning

En kanal har en riktning – in eller ut, beroende på om de skickar data till eller från styrenheten – och en ordnad samling data d. Samlingen d kan betraktas som en kombination av en stack och en lista ; i likhet med listor kan data läsas från godtycklig position i d, och i likhet med stacker kan data endast skrivas genom att lägga till i början. Detta innebär att exempelvis beskriver $d[0]$ det senast tillagda värdet, $d[1]$ det näst senast tillagda och så vidare; denna design har valts för att göra beskrivning av följder innehållande de senaste värdena så smidig som möjligt.

Styrenheten ansvarar för styrning, alltså samordning mellan in- och utkanaler. Styrenheten kan beskrivas som S där:

$$S = (I, U, R),$$

I är mängden av inkanaler, U mängden av utkanaler och R mängden av regler som styr systemet. Här antas att det finns ett (och endast ett) element r i R för varje element i U, sådant att r avbildar alla element i I på *ett* element i U. Det går att

skriva till en inkanal och det går att få uppdateringar från utkanaler. När en inkanal skrivs till uppdateras utkanalerna direkt.

4.1.2 Kombination med andra modeller

I enkla system kan in och utenheter betraktas som direkta motsvarigheter till ingångs- respektive utgångskomponenter. I mer komplexa system kan dock dessa komponenter ha specifika interna modeller och beteenden, vilket gör att kanaler blir en lämplig abstraktion för att separera kommunikation från intern implementation.

Genom att inte kräva att in- och utkomponenter följer ett strikt gränssnitt kan kanaler användas för att integrera olika modeller på ett modulärt sätt. Detta möjliggör konstruktion av system där olika delar kan beskrivas med olika interna representationer men ändå integreras genom meddelandebaserad kommunikation.

Denna integration gäller även mellan program skrivna i Channelog och externa system, eller andra Channelog-program. Kommunikation kräver enbart förmåga att skicka och ta emot meddelanden via definierade kanaler.

4.2 Språk

Channelog är ett logikspråk, med syntax baserad på Datalogs. Det utgör en praktisk tillämpning av den konceptuella modell som presenterats i 4.1[1]. Som nämnt i kapitel 1 är Channelogs syfte att uttrycka styrning av autonoma system. Ett väldigt enkelt exempel presenterades i kapitel 3.1, och kod som motsvarar det finns i figur 4.1, se översättning i bilaga B.1.

```
1 => ljussensor :: (Int).
2 <= lampa.
3
4 lampa_tänd(X) :-
5     (X) <- ljussensor[0:1],
6     X < 4.
7
8 ?- lampa_tänd(X) => lampa.
```

Figur 4.1: Channelogkod för exemplet i kapitel 3.1 med 4 som tröskelvärde.

I kodexemplet definierar rad 1 och 2 in- och utkanalerna i systemet, predikatdefinitionen på rad 4 till 6 bestämmer när lampan ska vara tänd och frågan på rad 8 skickar till lampan om det ska vara på eller av. På rad 5 finns uttrycket `(X) <- ljussensor[0:1]`; denna typ av uttryck kallas kanaluppackning, en struktur unik

[1] Jämför med exempelvis förhållandet mellan relationsalgebra och SQL

för Channelog som kommer beskrivas närmare i 4.2.1. Kanaluppackningen tilldelar `X` värden ur `ljussensor`. Typen av frågan kan härledas att vara ett heltal och svaret på frågan kan därför tolkas som ”om utkanalen har ett värde ska lampan tändas, annars ska den släckas”.

Channelog kan även hantera något mer avancerade system; nedan följer en beskrivning av ett sådant system, dess önskade beteende och ett Channelogprogram som åstadkommer detta.

Uppbyggnad: Systemet består av en termometer, ett reglage för inställning av önskad temperatur, ett element och en indikatorlampa.

Önskat beteende: Användaren kan via reglaget ge angivelse om önskad temperatur. Om termometerns värde underskrider detta ska användarens önskade temperatur skickas till elementet, och indikatorlampan ska tändas.

I figur 4.2 visas Channelogkod för att åstadkomma detta beteende, se översättning i bilaga B.2.

Från predikatdefinitionerna följer att kanalerna `element` och `element_lampa` är av typen heltal. I detta fall kan utkanalen `element_lampa` tolkas som att om det finns utdata i kanalen ska den sättas på och om det inte finns utdata så ska den vara avstängd.

```
 1 => reglage :: (Int).
 2 => termometer :: (Int).
 3 <= element.
 4 <= element_lampa.
 5
 6 element_ny_temp(ÖnskadTemp) :-
 7     (ÖnskadTemp) <- reglage[0:1],
 8     (NuvarandeTemp) <- termometer[0:1],
 9     NuvarandeTemp < ÖnskadTemp.
10
11 element_lampa_på(På) :-
12     element_ny_temp(På).
13
14 ?- element_ny_temp(X) => element.
15 ?- element_lampa_på(Y) => element_lampa.
```

Figur 4.2: Exempelprogram i Channelog.

I exemplet definieras två predikat, `element_ny_temp` och `element_lampa_på`. Frågorna på rad 14 och 15 kommer skicka sina svar till utkanalerna `element` respektive

`element_lampa`; svaren är det värden på `X` respektive `Y` som gör predikaten sanna. Detta innebär att om jämförelsen på rad 9 är sann kommer senaste värdet från reglagekanalen att skrivas och annars skrivs en tom lista. Nya meddelanden kommer att skrivas till utkanalerna varje gång predikaten förändras, det vill säga då ny data kommer från inkanalerna; ny data förändrar `reglage[0:1]` (senaste värdet i `reglage`kanalen) eller `termometer[0:1]`.

4.2.1 Syntax och semantik

I detta kapitel beskrivs språkets syntax och semantik. Dessa beskrivningar är inte uppdelade i separata delar, utan varje språkelements semantik beskrivs tillsammans med dess syntax. För att formellt definiera syntaxen för Channelog presenteras grammatiken i Extended Backus-Naur Form (EBNF) i figur 4.3. I exekveringsmiljön används grammatiken i Labeled Backus-Naur Form (LBNF), som kan ses i bilaga A, för att generera det abstrakta syntaxträdet med hjälp av BNFC.

```
<Program> := { ( <Inkanal> | <Utkanal> | <Hornklausul> ) "." }
<Hornklausul> := <Atom> ":-" <Faktor> { "," <Faktor> }
               | "?-" <Atom> "=>" <Kanalnamn>
<Faktor> := <Atom> | <Kanaluppackning> | <Jämförelse>
<Atom> := <Predikat> "(" [ <Term> { "," <Term> } ] ")"
<Kanaluppackning> := "(" [ <Term> { "," <Term> } ] ")" "<-" <Termkälla>
<Jämförelse> := <Term> ( "<" | ">" | "=" | "<=" | ">=" | "!=" ) <Term>
<Term> := <Variabelnamn> | <Heltal> | <Sträng>
<Termkälla> := <Kanalnamn> [ "[" <Heltal> ":" <Heltal> "]" ]
<Inkanal> := "=>" <Kanalnamn> "::" <Typsignatur>
<Utkanal> := "<=" <Kanalnamn>
<Kanalnamn> := <Predikat>
<Typsignatur> := "(" ( "Int" | "Str" ) { "," ( "Int" | "Str" ) } ")"
```

Figur 4.3: EBNF för Channelog. {}: noll eller flera, [ ]: noll eller en.

Channelog använder, utöver strängar och heltal, tre typer av symboler: predikatsymboler, kanalnamn och variabelnamn. Predikatsymboler och kanalnamn består av gemener, siffror och understreck, men måste börja med en gemen. Semantiskt representerar symbolerna predikat respektive kanaler i ett program. Variabelnamn kan bestå av både versaler och gemener, siffror och understreck, men måste börja med en versal, och representerar variabler. Reguljära uttryck för symbolerna visas i figur 4.4. Kanalnamn ingår däremot i EBNF-definitionen i figur 4.3, eftersom de har samma definition som predikatsymboler.

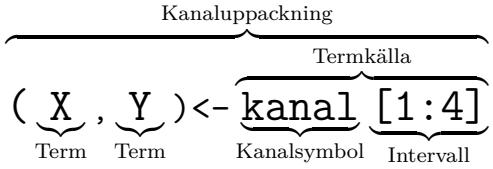
```
<Variabel> := <Versal> ( <Bokstav> | <Siffra> | "_" )*
<Predikat> := <Gemen> ( <Gemen> | <Siffra> | "_" )*
```

Figur 4.4: Reguljära uttryck för symboler i Channelog.

Språket kan övergripande beskrivas som en mängd kanaldeklarationer och hornklausuler. Hornklausuler är i form av frågor eller predikatdefinitioner. Predikatdefinitionens huvud består av en atom och kroppen av en mängd atomer, kanaluppackningar och booleska jämförelseoperationer. Atomer och frågor följer samma definition som beskrivet i kapitel 2.1, med begränsningarna att fakta inte är tillåtna och att frågor måste bindas till en utkanal. Channelog tillåter inte fakta, eftersom inkanaler fyller motsvarande roll; i Datalog utgör fakta den grundläggande data som predikatdefinitioner resonerar om, medan Channelogs predikatdefninitioner resonerar om data från inkanaler. Likheter och skillnader mellan Channelog och Datalog beskrivs närmare i 4.2.2.

I en välformad predikatdefinition är alla variabler bundna och huvudet utan konstanter. Konstanter i ett huvud tillåts inte eftersom det skapar fakta, vilket inte existerar i Channelog. En variabel i predikatdefinitionens kropp är bunden om den existerar i en eller flera atomer och/eller kanaluppackningar. Alltså är variabler i jämförelseoperationer och huvudet inte nödvändigtvis bundna. Jämförelseoperationer binder inte variabler för att vara konsekvent med att inte tillåta faktum; exempelvis etablerar `p(X) :- X < 6.` faktumet ”p är sann för alla heltal mindre än sex”. Förekomst i huvudet binder inte heller en variabel; alla variabler i huvudet måste finnas i kroppen.

Kanaluppackning används för att få åtkomst till en inkanals data likt en atom, vilket betyder att inkanalens ordnade data omvandlas till en oordnad mängd. Kanaluppackning består av en lista av termer och en termkälla, där termkällan består av en kanalsymbol och ett frivilligt intervall. Listan av termer motsvarar inkanalens datafält medan intervallet anger vilket urval av datapunkter i inkanalen som ska inkluderas. Om ett intervall inte anges kommer alla datapunkter i inkanalen inkluderas i kanaluppackningen. Intervallet, likt Python, inkluderar det första indexet och exkluderar det sista, samt tillåter negativa index och inkanalen nollindexeras. För anatomin av kanaluppackning, se figur 4.5.



Figur 4.5: Kanaluppackningens anatomi.

För att använda en inkanal behöver den deklareras med en typsignatur. En väldefinierad typsignatur är en tupel, som inte är tomma tupeln, bestående av nyckelorden för strängar och heltal. Vid kanaluppackning måste listan med termer stämma överens i längd och typer med typsignaturen av inkanalen i termkällan. Till skillnad från inkanaler behöver utkanaler inte deklareras och kan inte deklareras med en typsignatur. Exempelkod av Channelog kommer fortsättningsvis alltid deklarera utkanaler för tydlighet.

4.2.2 Likheter och skillnader jämfört med Datalog

Då Channelog är baserat på Datalog finns naturligtvis många likheter mellan de två språken. Den mest självklara likheten är användandet av hornklausuler. Predikatdefinitioner som de förekommer i Channelog är avsiktligt designade att efterlikna Datalogs snarare än Prologs. Detta innebär att alla termer som förekommer i en predikatdefinitions huvud också måste förekomma i dess kropp, samt att olika predikatdefinitioners inbördes ordning inte påverkar programmets semantik. Vidare stöttar Channelog enkel rekursion, men endast enkel rekursion, precis som Datalog. Ett exempel på rekursion i Channelog visas i kapitel 5.2.3.

Skillnader existerar självklart också, de kanske mest uppenbara är gällande kanaler; konceptet kanaler saknar helt motsvarighet i Datalog. Vidare kan inkanaler annoteras med typer och indexeras i predikatdefinitioner. Viktigt att anmärka är att i Channelog måste inkanaler deklareras; deklaration är inte ett koncept som existerar i Datalog.

Mindre uppenbara skillnader finns också; Channelog tillåter, till skillnad från Datalog, jämförelse mellan termer i form av booleska jämförelseoperationer, utöver den implicita likhetsoperationen mellan likadana variabler. Vidare tillåter Datalog hornklausuler på formen `human(socrates)` som kortform av `human(socrates) :- true`. Channelog tillåter inte någotdera, en konsekvens av att – som nämnts ovan – kanaler nu fyller funktionen som sanningskällor. Detta innebär vidare att medan ett Datalogprograms fakta inte förändras under programmets körning så kommer ett Channelogprogram successivt bygga upp en större och större mängd ”sanningar”, allt eftersom att data kommer från inkanalerna.

4.2.3 Skillnader jämfört med modellen

Precis som vid jämförelse med Datalog förekommer skillnader huvudsakligen i förhållande till kanaler. Modellen förutsätter möjligheten att indexera både in- och utkanaler; språket tillåter endast indexering av inkanaler. Denna skillnad kommer sig av att modellen och språket fyller olika funktioner. Modellens huvudsakliga syfte är att göra system begripliga, varför enhetlig behandling av in- och utkanaler är önskvärd. Språkets är att styra faktiska system, i vilket fall det ofta räcker att endast indexera indata.

Ytterligare en skillnad är hur regler fungerar. I modellen antas en regel per utkanal, men då antas att reglerna kan innehålla logiska disjunktioner; enda sättet att uttrycka logisk disjunktion i Channelog är genom att inkludera en predikatdefinition per element i disjunktionen.

4.3 Språköversättning

Som nämnts i kapitel 3 har Channelog utvecklats med avsikten att interagera med en SQL-databas. För att möjliggöra detta översätts atomer, predikatdefinitioner,

frågor och kanaler – alltså de strukturer som existerar i Channelog – till strukturer i SQL. Nedan följer detaljbeskrivning av hur detta görs. Viktigt att notera är att den exekveringsmiljö som utvecklats inte klarar av att genomföra alla översättningarna; vilka som inte stöttas beskrivs i kapitel 5.2.

4.3.1 Inkanaler

Som nämndes i 4.1.1 kan inkanaler liknas vid en stack, alltså en ordnad mängd där nya värden läggs till på toppen. För att åstadkomma detta i SQL översätts varje inkanal till en tabell, med samma namn som kanalnamnet och en id-kolumn som primärnyckel. Id-kolumnens värde är sådan att den senaste indatan i inkanalen har det högsta värdet i id-kolumnen och är ett mer än den näst senaste indatan. För den specifika implementationen i SQLite åstadkoms id-kolumnens beteende med nyckelordet `AUTOINCREMENT`[16]. De andra kolumnerna i tabellen får sin ordning och typ från typsignaturen av inkanalen, se figur 4.6.

```
=> inkanal :: (Int, Str).
```

(a) Channelog.

```
CREATE TABLE inkanal (
    id INTEGER PRIMARY KEY
         AUTOINCREMENT,
    A  INTEGER NOT NULL,
    B  TEXT    NOT NULL);
```

(b) SQL.

Figur 4.6: Översättning av inkanal.

Kanaluppackning representeras inte i sig i SQL, utan är uppdelad i en vy och ett urval på denna vy i predikatdefinitioner – termkällan respektive listan av termer. Om termkällan inte har ett intervall kommer det skapas en vy med alla rader, men utan id-kolumnen, från kanaltabellen. Har termkällan ett intervall kommer en extra WHERE-klausul inkluderas i vyfrågan som enbart tar med de rader där id-kolumnens värde är mellan från-indexet och till-indexet (exkluderat), se figur 4.7.

```
inkanal[1:4] % Termkälla
```

(a) Channelog.

```
CREATE VIEW inkanal_intervall AS
SELECT A, B
FROM inkanal
WHERE id <= (SELECT max(id) FROM inkanal) - 1
  AND id >  (SELECT max(id) FROM inkanal) - 4;
```

(b) SQL.

Figur 4.7: Översättning av termkälla med intervall.

4.3.2 Variabelkopplingar i predikatdefinitioner

I kroppen på en predikatdefinition begränsar variabler sanningsvärdena av predikatet. För att uppnå detta beteende behöver termer i atomer och kanaluppackningar med samma variabelnamn ”kopplas ihop” med ekvivalenser. Termer med samma variabelnamn är ekvivalenta eftersom de representerar samma variabel, som kan anta endast ett specifikt värde.

Atomer och kanaluppackningar är representerade i SQL sådana att de kan behandlas på samma vis, genom ett urval på den vy/tabell med samma namn som predikatsymbolen/kanalnamnet. Fortsättningsvis kommer kanaluppackningar och atomer i kroppen på en predikatdefinition gemensamt benämnas som kroppselement och kommer att betraktas som tabeller. Notera att atomer representeras som ett urval på en vy av en predikatdefinition.

Variabelkopplingar kan modelleras som θ-joins med ett ekvivalensvillkor, eftersom förekomster av samma variabelnamn i olika kroppselement introducerar likhetsrestriktioner mellan motsvarande termer. Därför översätts en predikatdefinitions kropp till `INNER JOIN`-operationer (förkortas som `JOIN` fortsättningsvis) på tabellerna motsvarande de ingående kroppselementen, som i figur 4.8. I detta och följande exempel kommer variabelnamn användas som kolumnnamn för att göra exemplen så tydliga som möjlig; för beskrivning av hur kolumnnamn faktiskt genereras, se kapitel 4.4.

```
_ :- p(X), q(X), r(X).
```

(a) Channelog.

```
... FROM p
JOIN q ON q.X = p.X
JOIN r ON r.X = q.X;
```

(b) SQL.

Figur 4.8: Översättning av enkel predikatdefinition.

Eftersom kroppselementen i en predikatdefinition utgör en oordnad mängd, men `JOIN`-klausuler nödvändigtvis är ordnade, måste översättaren introducera en ordning av `JOIN`-operationer sådan att varje `JOIN` är väldefinierad. För att en `JOIN`-operation ska vara korrekt krävs att de tabeller som används i dess villkor redan har introducerats i tidigare steg av frågan. Detta innebär att en `JOIN`-operation endast kan utföras på ett kroppselement efter att alla variabler som begränsar den – alltså begränsar vilka rader ur dess motsvarande tabell som ska ingå i resultatet – har bundits till tidigare relationer. Exemplet i figur 4.9 förtydligar detta problem.

```
_ :- p(X), q(X, Y), r(Y).
```

(a) Channelog.

```
... FROM p
JOIN r ON r.?=p.?
JOIN q ON q.X=p.X;
```

(b) Fel ordning.

```
... FROM p
JOIN q ON q.X=p.X
JOIN r ON r.Y=q.Y;
```

(c) Rätt ordning.

Figur 4.9: Översättning med fel och rätt ordning av `JOIN`-operationer.

I den felaktiga ordningen försöker relationen r `JOIN`:as innan relationen q, trots att p och r nödvändigtvis behöver begränsas via q. Därmed saknas nödvändig bindning, vilket leder till en icke väldefinierad `JOIN`-operation. För att introducera ordning och säkerställa väldefinierade `JOIN`-operationer skapas en graf motsvarande kroppen enligt algoritmen beskriven i pseudokod i figur 4.10 nedan.

```
kropp: lista av kroppselement
graf = tom

för kroppselement ke i kropp:
    skapa nod motsvarande ke i graf

för kroppselement ke i kropp:
    för variabel v i ke:
        kes = alla obesökta kroppselement i kropp
              som också innehåller v
        skapa riktad kant i graf från ke
               till varje kroppselement i kes
```

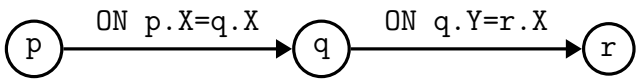
Figur 4.10: Pseudokod beskrivande algoritm för omvandling från kropp till graf.

I grafen motsvarar noder kroppselement och kanter specifika restriktioner mellan kroppselementen, se figur 4.11. En `JOIN`-operation mellan två relationer motsvarar

således en kant i grafen. Genom denna representation kan en korrekt ordning av `JOIN`-operationer erhållas, genom att traversera grafen. Notera att inte alla predikatdefinitioner kan översättas till en följd av `JOIN`-operationer med denna algoritm. Predikatdefinitioner vars motsvarande graf innehåller cykler kan inte representeras som ett träd och kräver därför en annan strategi för översättning; detta diskuteras närmare i kapitel 5.2.1.

```
_ :- p(X), q(X, Y), r(Y).
```

(a) Exempel på predikatdefinition.



(b) Grafrepresentation av kroppen.

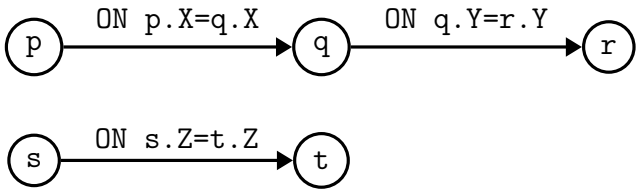
Figur 4.11: Exempel på predikatdefinition och dess grafrepresentation.

Restriktionerna som binder samman kroppselementen är i praktiken transitiva. I exemplet i figur 4.9 och figur 4.11 igen: relationsparen p och q samt q och r begränsar varandra direkt genom restriktioner, men det finns en indirekt restriktion mellan p och r; när r `JOIN`:as med resten begränsas även rader från p.

Om grafen består av flera disjunkta, sammanhängande delgrafer, som i figur 4.12, saknas gemensamma variabler mellan delgraferna. Eftersom restriktion är en transitiv relation är det enda fallet då två kroppselement inte begränsar varandra om de är disjunkta. Om en predikatdefinition innehåller två kroppselement som inte begränsar varandra, kommer resultatet vara ekvivalent med den kartesiska produkten av kroppselementens motsvarande tabeller. Detta eftersom de två kroppselementen inför restriktioner på disjunkta mängder av tupler, och kan därför uppfyllas oberoende av varandra. Som resultat kommer alla kombinationer av deras tupler vara giltiga i konjunktionen, alltså en kartesisk produkt.

```
_ :- p(X), q(X, Y), r(Y), s(Z), t(Z).
```

(a) Exempel på predikatdefinition.



(b) Grafrepresentation av kroppen.

Figur 4.12: Exempel på restriktioner av disjunkta variabelkopplingar.

4.3.3 Predikatdefinitioner

Vid översättning av hela predikatdefinitioner behöver huvud och kropp behandlas på olika sätt. Huvudet består som bekant av en atom, medan kroppen består av en konjunktion av en eller flera atomer, kanaluppackningar och jämförelseoperationer. I 4.3.2 beskrevs hur variabelkopplingar modelleras och nedan kommer resterande delar av en predikatdefinition att beskrivas, samt hur de passar ihop i en gemensam vy i SQL.

Utöver kroppselement kan även jämförelseoperationer förekomma i kroppen av predikatdefinitioner. Dessa består av en jämförelseoperator och två termer. Som nämnt i kapitel 4.2.1 måste termerna vara bundna för en väldefinierad predikatdefinition vilket betyder att det går att representera termerna som kolumner i tabeller från kroppselement för variabler, eller konstanter. I SQL uttrycks alltså jämförelseoperationer i en `WHERE`-klausul, efter variabelkopplingarna har gjorts, med de specificerade kolumnerna och konstanterna, se figur 4.13.

```
_ :- p(X), q(X), X > 4.
```

(a) Channelog.

```
... -- JOIN-operationer
WHERE p.X > 4;
```

(b) SQL.

Figur 4.13: Översättning av jämförelseoperation.

Konstanter och variabler kan förekomma som termer i kroppselement. Konstanta termer översätts till en `WHERE`-klausul i SQL, där den aktuella kolumnen begränsas till det givna konstanta värdet. Även de fall där det finns dubbletter av en variabel i ett kroppselement översätts till en `WHERE`-klausul, med en likhet mellan de termer med samma variabel. Detta säkerställer att endast de tupler som uppfyller villkoren inkluderas i resultatet, se figur 4.14.

```
_ :- p(2,"a", X, X).
```

(a) Channelog.

```
...
WHERE p.A = 2
  AND p.B = 'a'
  AND p.C = p.D;
```

(b) SQL.

Figur 4.14: Översättning av termer i kroppselement.

Varje variabel i huvudet av en predikatdefinition kan alltid bindas ihop med minst en variabel i ett kroppselement, som nämnt i kapitel 4.2.1. I SQL innebär detta att värdet av variabeln kan hämtas från godtycklig kolumn där variabeln förekommer.

Resultatet blir korrekt eftersom alla förekomster av samma variabel nödvändigtvis antar samma värde i den resulterande relationen genom `JOIN`-villkoren. Huvudet åstadkoms således i `SELECT`-satsen om rätt kolumner väljs, se figur 4.15.

```
h(X,Y) :- p(X), q(X, Y), r(Y).
```

(a) Channelog.

```
SELECT p.X, q.Y
FROM ...;
```

(b) SQL.

Figur 4.15: Översättning av huvud.

I det fall där en variabel endast förekommer i *en* term saknar den kopplingar till övriga termer i kroppen. En sådan variabel introducerar därmed inga restriktioner på övriga termer. I SQL innebär detta att ingen explicit begränsning behöver införas, däremot kan variabelns värde påverka resultatet om variabeln även förekommer i huvudet av predikatdefinitionen.

Slutligen kan en predikatdefinition som helhet översättas till en vy i SQL, med huvudatomens predikatsymbol som namn. `SELECT`-satsen specificerar de kolumner som motsvarar variablerna i predikatdefinitionens huvud. `FROM`-delen innehåller en tabell från varje disjunkt delgraf från variabelkopplingarna, vilket motsvarar en kartesisk produkt. Därefter appliceras `JOIN`-operationer i ordningen från traverseringen av grafen, som säkerställer att alla operationer är väldefinierade. Avslutningsvis används en `WHERE`-klausul för att införa restriktioner från eventuella konstanter, dubbletter och jämförelseoperationer i predikatdefinitionen. Se figur 4.16 för ett komplett exempel på en predikatdefinition; exemplet har samma variabelkopplingar som i figur 4.12. Observera variabeln `W` som endast förekommer i *en* term i kroppen.

```
h(X,W) :- p(X,"c"), q(X,Y),
          r(5,Y), s(Z),
          t(Z, W),
          X != Z, Y < 5.
```

(a) Channelog.

```
CREATE VIEW h AS
SELECT p.X, t.W
FROM p, s
JOIN q ON q.X = p.X
JOIN r ON r.Y = q.Y
JOIN t ON t.Z = s.Z
WHERE p.B = 'c'
  AND r.A = 5
  AND p.X <> s.Z
  AND q.Y < 5;
```

(b) SQL.

Figur 4.16: Översättning av predikatdefinition.

4.3.4 Frågor och utkanaler

Både frågor och predikatdefinitioner är hornklausuler, men översättningen av frågor är sådan att den är en delmängd av översättningen av predikatdefinitioner. Eftersom frågor endast består av en atom, och inget huvud i samma mening som predikatdefinitioner, är de enda begränsningarna som kan förekomma på frågor konstanter och dubbletter av variabler. Alltså representeras en fråga av en vy med en `WHERE`-klausul. Observera likheten mellan predikatdefinitionen i figur 4.14 och frågan i figur 4.17.

Utkanaler används för att få ett resultat från Channelogprogrammet och modelleras därför som en `SELECT`-sats. Eftersom utkanaler nödvändigtvis sammanfaller med frågor används `SELECT`-satsen på frågevyn, se nedre `SELECT`-sats i figur 4.17. Resultatet av denna `SELECT`-sats är en lista av n-tupler, där varje rad i vyn utgör en tupel och n är antalet kolumner i vyn, alltså ariteten av frågans atom. I exemplet i figur 4.17 har predikatet `p` en aritet av fyra och kommer därför ge en lista av 4-tupler till utkanalen. För hur och när utkanalens `SELECT`-sats används, se 4.4.

```
?- p(2,"a", X, X) => utkanal.
```

(a) Channelog.

```
CREATE VIEW fråga AS
SELECT * FROM p
WHERE p.A = 2
  AND p.B = 'a'
  AND p.C = p.D;

SELECT * FROM fråga;
```

(b) SQL.

Figur 4.17: Översättning av fråga.

4.3.5 SQL-implementation

Den språköversättning som presenterats här har utvecklats mot SQLite, då denna implementation bedömts lämpligast för de behov som detta projekt har. Det innebär inte nödvändigtvis att översättningen är bunden till denna specifika SQL-implementation. Arbete lagts på att försöka minimera implementationsspecifika funktioner och i så stor mån som möjligt endast använda funktioner i SQL-standarden. Dock har SQL-standarden inte kunnat följas helt[2]. I inkanaler används typen `TEXT` för strängar; typen stöds av en del SQL-implementationer men är inte en del av standarden. Nyckelordet `AUTOINCREMENT` används för implementationen av ordnade mängder vilket är specifikt för SQLite. Potentiella möjligheter till portande till annan SQL-implementation diskuteras närmare i kapitel 5.3.1.

Vid översättning av Channelog till SQL finns det varken explicita namn på mål eller termkällor, eller explicita kolumnnamn för vyer och tabeller. För mål och

[2]Inte minst på grund av den bristande tillgängligheten av SQL-standarden för privatpersoner.

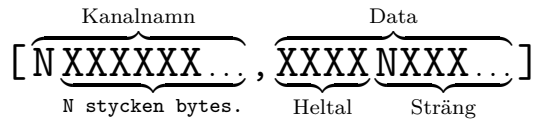
termkällor baseras namn på vyerna från atomen i målet respektive intervallet och kanalnamnet i termkällan. Namnen är deterministiskt genererade och är unika för unika atomer/termkällor. Kort omvandlas målet/termkällans representation i det abstrakta syntaxträdet till en sträng som sedan saniteras för databasen. För konsekvent hämtning av rätt kolumn, vid konstruktion av vyer, är kolumnnamn också deterministiskt genererade. Kolumnnamnen är baserade på positionen av kolumnen i följden `A, B, ..., Z, AA, AB, ...`.

4.4 Exekveringsmiljö

Översättningen går till på det viset att Channelogs grammatiska struktur är definierad i LBNF-format, vilket passeras till verktyget BNFC. Verktyget skapar en lexer, parser och datastrukturer som representerar de grammatiska strukturer som finns i Channelog. Lexern kan sedan köras på Channelogkod och genererar tokens som parsern använder för att konstruera det abstrakta syntaxträdet. Noderna i trädet avgör om utvärderaren ska konstruera en inkanal med motsvarande tabell i databasen, skapa en vy i databasen eller om den ska förbereda en vy motsvarande en fråga med tillhörande utkanal.

När utvärderaren stöter på en kanaldeklaration kollar den på de underordnade noderna och plockar ut detaljer såsom kanalnamnet och typsignaturen. Dessa paketeras i ett mer lätthanterligt format och används för att förbereda en `CREATE TABLE`-klausul på samma vis som i figur 4.6.

Kommunikation mellan styrenheten och periferienheter sker via sockets. Styrenheten binder en port till en Subscriber-socket för inkanaler och en port till en Publisher-socket för utkanaler. Inenheter förväntas koppla upp sig mot styrenhetens Sub-socket som publicister och ställdon mot dess Pub-socket som prenumeranter, i enlighet med hur Pub/Sub-kommunikation beskrivits i kapitel 2.2.1. Meddelanden som skickas paketeras som bitsträngar och är tvådelade; den ledande delen innehåller kanalnamnet och den efterträdande delen är en lista av data. Strängar (inklusive kanalnamn) paketeras som Pascalsträngar, alltså bitsträngar med strängens längd som första byte. Heltal paketeras som 32 bitars heltal, med den mest signifikanta byten först. Detta illustreras i figur 4.18.



Figur 4.18: Meddelande på en kanal med typsignatur (int,str); ett tecken motsvarar en byte.

För avkodning i styrsystemet matchas först kanalnamnet i meddelandet mot en inkanal, för att hämta dennas typdeklaration. Typdeklarationen används sedan

för att avkoda bitsträngen, under antagandet att strängar och heltal kodats som beskrivits ovan.

4.5 Målsystem

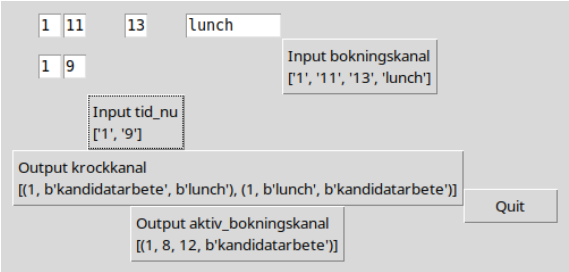
I kapitel 1.1 beskrevs ett exempelsystem utvecklat som måttstock för språket och exekveringsmiljön. Detta kan implementeras i Channelog, som visas i figur 4.19.

```
=> bokningskanal :: (Int, Int, Int, Str).
=> tid_nu :: (Int, Int).
<= krockkanal.
<= aktiv_bokningskanal.

bokning(Dag, Start, Slut, Namn) :-
  (Dag, Start, Slut, Namn) <- bokningskanal,
  Start < Slut.

aktiv_bokning(Dag, Start, Slut, Namn) :-
  (Dag, Timme) <- tid_nu[0:1],
  bokning(Dag, Start, Slut, Namn),
  Start <= Timme,
  Slut > Timme.

krockar(Dag, A, B) :-
  bokning(Dag, Start1, Slut1, A),
  bokning(Dag, Start2, Slut2, B),
  Start1 < Slut2,
  Start2 < Slut1,
  A != B.

?- krockar(Dag, A, B)
  => krockkanal.

?- aktiv_bokning(Dag, Start, Slut, Namn)
  => aktiv_bokningskanal.
```

Figur 4.19: Exempelprogram där krockar och aktiva bokningar skickas.

I enlighet med hur in- och utkanaler beskrivits i 4.2.1 översätts in- respektive output till kanaler. Predikatdefinitioner används för att beskriva systemets beteende och frågor används för att skicka utdata. Denna Channelogkod kan översättas till SQL av prototypen, och den automatiskt genererade översättningen kan ses i bilaga B.5.

För att testa funktionaliteten skapades ett enkelt pythonprogram med grafiskt



Figur 4.20: Enkelt grafiskt gränssnitt

användargränssnitt, vilket visas i figur 4.20. Gränssnittet utvecklades med `Tkinter` [17].

Detta pythonprogram skickar och tar emot data via ZeroMQ-sockets, vilka skapas med biblioteket `PyZMQ` [18]. Pythonprogrammet skickar data till prototypen via inkanaler, vilken svarar med data i utkanalerna. Genom att använda pythonprogrammet kunde användare skicka data till prototypen och fick sedan automatiskt svar, i enlighet med vilka frågor som ställts i Channelogkoden.

5

Diskussion

5.1 Kommunikationsarkitektur

Som nämnts i kapitel 4.4 är systemets kommunikation uppbyggd kring en Pub/Sub-arkitektur. Inenheter publicerar data, utenheter prenumererar på data, styrsystemet gör båda. Således erhålles ett reaktivt system; arbete i styrsystemet sker nödvändigtvis som reaktion på data från en eller flera inenheter. Pub/Sub-arkitektur är inte det enda sätt att åstadkomma ett reaktivt system; det hade även kunnat åstadkommas genom en Req/Rep-arkitektur, en arkitektur som beskrivits närmare i kapitel 2.2.2.

Ingen av arkitekturerna är nödvändigtvis överlägsen den andra; oavsett val får avvägningar göras. Valet av Pub/Sub innebär att exekveringsmiljön är väl lämpad till situationer där utenheter endast behöver uppdateras när en förändring sker, exempelvis styrning av lampor eller termostater; om ljusförhållandena respektive temperaturen inte ändras finns heller ingen anledning för dessa utenheter att ändras. Exekveringsmiljön lämpar sig dock sämre för system där kommunikation med utenheter måste ske av andra skäl än ny indata, vilket gör system med tvåvägskommunikation svårimplementerade. Detta är en svaghet hos systemet. Tvåvägskommunikation kan dock åstadkommas, genom att ansluta en enhet till både en inkanal och en utkanal och skriva styrsystemets program på ett lämpligt sätt, se exempel i figur 5.1, se bilaga B.3 för översättning av exemplet.

```
=> ska_rotera :: (Int).    % Meddelanden från utenheten.
=> motorsensor :: (Int).
<= motor.
<= faktisk_rotation.

rotera(X) :- (X) <- ska_rotera[0:1].
faktiskt(X) :- (X) <- motorsensor[0:1].

?- rotera(Ny_position) => motor.
?- faktiskt(Faktisk_position) => faktisk_rotation.
```

Figur 5.1: Exempel på tvåvägskommunikation.

I exemplet ovan finns en motor kopplad till kanalerna `motor` och `motorsensor`. Till `motor` skrivs den position motorn ska rotera till, och motorn svarar sedan med angivelse om vart den faktiskt roterat i `motorsensor`. Sist vidarebefordras denna information till en annan utkanal, `faktisk_rotation`. Detta exempel är väldigt enkelt, men demonstrerar den allmänna idén om hur tvåvägskommunikation kan upprättas. Hade uppbyggnad istället skett kring en Req/Rep-arkitektur hade kommunikation från utenheter till styrsystem potentiellt blivit enklare, men med nackdelen att alla enheter som annars hade kunnat vara passiva tvingas att regelbundet fråga styrsystemet efter uppdateringar.

5.2 Exekveringsmiljöns begränsningar

Som nämnt i kapitel 4.4 stöttar exekveringsmiljön inte alla delar av Channelog som presenterades i kapitel 4.2, även om översättningsalgoritmer har utarbetats. De huvudsakliga begränsningarna är

1. I den graf som motsvarar en predikatdefinitions kropp måste varje delgraf kunna representeras som ett träd.

2. Varje predikat måste definieras av en och endast en predikatdefinition.

3. Rekursion är inte implementerat.

5.2.1 Krav på trädstruktur

Kravet att kunna representera predikatdefinitioners kroppar som träd kommer från att de översätts till en följd av `JOIN`-klausuler. För att denna följd ska utgöra en korrekt översättning måste – som nämndes i kapitel 4.3.2 – noderna i motsvarande graf besökas i en viss ordning. Vidare får varje nod bara besökas en gång, eftersom varje besök motsvarar en `JOIN`-klausul. Om en nod måste besökas mer än en gång

vid en graftraversering skulle detta innebära en logisk disjunktion – se figur 5.2a – snarare än den önskade logiska konjunktionen, se figur 5.2b.

```
_ :- p(X, Y), q(Y, Z), r(Z, X)
```

```
... FROM p
JOIN q ON p.Y = q.Y
JOIN r ON p.X = r.X
JOIN r ON q.Z = r.X
```

(a) Inkorrekt översättning.

```
... FROM p
JOIN q ON p.Y = q.Y
JOIN r ON p.X = r.X
      AND q.Z = r.Z
```

(b) Korrekt översättning.

Figur 5.2: Problem kring översättning av klausuler med cykler.

Att kräva trädstruktur säkerställer att en korrekt ordning kan erhållas, och innebär att predikatdefinitioner som annars skulle stöta på konjunktionsproblemet inte tillåts. Det är viktigt att notera att denna begränsning inte är en inneboende egenskap hos SQL eller Datalog utan följd av den specifika översättningsalgoritm som används för implementationen av Channelog; predikatdefinitioner vars graf innehåller cykler kan utgöra en väldefinierad fråga.

5.2.2 Översättning av godtycklig struktur

Då kravet på trädstruktur har sitt ursprung i användningen av `JOIN` kan det kringgås genom att välja en annan SQL-struktur. Exemplet i figur 5.3 med kartesisk produkt och `WHERE`-klausul.

```
... FROM p
JOIN q ON c1
JOIN r ON c2
...
```

(a) Nuvarande översättning.

```
... FROM p, q, r ...
    WHERE c1 AND c2 ...
```

(b) Alternativ översättning.

Figur 5.3: Utbyte av `JOIN`.

I översättningen i figur 5.3b ovan kommer `FROM` tolkas som kartesisk produkt av de ingående tabellerna, vars värden sedan filtreras utefter `WHERE`-klausulen. Således erhålles samma beteende som koden i 5.3a, men utan att införa krav på trädstruktur.

Eftersom varje predikatdefinition översätts till en tabell, är predikat som definieras av fler än en predikatdefinition heller inte tillåtna. Detta är enklare att implementera än det förra; en `UNION` kan användas för att sammanfoga de olika definitionerna.

5.2.3 Översättning av rekursiva predikat

Rekursion är en del av Datalog och är därför också en del av den konceptuella modellen för Channelog. Ett exempel med ett rekursivt predikat är figur 5.4 som har det rekursiva predikatet `rutt`. Resultatet blir sant för alla par av punkter som har en rutt med en eller flera vägar mellan varandra. Notera att Channelogs enkla typsystem inte tillåter tupler som en typ; mer självklart vore kanske `väg :: ((Int, Int), (Int, Int))`, men detta tillåter inte Channelog.

```
=> väg :: (Int, Int, Int, Int)

rutt(X1, Y1, X2, Y2) :-
    (X1, Y1, X2, Y2) <- väg.
rutt(X1, Y1, X3, Y3) :-
    (X1, Y1, X2, Y2) <- väg.
    rutt(X2, Y2, X3, Y3).
```

Figur 5.4: Exempel med rekursion.

Det går att skriva en SQL fråga som ger samma resultat som exemplet med rekursion, som i figur 5.5. Eftersom det finns ett rekursivt predikat som har en SQL-tolkning kan det finnas flera, men en allmän algoritm för översättning av rekursiva predikat har inte utvecklats.

```
CREATE TABLE väg (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    A  INTEGER NOT NULL,
    B  INTEGER NOT NULL,
    C  INTEGER NOT NULL,
    D  INTEGER NOT NULL);
CREATE VIEW rutt AS
WITH RECURSIVE rekursiv_rutt AS (
    SELECT * FROM väg
    UNION ALL
    SELECT väg.A AS A, väg.B AS B
         , rekursiv_rutt.C AS C, rekursiv_rutt.D AS D
    FROM väg JOIN rekursiv_rutt
    ON  väg.C = rekursiv_rutt.A
    AND väg.D = rekursiv_rutt.B
)
SELECT * FROM rekursiv_rutt;
```

Figur 5.5: Översättning av rekursionexempel.

Att finna en algoritm för översättning av rekursiva frågor vore ett intressant arbete för framtiden då det skulle möjliggöra att implementationen kan stödja rekursiva predikat. Med rekursiva predikat kan många fler problem lösas än med predikat utan rekursion.

5.2.4 Övriga begränsningar

Utöver de ovanstående begränsningarna finns det begränsningar som kommer från kommunikationsprotokollet beskrivet i kapitel 4.4. Eftersom enbart en byte används för att koda längden på strängar får kanalnamn och strängar som skickas vara högst 255 tecken långa. Om denna gräns överskrids kommer de efterträdande tecknen att ignoreras (ifall ingen vidare data förväntas) eller tolkas som del av nästa datastycke. Ytterligare en restriktion är på heltalen; utanför intervallet $[2^{31}, 2^{31} - 1]$ är inte tillåtna eftersom talen representeras som 32-bitars heltal med tecken.

5.3 Vidare arbete

Som nämnts i 5.1 är varken Pub/Sub- eller Req/Rep-arkitektur ett självklart överlägset val för systemets kommunikation. Uppbyggnaden av systemet i detta projekt har gjorts kring Pub/Sub-arkitektur, men om detta är en optimal lösning tål att undersökas. Utveckling av en implementation byggd på Req/Rep-arkitektur hade möjliggjort jämförelser, och hade kunnat göra avvägningar tydligare. Ett annat alternativ är implementation av ett hybridsystem som utnyttjar någon kombination av dessa kommunikationsarkitekturer; ett sådant system hade potentiellt kunnat kombinera fördelarna från båda.

5.3.1 Språkets portabilitet

Ytterligare en intressant aspekt att undersöka är översättningens portabilitet. Den nuvarande implementationen är anpassad efter SQLite:s dialekt och dess specifika implementation. Det innebär att vissa konstruktioner kan behöva modifieras för att vara kompatibel med andra SQL-dialekter och databashanterare.

Det vore även relevant att undersöka hur systemet kan anpassas till andra datamodeller än den relationsbaserade. En möjlig riktning är att porta systemet till en grafdatabas, där data representeras som noder och relationer istället för tabeller.

5.3.2 Språkutökning

För att skapa mer intressanta system kan utökningar göras till språket, med inspiration av andra språk. Ett exempel på aggregationsfunktioner som skulle kunna läggas till visas i figur 5.6, se B.4 för exempel på översättning. Vidare skulle möjlighet att radera data från inkanaler eventuellt kunna vara fördelaktigt att lägga till, eller aritmetiska operatorer.

```
=> bokningskanal :: (Int, Int, Int).
<= krockkanal.
<= ok_bokningskanal.

bokning(Start, Slut, Id) :-
  (Id, Start, Slut) <- bokningskanal,
  Start < Slut.

krockar(A, B) :- A != B,
  bokning(Start1, Slut1, A),
  bokning(Start2, Slut2, B),
  Start1 < Slut2,
  Start2 < Slut1.

ok_bokning(A) :- 0 = count{ ?- krockar(A, B) }.

?- krockar(A, B)
  => krockkanal.
?- ok_bokning(A)
  => ok_bokningskanal.
```

Figur 5.6: Exempel med en aggregationsfunktion.

Det går också att tänka sig att man kan tillåta att man binder en variabel med likhet vilket skulle möjliggöra att man kan skriva program där en variabel binds till ett värde av en konstant eller ett aggregationsresultat om man utökat språket med aggregationsfunktioner. Denna typ av likhetsanvändning kan jämföras med tilldelning i andra språk. Man skulle då också kunna introducera ett nytt sätt att ange att en variabel ska ha samma värde som en annan utan att behöva ge dem samma namn vilket visas i figur 5.7.

```
p(Y) :- q(X), X = Y.
```

(a) Exempel med likhet.

```
p(X) :- q(X).
```

(b) Ekvivalent exempel utan likhet.

Figur 5.7: Exempel där likhet mellan variabler kan uttryckas genom att använda samma variabelnamn.

5.4 Slutsats

Det färdiga systemet består av ett styrsystem, byggt ovanpå en SQL-databas. Systemet har förmåga till slutledning, samt besvarande av ställda frågor. Det har även möjlighet att kommunicera med andra enheter via sina kanaler.

Genom att i stor utsträckning begränsa koden till standard-SQL blir systemet kompatibelt med många av de vanligaste databashanterare. Den enda anpassning som måste göras är ordnande av tuplerna i inkanalerna samt typen för strängar, då detta bygger på funktionalitet utanför standard-SQL och är specifik för varje databashanterare. Systemets funktionalitet är i princip helt oberoende av vilken databashanterare man använder.

Sammanfattningsvis visar arbetet att en översättning från Datalog till SQL räcker för att uttrycka regler för vissa system, som exempelsystemet beskrivet i kapitel 1.1 och 4.5. Vidare arbete hade kunnat utöka uttrycksfullheten och förenkla hantering av mer avancerade scenarier.

Litteratur

[1] P. KÖRNER m. fl., "Fifty Years of Prolog and Beyond," *Theory and Practice of Logic Programming*, årg. 22, nr 6, s. 776–858, 2022. DOI: 10.1017/S1471068422000102.

[2] V. Dahl, "On database systems development through logic," *ACM Trans. Database Syst.*, årg. 7, nr 1, s. 102–123, mars 1982, ISSN: 0362-5915. DOI: 10.1145/319682.319700. URL: https://doi.org/10.1145/319682.319700.

[3] S. S. Huang, T. J. Green och B. T. Loo, "Datalog and emerging applications: an interactive tutorial," i *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '11, Athens, Greece: Association for Computing Machinery, 2011, s. 1213–1216, ISBN: 9781450306614. DOI: 10.1145/1989323.1989456. URL: https://doi.org/10.1145/1989323.1989456.

[4] A. Horn, "On sentences which are true of direct unions of algebras," *Journal of Symbolic Logic*, årg. 16, nr 1, s. 14–21, 1951. DOI: 10.2307/2268661.

[5] S. R. Buss, "Chapter I - An Introduction to Proof Theory," i *Handbook of Proof Theory*, ser. Studies in Logic and the Foundations of Mathematics, S. R. Buss, utg., vol. 137, Elsevier, 1998, s. 1–78. DOI: https://doi.org/10.1016/S0049-237X(98)80016-5. URL: https://www.sciencedirect.com/science/article/pii/S0049237X98800165.

[6] M. H. Van Emden och R. A. Kowalski, "The Semantics of Predicate Logic as a Programming Language," *J. ACM*, årg. 23, nr 4, s. 733–742, okt. 1976, ISSN: 0004-5411. DOI: 10.1145/321978.321991. URL: https://doi.org/10.1145/321978.321991.

[7] S. Ceri, G. Gottlob och L. Tanca, *Logic Programming and Databases*, G. Schlageter, F. Stetter och E. Goto, utg. Springer-Verlag, 1990, s. 1–11.

[8] S. Ceri, G. Gottlob och L. Tanca, "What you always wanted to know about Datalog (and never dared to ask)," *IEEE Transactions on Knowledge and Data Engineering*, årg. 1, nr 1, s. 146–166, 1989. DOI: 10.1109/69.43410.

[9] MQTT.org. "MQTT: The Standard for IoT Messaging," hämtad 4 maj 2026. URL: https://mqtt.org/.

[10] P. Hintjens, *ZeroMQ*, A. Oram och M. Gulick, utg. O'Reilly Media, Inc., 2013, ISBN: 978-1-449-33406-2.

[11] R. T. Fielding och J. Reschke, *Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing*, RFC 7230, juni 2014. DOI: 10.17487/RFC7230. URL: https://www.rfc-editor.org/info/rfc7230.

[12] M. Forsberg och A. Ranta, ”The Labelled BNF Grammar Formalism,” Department of Computing Science Chalmers University of Technology och the University of Gothenburg, 2005.
[13] A. Abel m. fl. ”The BNF Converter,” hämtad 5 maj 2026. URL: https://bnfc.digitalgrammars.com/.
[14] The ZeroMQ authors. ”ZeroMQ,” hämtad 4 maj 2026. URL: https://zeromq.org/.
[15] T. Wittner. ”zeromq4-haskell: Bindings to ZeroMQ 4.x,” hämtad 4 maj 2026. URL: https://hackage.haskell.org/package/zeromq4-haskell.
[16] SQLite Development Team. ”SQLite Autoincrement,” hämtad 17 maj 2026. URL: https://sqlite.org/autoinc.html.
[17] Python Software Foundation. ”tkinter — Python interface to Tcl/Tk,” hämtad 5 maj 2026. URL: https://docs.python.org/3/library/tkinter.html.
[18] B. E. Granger och M. Ragan-Kelley. ”PyZMQ: Python bindings for ØMQ,” hämtad 5 maj 2026. URL: https://pypi.org/project/pyzmq/.

A

Channelog LBNF

```
{-
Förkortningar som används för labels
B   - Boolesk jämförelseoperator
C   - Klausul
Ch  - Kanal
ChP - KanalPredikat
F   - Faktor
HC  - HornKlausul
Prg - Program
R   - Omfång (Range)
S   - Påstående
SI  - Signerat heltal
T   - Term
TS  - TypSignatur
Ty  - Typ -}

entrypoints Program ;

token Var (upper (letter | digit | '_')*) ;
token Pred (lower (lower | digit | '_')*) ;

-- Heltal med tecken, Integer tolkas som endast positiva heltal
SI_Int. SignedInt ::= Integer ;
SI_Neg. SignedInt ::= "-" Integer ;


-- Typsignaturer för inkanaler
TS_Int. SingleType ::= "Int" ;
TS_Str. SingleType ::= "Str" ;

separator SingleType "," ;

Ty. Type ::= "(" [SingleType] ")" ;
```

```
-- Booleska jämförelseoperatorer
B_Ls.     BoolOp ::= "<"  ;
B_Gr.     BoolOp ::= ">"  ;
B_Eq.     BoolOp ::= "="  ;
B_LsOrEq. BoolOp ::= "<=" ;
B_GrOrEq. BoolOp ::= ">=" ;
B_NoEq.   BoolOp ::= "!=" ;

-- Kanaler
Ch_In.  ChannelIn ::= "=>" Pred ;
Ch_InT. ChannelIn ::= "=>" Pred "::" Type ;

Ch_Out. ChannelOut ::= "<=" Pred ;

TS. TermSource ::= Pred "[" SignedInt ":" SignedInt "]" ;

-- Liknande definitioner till Datalog
T_Var. Term ::= Var ;
T_Int. Term ::= SignedInt ;
T_Str. Term ::= String ;

separator Term "," ;

F_Atom.    Factor ::= Pred "(" [Term] ")" ;
F_Channel. Factor ::= "(" [Term] ")" "<-" TermSource ; -- kanaluppack
F_BoolOp.  Factor ::= Term BoolOp Term ;

separator Factor "," ;

C. Clause ::= [Factor] ;

HC_Rule. HornClause ::= Factor ":-" Clause ;
HC_Goal. HornClause ::= "?-" Factor "=>" Pred ;


-- Statements
S_ChR. Statement ::= ChannelIn ;
S_ChL. Statement ::= ChannelOut ;
S_HC.  Statement ::= HornClause ;

terminator Statement "." ;

Prg. Program ::= [Statement] ;

comment "%"
```

B

Översättningar av exempel

Nedan presenteras SQL-översättningar av Channelogprogram från exempel i rapporten. Namnen i översättningarna stämmer inte överens med de som skulle fås från exekveringsmiljön; de som används här har valts för att ge läsligare kod. `SELECT`-satser från översättning av utkanaler utelämnas.

B.1 Översättning av exempel i figur 4.1

```
CREATE TABLE ljussensor (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    A  INTEGER NOT NULL);

CREATE VIEW ljussensor_intervall AS
SELECT A FROM ljussensor
WHERE id <= ( SELECT max ( id ) FROM ljussensor ) - 0
  AND id >  ( SELECT max ( id ) FROM ljussensor ) - 1;

CREATE VIEW lampa_tänd AS
SELECT A FROM ljussensor_intervall
WHERE A < 4;

CREATE VIEW mål AS
SELECT * FROM lampa_tänd;
```

B.2 Översättning av exempel i figur 4.2

```
CREATE TABLE reglage (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    A  INTEGER NOT NULL);

CREATE TABLE termometer (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    A  INTEGER NOT NULL);

CREATE VIEW reglage_intervall AS
SELECT A FROM reglage
WHERE id <= ( SELECT max ( id ) FROM reglage ) - 0
  AND id >  ( SELECT max ( id ) FROM reglage ) - 1;

CREATE VIEW termometer_intervall AS
SELECT A FROM termometer
WHERE id <= ( SELECT max ( id ) FROM termometer ) - 0
  AND id >  ( SELECT max ( id ) FROM termometer ) - 1;

CREATE VIEW element_ny_temp AS
SELECT reglage_intervall.A AS A
FROM reglage_intervall, termometer_intervall
WHERE termometer_intervall.A < reglage_intervall.A;

CREATE VIEW element_lampa_på AS
SELECT A FROM element_ny_temp;

CREATE VIEW mål_element_ny_temp AS
SELECT * FROM element_ny_temp;

CREATE VIEW mål_element_lampa_på AS
SELECT * FROM element_lampa_på;
```

B.3 Översättning av exempel i figur 5.1

```
CREATE TABLE ska_rotera (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    A  INTEGER NOT NULL);

CREATE TABLE motorsensor (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    A  INTEGER NOT NULL);

CREATE VIEW ska_rotera_intervall AS
SELECT A FROM ska_rotera
WHERE id <= ( SELECT max ( id ) FROM ska_rotera ) - 0
  AND id >  ( SELECT max ( id ) FROM ska_rotera ) - 1;

CREATE VIEW motorsensor_intervall AS
SELECT A FROM motorsensor
WHERE id <= ( SELECT max ( id ) FROM motorsensor ) - 0
  AND id >  ( SELECT max ( id ) FROM motorsensor ) - 1;

CREATE VIEW rotera AS
SELECT A FROM ska_rotera_intervall;

CREATE VIEW faktistkt AS
SELECT A FROM motorsensor_intervall;

CREATE VIEW mål_rotera AS
SELECT * FROM rotera;

CREATE VIEW mål_faktiskt AS
SELECT * FROM faktiskt;
```

B.4 Översättning av exempel i figur 5.6

Eftersom det inte har utvecklats en generell metod att översätta aggregationsfunktioner från Channelog till SQL presenteras här två manuellt skapade översättningar av exemplet.

```
CREATE TABLE bokningskanal (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    A  INTEGER NOT NULL,
    B  INTEGER NOT NULL,
    C  INTEGER NOT NULL);

CREATE VIEW bokning AS
SELECT bokningskanal.B AS A,
       bokningskanal.C AS B,
       bokningskanal.A AS C
FROM bokningskanal
WHERE bokningskanal.B < bokningskanal.C;

CREATE VIEW krockar AS
SELECT bokning_A.C AS A, bokning_B.C AS B
FROM bokning AS bokning_A, bokning AS bokning_B
WHERE bokning_A.C <> bokning_B.C
  AND bokning_A.A <  bokning_B.B
  AND bokning_B.A <  bokning_A.B;

-- Exempelöversättning 1
CREATE VIEW ok_bokning AS
WITH krockantal AS (
    SELECT bokning.C AS A, COUNT(krockar.B) AS antal
    FROM bokning
    LEFT JOIN krockar ON bokning.C = krockar.A
    GROUP BY bokning.C
)
SELECT A
FROM krockantal
WHERE antal = 0;

-- Exempelöversättning 2
CREATE VIEW ok_bokning AS
SELECT bokning.C AS A
FROM bokning
WHERE (
    SELECT COUNT(*)
    FROM krockar
```

```
    WHERE krockar.A = bokning.C
) = 0;

CREATE VIEW mål_krockar AS
SELECT * FROM krockar;

CREATE VIEW mål_ok_bokning AS
SELECT * FROM ok_bokning;
```

B.5 Genererad översättning av målexemplet från kapitel 1.1 och 4.5

```
/* bokning(A,B,C,D) */
CREATE VIEW bokning AS
    SELECT DISTINCT PA.A AS A,
                    PA.B AS B,
                    PA.C AS C,
                    PA.D AS D FROM bokningskanal PA
    WHERE PA.B<PA.C;

/* aktiv_bokning(A,B,C,D) */
CREATE VIEW aktiv_bokning AS
    SELECT DISTINCT PA.A AS A,
                    PB.B AS B,
                    PB.C AS C,
                    PB.D AS D FROM tid_nu PA
    JOIN bokning PB ON PA.A=PB.A
        WHERE PB.B<=PA.B AND PB.C>PA.B;

/* krockar(A,B,C) */
CREATE VIEW krockar AS
    SELECT DISTINCT PA.A AS A,
                    PA.D AS B,
                    PB.D AS C FROM bokning PA
    JOIN bokning PB ON PA.A=PB.A
        WHERE PA.B<PB.C
              AND PB.B<PA.C
              AND PA.D<>PB.D;

/* Goal_R_krockar_V_Dag_V_A_V_B_(A,B,C) */
CREATE VIEW Goal_R_krockar_V_Dag_V_A_V_B_ AS
    SELECT * FROM krockar;

/* Goal_R_aktiv_bokning_V_Dag_V_Start_V_Slut_V_Namn_(A,B,C,D) */
CREATE VIEW Goal_R_aktiv_bokning_V_Dag_V_Start_V_Slut_V_Namn_ AS
    SELECT * FROM aktiv_bokning;

CREATE TABLE bokningskanal (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    A INTEGER NOT NULL,
    B INTEGER NOT NULL,
    C INTEGER NOT NULL,
    D TEXT NOT NULL
);
```

```
CREATE TABLE sqlite_sequence (
    name,
    seq
);

CREATE TABLE tid_nu (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    A INTEGER NOT NULL,B INTEGER NOT NULL
);
```

B.6 Översättning av målsystem

```
/* bokning(A,B,C,D) */
CREATE VIEW bokning AS
    SELECT DISTINCT PA.A AS A,
                    PA.B AS B,
                    PA.C AS C,
                    PA.D AS D FROM bokningskanal PA
    WHERE PA.B<PA.C;

/* aktiv_bokning(A,B,C,D) */
CREATE VIEW aktiv_bokning AS
    SELECT DISTINCT PA.A AS A,
                    PB.B AS B,
                    PB.C AS C,
                    PB.D AS D FROM tid_nu PA
    JOIN bokning PB ON PA.A=PB.A
        WHERE PB.B<=PA.B AND PB.C>PA.B;

/* krockar(A,B,C) */
CREATE VIEW krockar AS
    SELECT DISTINCT PA.A AS A,
                    PA.D AS B,
                    PB.D AS C FROM bokning PA
    JOIN bokning PB ON PA.A=PB.A
        WHERE PA.B<PB.C
              AND PB.B<PA.C
              AND PA.D<>PB.D;

/* Goal_R_krockar_V_Dag_V_A_V_B_(A,B,C) */
CREATE VIEW Goal_R_krockar_V_Dag_V_A_V_B_ AS
    SELECT * FROM krockar;

/* Goal_R_aktiv_bokning_V_Dag_V_Start_V_Slut_V_Namn_(A,B,C,D) */
```

```
CREATE VIEW Goal_R_aktiv_bokning_V_Dag_V_Start_V_Slut_V_Namn_ AS
    SELECT * FROM aktiv_bokning;

CREATE TABLE bokningskanal (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    A INTEGER NOT NULL,
    B INTEGER NOT NULL,
    C INTEGER NOT NULL,
    D TEXT NOT NULL
);

CREATE TABLE sqlite_sequence (
    name,
    seq
);

CREATE TABLE tid_nu (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    A INTEGER NOT NULL,B INTEGER NOT NULL
);
```