LARS M GUSTAFSSON

Händelsestyrd textgenerering

Abstract

This paper describes the system RADAR, an event-driven text generator. The system reads input from a radar-system i.e. *time,id,position*, and generates comments in Swedish about the objects on the screen. The problem as such is quite easy to grasp and is not discussed much, but the techniques that have been utilized are described in more detail. The system is written in an object-oriented environment, implemented as a meta-interpreter in Prolog. Another technique that plays a major role is Data Driven Execution, this is also implemented on top of a Prolog-system. The source code for the entire system is available in C-Prolog and fully portable. The system makes an object-instance for every new physical object on the Radar-screen and lets the objects themselves generate comments about their situation. The Data Driven Execution rules generate comments on the simplest level, i.e. X appears, Y disappears. Other rules try to combine these simple observations with the comments generated by the objects themselves to more complex phrases and sentences.

1 Inledning

Detta papper beskriver uppbyggnaden av programmet RADAR, som är implementerat helt i C-Prolog. En ursprungsversionen till programmet skrevs på Inst. för Allmän Språkvetenskap vid Lunds Universitet. Den nuvarande versionen av systemet är utvecklat i huvudsak vid Carnegie-Mellon University i Pittsburgh och CoTech AB i Lund. Det problem som studerats är att utifrån informationen från en övervakningsradar generera kommentarer i realtid. Som framgår av titeln så är det beskrivna systemet baserat på att textgenereringen sker fortlöpande. Det finns alltså ingen möjlighet att planera längre sammanhängande yttranden där hänsyn tas till senare händelser. Problemet som sådant är ganska lättfattligt och inte mycket att orda om. Däremot så kommer dom tekniker som använts att beskrivas mer i detalj och deras förtjänster vid denna typ av textgenerering kommer förhoppningsvis att framgå.

Proceedings of NODALIDA 1989, pages 30-40

2 Använda tekniker

Det är i första hand två metodiker jag använt mig av.

1. OOP–Object Oriented Programming
2. DDE–Data Driven Execution

Jag kommer först att beskriva (motiveringen för och implementeringen av) objektorienteringen. Därefter kommer jag att beskriva den datadrivna exekveringen, för att till sist komma in på hur dessa båda tekniker kan knytas samman till ett system.

3 Varför objektorientering?

1. Det är i det här fallet naturligt med ett objekttänkande eftersom det finns en naturlig korrespondens mellan objekten i yttervärlden och deras representation i en objektorienterad programmeringsmiljö.
2. Modulariteten ligger på klassnivån (konceptnivån), där man i klassbeskrivningen anger de variabler och metoder som tillsammans med ärvda variabler och metoder utgör objektbeskrivningen.
3. Sen bindning, dvs. dynamisk allokering av nya objektinstanser under exekveringen. Men även möjligheten att fortlöpande lägga till funktioner och datatyper som inte kunnat förutses vid den ursprungliga programkonstruktionen.
4. Ärvning, detta är visserligen inte nödvändigt för OOP, men väldigt naturligt och arbetsbesparande. Ärvning skapar också en genomgående konsistens i programmet. Detta kan naturligtvis (som traditionellt) till en viss nivå uppnås genom en hård disciplin hos programmeraren, men det är vare sig önskvärt eller effektivt.

Problemets realtidskaraktär gör att multipel ärvning är av stor nytta. Man kan då specificera objektrepresentationen mer i detalj efterhand som informationen kommer in. Tex. kan man låta ett okänt objekt först ärva egenskaper från klassen Flygplan. Därefter när man fått nya indikationer (fart, höjd osv.) så kan man precisera objektet genom ärvning från klassen Jetplan. Denna precisering kan fortgå under hela objektinstansens livslängd genom ytterligare observationer, tex. visuella rapporter. Möjlighet finns även att ta bort ärvningar som visat sig bero på felaktiga informationer.

4 Implementationer av OOP

Smalltalk-80
Simula
Ada
C++
Objective-C
Scheme
Flavours
Prolog meta-interpretator

Detta är några av dom möjligheter som finns för den som vill använda sig av objektorienterad programmering. De olika språken har naturligtvis sina fördelar respektive nackdelar, ADA tex. saknar en naturlig ärvningsmekanism. Smalltalk-80 är den mest renodlade implementationen och tillåter inget annat än objekt-programmering, vilket försvårar kontruktion av hybridsystem.

För min implementation så har jag skrivit en objektmekanism som en metainterpretator i Prolog. Detta gör det möjligt att konstruera hybridsystem där vissa delar är objektorienterade medan andra delar av programmet utnyttjar andra problemlösningsparadigmer.

5 OOP i Prolog

Objekten representeras som enhetsklausuler i Prolog med metoderna och variablerna i en lista.

```
object(name,[metod_1, metod_2,..., metod_n]).
```

Den hierarkiska informationen lagras i en separat klausul.

```
isa( Obj, Obj_Super ).
```

Möjligheter finns också att definiera andra typer av relationer, tex.

```
partof( Obj, Obj_O ).
```

All kommunikation mellan objekten sker med predikatet **send**, tex.

`send(Name, show).`	Metoden **show** gör att objektet ritar ut sig på skärmen.
`send(Name, alert(A)).`	Returnerar värdet på alert i A.
`send(plane, create_instance(Name)).`	Skapar en instans av klassen plane, ett unikt namn (id) returneras i Name ; om Name är instansierat så blir istället detta objektets namn.
`send (Name, set(description, viggen_1).`	
`send(Name, kill(description).`	Sätter/tar bort variabler.

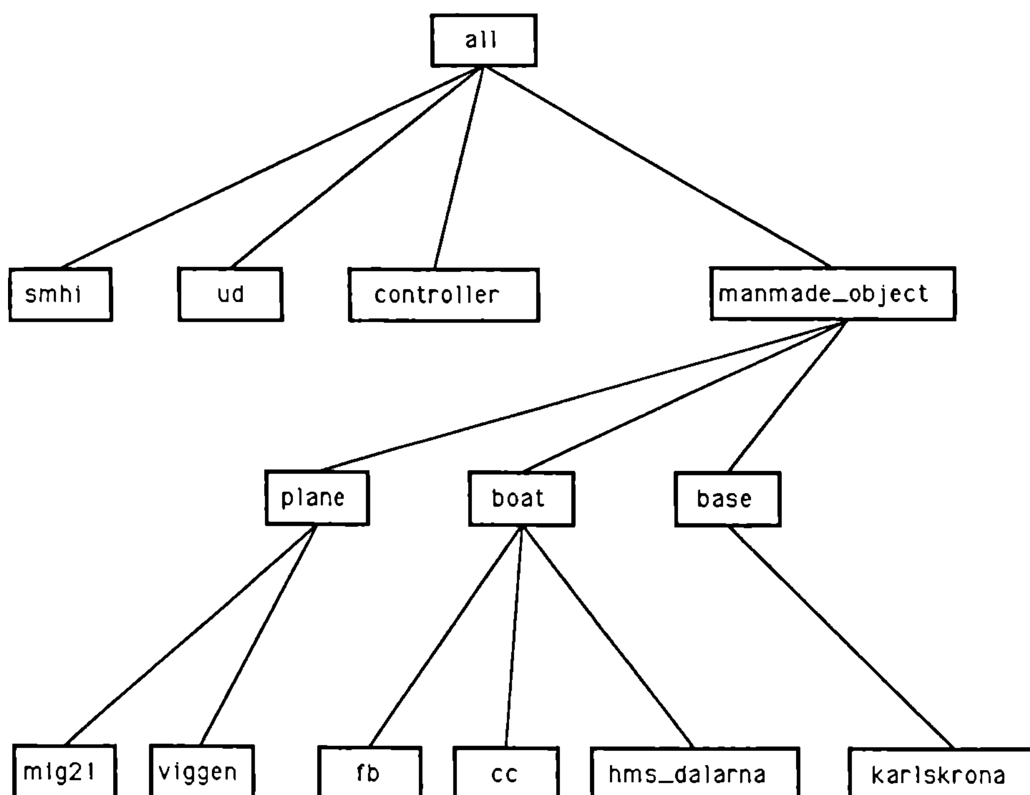

Figure 1: Object hierarchy in Radar

6 Objekthierarkien

Detta är en förminskad version av den objekthierari som jag använt mig av i systemet. *Controller* är huvudobjektet som kontrollerar resten av systemet och innehåller metoder för anpassning till mekanismen för mönsterdriven exekvering som kommer att beskrivas senare. Andra exempel på objekt är *smhi*, som anropas för väderinformation och ud som hanterar information om det politiska läget. Alla relationerna i denna hierarki är av 'isa'-typ, alla löven är alltså objektinstanser.

7 Metoder

Här är några av dom metoder som objekten i programmet är utrustade med.

Metoderna i objektet 'all' ärvs av alla andra objekt i systemet. Dessa metoder skulle visserligen kunna vara inbyggda systemfunktioner men på detta sätt blir systemet 'renare' och man kan även modifiera dessa metoder om så skulle önskas. Metoden internals hos klassen manmade_objects finns hos alla objekt i systemet. Det är denna metod som ger objekten möjlighet att själv generera kommentar-

Objects	Methods
all	create_instance
	set(NewMethod)
	kill(Method)
smhi	visibility(Visi)
ud	alert(Alert)
controller	start
manmade_object	show
	see(List, Sdist)
	firing_range(Range)
	country(Country)
	internals

Figure 2: Methods in the classes of Radar I

Objects	Methods
plane	moving_dimensions(3)
boat	moving_dimensions(2)
base	moving_dimensions(0)
	internals*
	show*

* Overides the inherited methods.

Figure 3: Methods in the classes of Radar II

er utifrån varje objekts egna speciella förutsättningar. Metoden internals finns alltså hos alla objekt, men funktionen varierar beroende på objektets typ.

8 Data Driven Exekvering

Grundprincipen med Data Driven Exekvering är att man har en gemensam dataarea —"blackboard", där alla fakta lagras. Runt denna area ligger ett antal regler som kan påverka innehållet i dataarean om vissa triggvilkor är uppfyllda. Dessa regler kan triggas av "IF-ADDED" eller "IF-ERASED" vilkor. Dom väntar alltså på att ett visst mönster av information skall dyka upp eller försvinna och på så sätt trigga actiondelen av regeln. Dessa regler skulle direkt kunna överföras till en ren parallell maskinarkitektur eftersom det inte finns någon speciell ordningsföljd mellan reglerna.

Detta gör att man kan trigga outputregeln (Rule_5) vid vissa tidpunkter så att systemet genererar "den bästa" output det hunnit skapa fram till denna tidpunkt. Vid komplexa scenarios kan detta vara nödvändigt för att systemet skall kunna jobba i realtid.

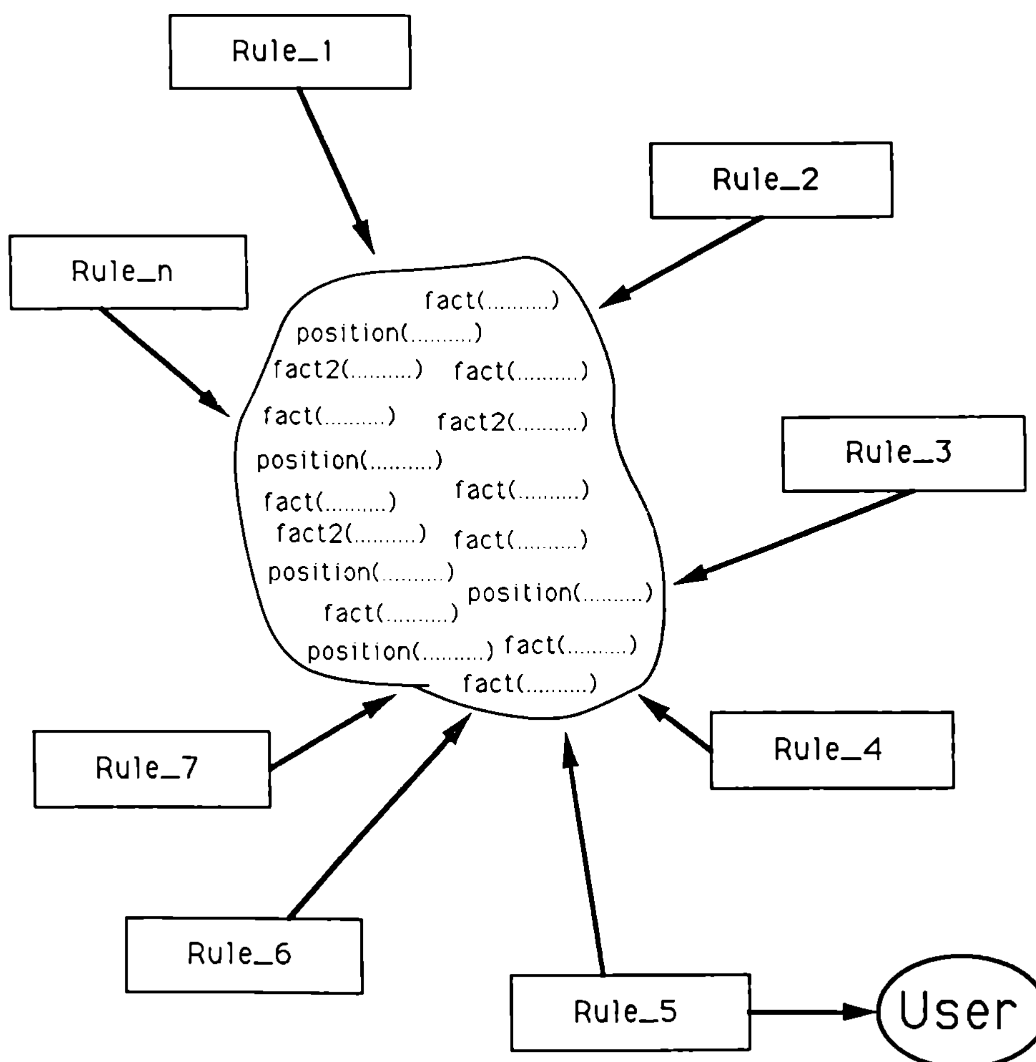

Figure 4: The structure of the pattern-matcher

```
% Rule_1
[condition_1, condition_2,...,condition_n]
     --->
     [action_1,action_2,...,action_n]
-
-
% Rule_n
[condition_1, condition_2,...,condition_n]
     --->
     [action_1,action_2,...,action_n]
%End Rule
[]   ---> [stop].

All rules are tried starting from the top,
whenever a rule triggers, the matcher restarts
from the first rule.

All rules are tried until no more matches
are possible (and the matcher reaches the
End Rule).
```

Figure 5: Rule format in pattern-matcher

9 Regelformat

I figur 5 vises regelformatet där "condition" kan vara förekomsten eller frånvaron av ett visst mönster i dataarean. Action är de operationer som utförs då regeln triggas. Dessa operationer påverkar innehållet i dataarean så att ett nytt tillstånd uppstår som eventuellt kan trigga nya regler. Principen är alltså att man lägger in fakta om objekten på den lägsta nivån (fysisk position) och startar därefter matchningsmekanismen och låter den stegvis bygga upp mer komplexa uttryck ända upp till det slutliga yttrandet.

10 Generering av enkla observationer

Genereringen av enkla observationer kan ske på tre olika sätt.

1. Generering utifrån matchning från position-nivån.
2. Generering av en instans av manmade_object via metoden internals.
3. Generering av ett objekt av klassen "base" som har en speciell variant av internals som kontrollerar alla objekt som närmar sig och passerar osv.

Om man efter programkonstruktionen vill lägga till ett objekt som ställer speciella krav på på att nya kommentarer genereras, så är det bara att se till

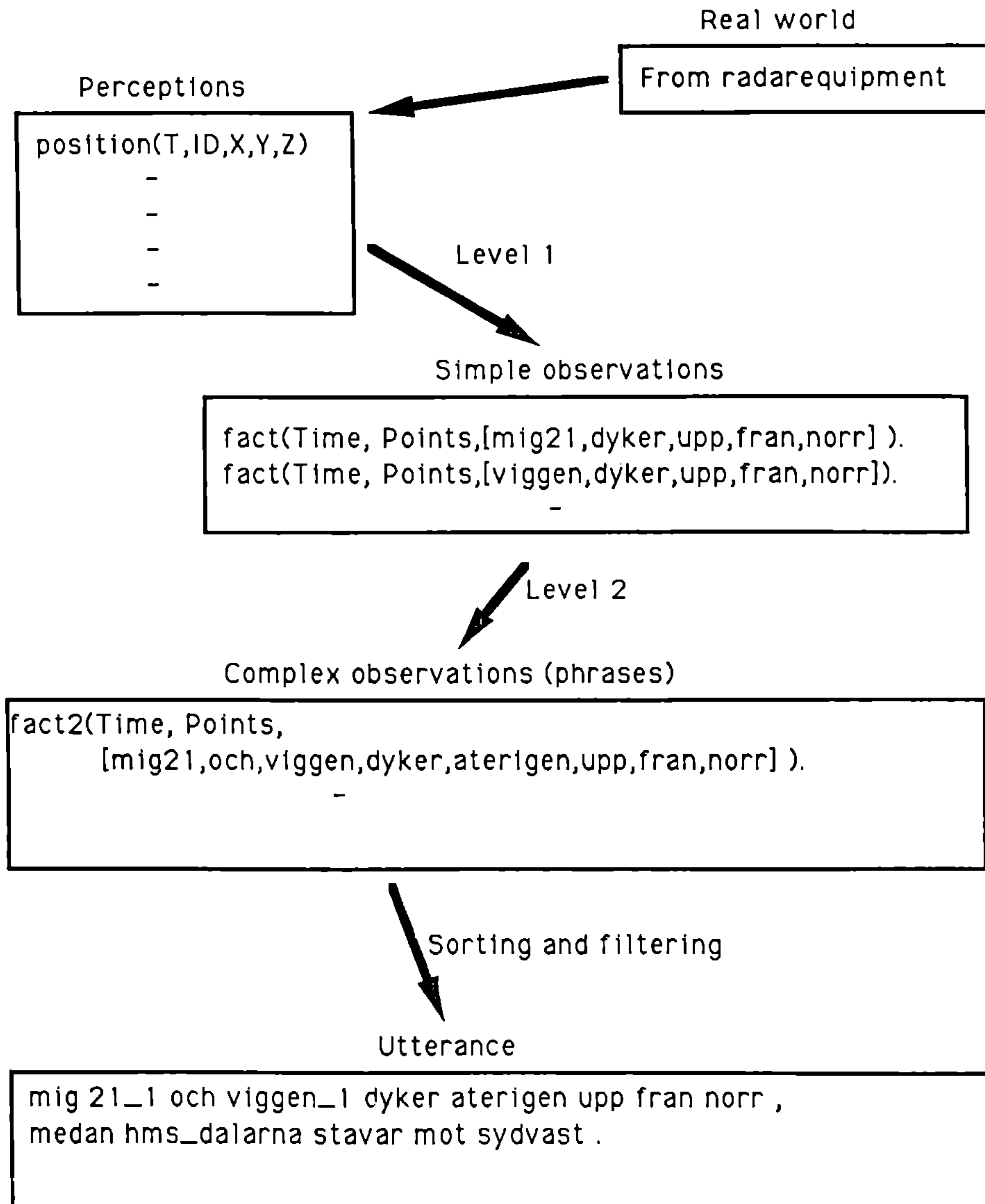

Figure 6: Match levels in Radar

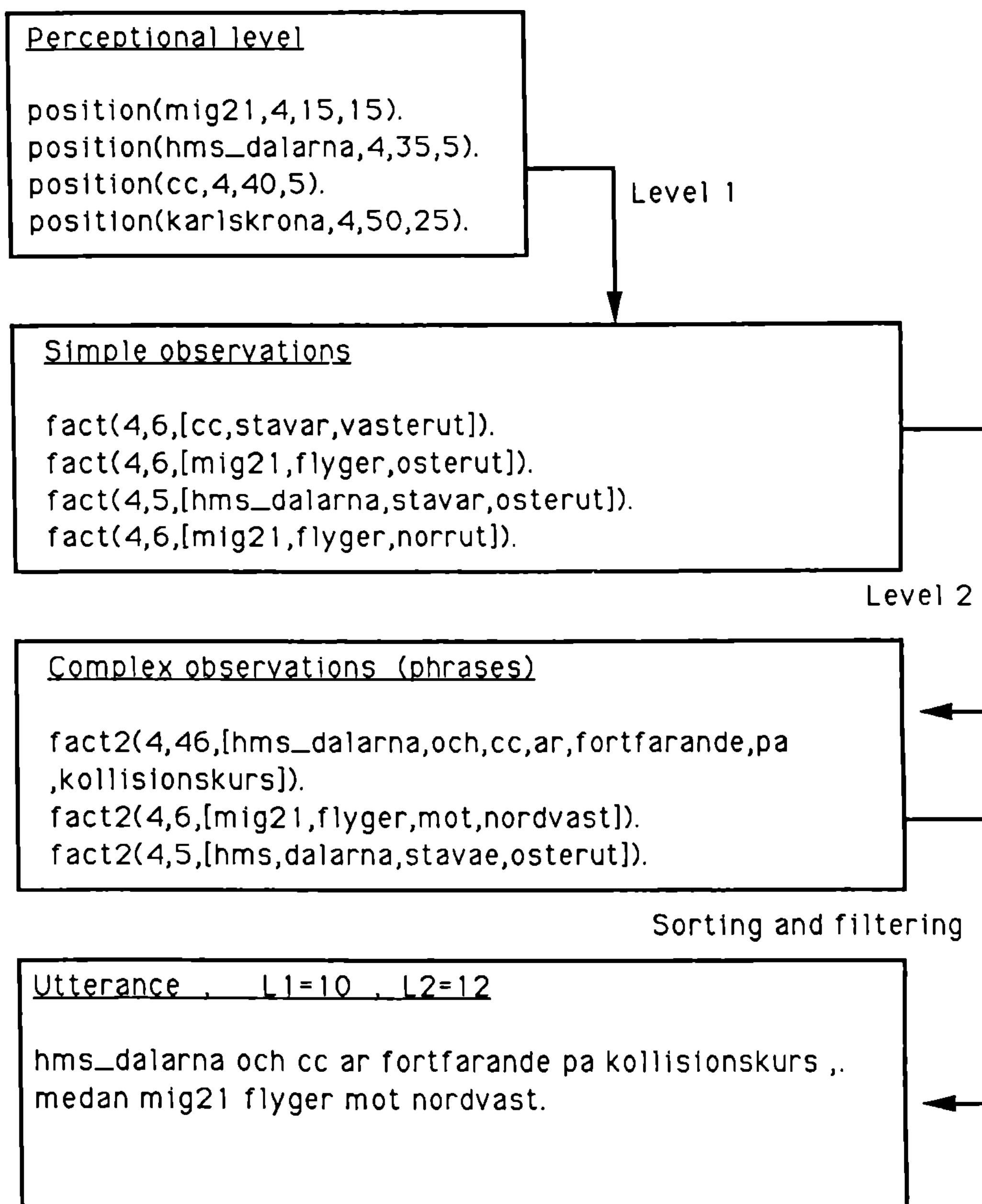

Figure 7: Sample stages in the matching process

att dessa genereras av objektet själv i metoden internals. Det övriga systemet behöver alltså inte modifieras.

11 Faktanivåer i systemet

Systemet utgår från informationen från en övervakningsradar, bestående av *tid,id,position*. Från denna information finns det ett antal regler som detekterar uppdykande och försvinnande objekt. Vidare finns det på denna nivå andra regler som detekterar kurs och hastighet osv. Dessa regler skapar dom enkla observationer som syns i figur 6 på nivån "simple-observations".

Där variabeln "Points" indikerar hur intressant denna information bedöms vara. Detta räknas ut ifrån objektets egenpoäng, händelsens poäng, nationalitet, beredskapsgrad osv. Dessa enkla observationer kombineras sedan ihop av en annan uppsättning regler till kompletta fraser. Poängen från den nedersta nivån förs upp till överliggande nivå genom att man tar hänsyn till bl.a. dom ingående objektens poäng, händelsens poäng samt eventuella samordningspoäng. Utifrån dessa fraser bildas sedan ett yttrande mha. ett antal sorterings och filtreringsregler. De egenskaper hos objekten som behövs vid matchningen fås naturligtvis

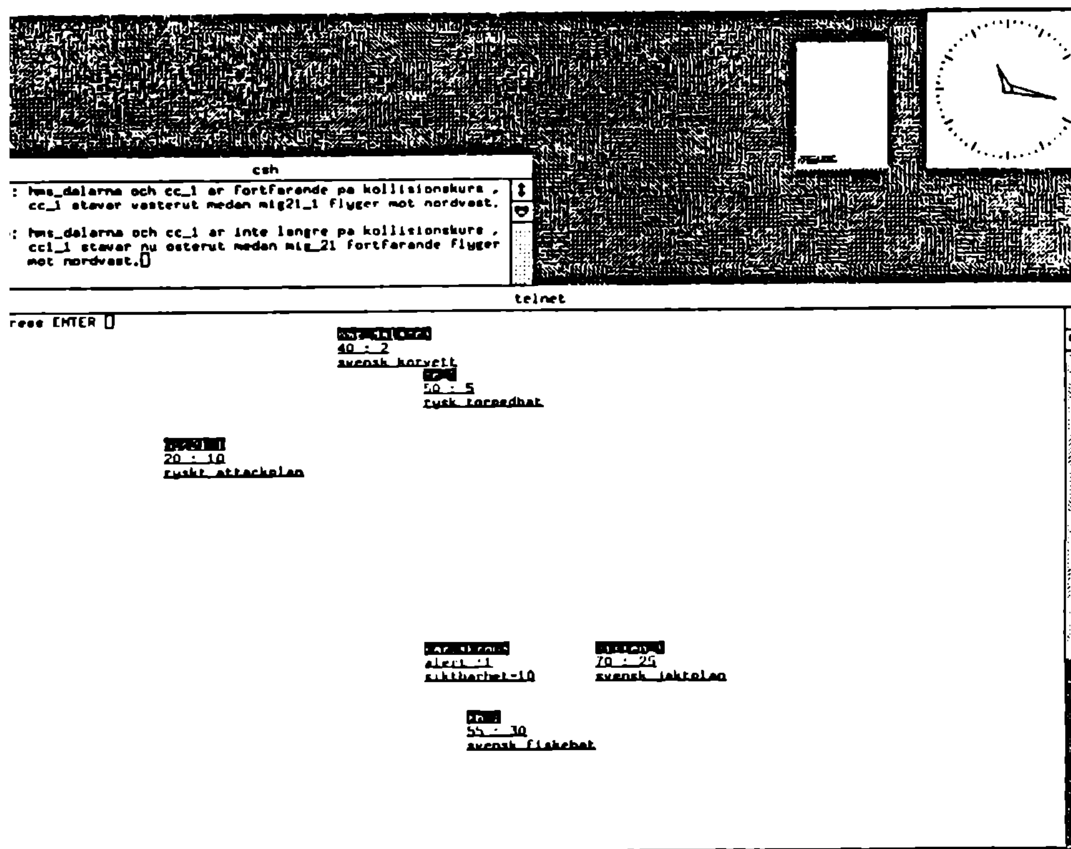

Figure 8: Interacting with Radar

genom att man via "send" till respektive objekt frågar om den önskade egenskapen.

I figur 7 ser man hur hela genereringsprocessen går till vid en viss tidpunkt.

12 Systemets användargränssnitt

I figur 8 ser man ett exempel på hur bildskärmen kan se ut vid en given tidpunkt i prototypsystemet.

Litteratur

Cox, Brad J. 1986. *Object Oriented Programming, An Evolutionary Approach.* Addison-Wesley, Reading, Massachusetts.

Covington, Michael A., Donald Nute, Andre Vellino 1988. *Prolog Programming in Depth.* Scott, Foresman and Co., London.

Bratko, Ivan. 1986. *Prolog Programming for Artificial Intelligence.* Addison-Wesley, Wokingham.

Clocksin, William F., Christopher S. Mellish. 1981. *Programming in Prolog.* Springer, Heidelberg.

Goldberg, and Robson. 1983. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, Reading, Massachusetts.

Fornell, Jan. Sigurd, Bengt 1983. Commentator. Praktisk Lingvistik, 8. Lund.

Cognitive Technology AB
Box 1691
221 01 LUND
lmg@hum.gu.se