CODEN: LUTFD2/(TFRT-5456)/1–85/(1992)

Interaktiv grafisk editering och simulering av Grafcet

Sven Jonasson

Institutionen för Reglerteknik
Lunds Tekniska Högskola
Mars 1992

Department of Automatic Control
Lund Institute of Technology
P.O. Box 118
S-221 00 Lund Sweden

Document name
MASTER THESIS

Date of issue
March 1992

Document Number
CODEN: LUTFD2/(TFRT–5456)/1-85/(1992)

Author(s)
Sven Jonasson

Supervisor
Anders Blomdell, Dag Brück, Rolf Johansson

Sponsoring organisation

Title and subtitle
Interactive Graphical Editing And Simulation Of Grafcet

Abstract

Grafcet is a function chart standard that is used to describe the function and behaviour of control systems by means of a graphical representation and short describing texts. This master thesis considers implementation of a graphical environment that allows realization of a control system described by Grafcet.

Key words
C++, Grafcet, InterViews

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title

ISBN

Language
Swedish

Number of pages
85

Recipient's notes

Security classification

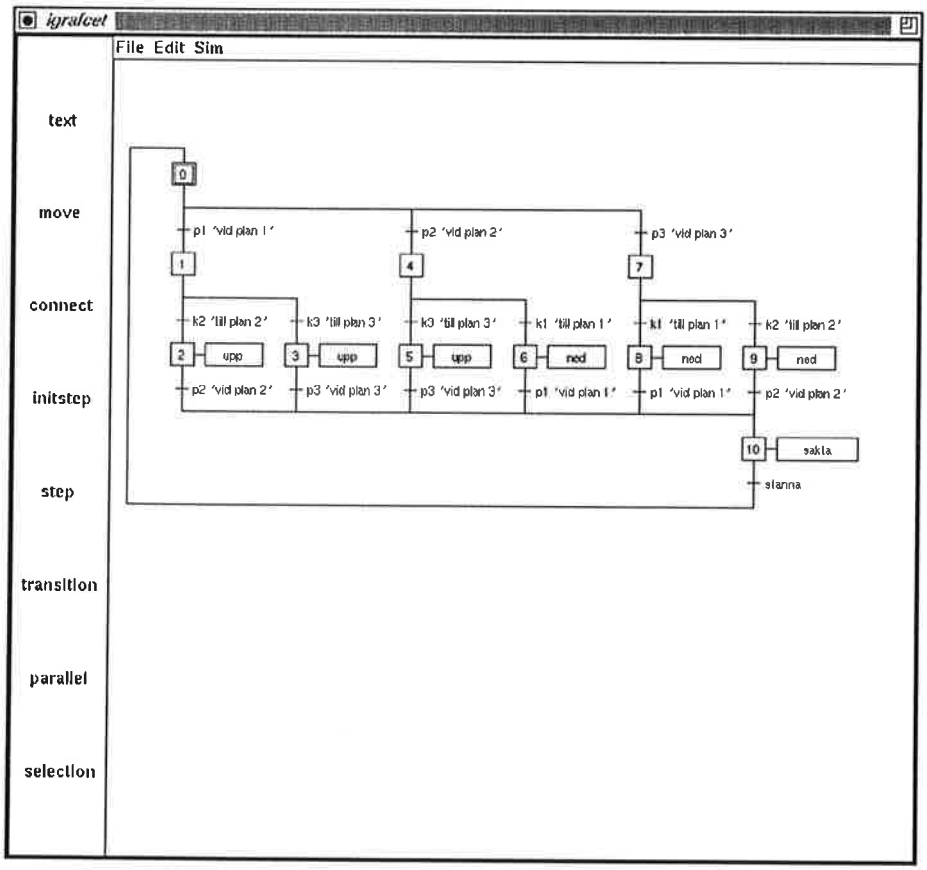
Innehållsförteckning

1. Inledning

Grafcet (Bayard, 1989; IEC, 1988) är en funktionsdiagramstandard som används för att dokumentera funktionsbeskrivningar med enkla symboler och kortfattad beskrivande text. Standarden är ett neutralt beskrivningssätt som är lätt att förstå för de olika yrkesgrupper som är involverade i framtagningen av styrsystem.

Ett styrsystem med ovanstående funktionsbeskrivning konstrueras oftast med PLC-teknik (Pragrammable Logic Control), men eftersom all information anges vid beskrivningen av ett funktionsdiagram kan det tyckas vara dubbelarbete att ange samma information igen vid PLC-programmering.

Det här examensarbetet går ut på att konstruera en grafisk editor för Grafcet funktionsdiagram samt ett styrsystem som realiseras direkt ur den text och den grafiska information som användaren ger i sitt funktionsdiagram. Funktionsdiagrammet är med andra ord inte bara till för dokumentation som i fallet då styrsystem realiseras med PLC-teknik, utan också för att generera ett styrsystem. Vid editeringen tillhandahålls symboler enligt Grafcet standarden, och användaren tillåts bara förbinda dem enligt standarden.



Figur 1.1 Användargränssnittet bör inte skapa några problem för den som arbetat i interaktiva miljöer tidigare. Här visas ett Grafcet funktionsdiagram för styrfunktionen av en hiss.

Användarmiljön enligt figur 1.1 är konstruerad så att alla som har använt ett ritprogram ska känna igen sig. I figuren visas ett exempel på ett funktionsdiagram för styrfunktionen av en hiss. Funktionsdiagrammet, som är skapat i programmet, innehåller förutom steg och övergångar även alternativa vägar.

I programmet, som jag kallar igrafcet, kan ett eller flera funktionsdiagram skapas. Symboler väljs i paletten till vänster och placeras ut på den grafiska ytan. De förbinds med varandra och text knyts till steg och övergångar för att ange övergångsvillkor och order. Flera funktionsdiagram kan skapas på samma rityta och de fungerar oberoende av varandra om så önskas, men de kan även synkroniseras med varandra. Om flera funktionsdiagram skapas på samma bild kommer de att representeras tillsammans i den underliggande strukturen.

För att konstruera den grafiska editorn och styrsystemet har jag använt programspråket C++ (Stroustrup, 1991) och klassbiblioteket InterViews (Linton et al., 1989). InterViews är ett bibliotek för grafik och användarkommunikation, det är skrivet i C++ och arbetar under fönsterhanteringsystemet X. Dialogboxar, knappar och menyer etc är typiska klasser som finns tillgängliga i InterViews för att kunna bygga gränssnitt.

Jag har även tittat närmare på teorin för Petrinät (Murata, 1989). Petrinät är släkt med Grafcet funktionsdiagram och jag har fått en del ideer därifrån som utnyttjas för att skapa styrsystemet.

I kapitel 2, 3 och 4 beskrivs programmets strukturer, samt hur de olika klassbiblioteken är uppbyggda och knutna till varandra. Kapitel 5 är en funktionsbeskrivning som anger hur programmet används.

Rapporten förklarar inte Grafcet, InterViews eller C++ så den som önskar bättra på sina kunnskaper på de områdena hänvisas till litteratur i kapitel 7.

2. Programstrukturer

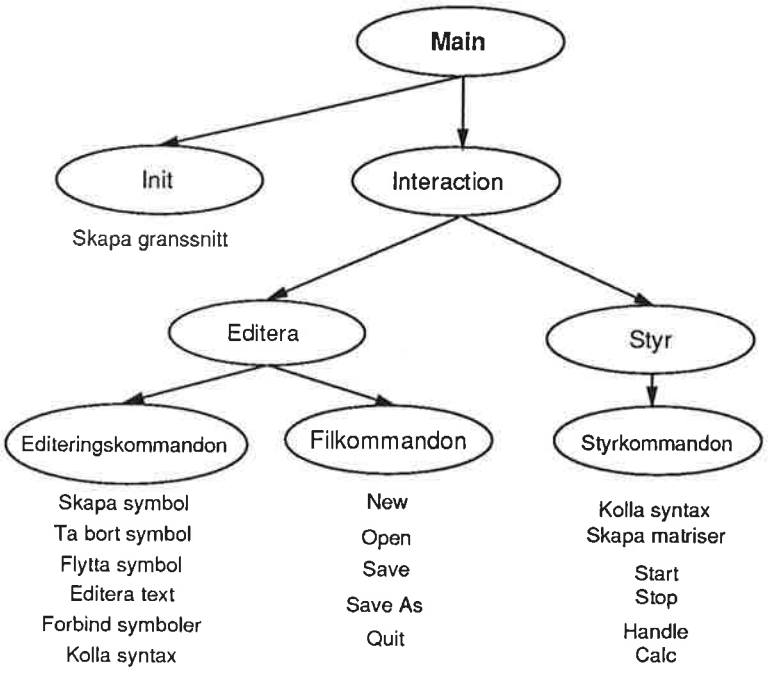
2.1 Strukturering av problemet

För att förklara hur programmet fungerar och är uppbyggt kommer jag att resonera med utgångspunkt från olika figurer. När man programmerar objektorienterat går arbetet ut på att deklarera och definiera klasser, bygga klassbibliotek, och då är det lämpligt att hitta naturligt begränsade uppgifter hos programmet, strukturera, och skapa klasser av dem. När man skapar en klass försöker man hitta en tidigare definierad klass med de grundläggande egenskaper som man är intresserad av, i den nya klassen lägger man sedan till de funktioner och datatyper man behöver för att ge den nya klassen rätt egenskaper. Man deklarerar eventuellt om en del funktioner hos basklassen så att de passar den nya klassen.

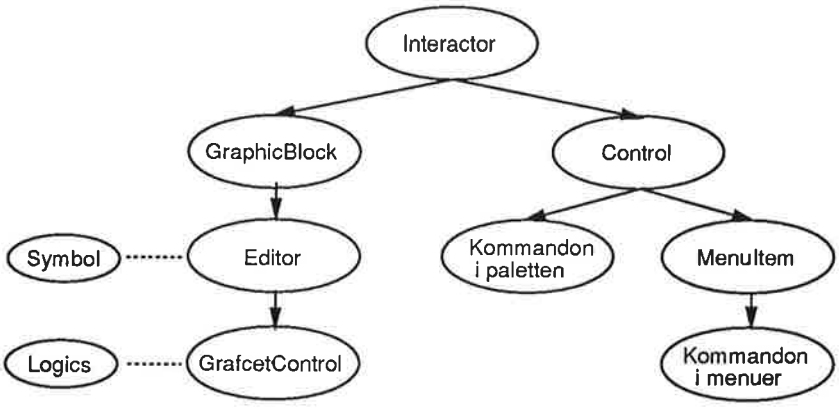
2.2 Interaktion

Editering och styrning

Programmet är till för att skapa ett styrsystem ur ett funktionsdiagram, så editering av funktionsdiagram samt konvertering av funktionsdiagrammet till styrbar information är programmets huvuduppgifter. En editorklass och en styrningsklass definieras därför. I figur 2.1 visas vilka typer av funktioner som implementeras i respektive klass.



Figur 2.1 Strukturering av viktiga funktioner hos programmet.



Figur 2.2 Varje byggblock i figuren motsvarar en klass och pilarna visar hur klasserna ärvs. Den streckade linjen mellan Symbolklassen och Editorklassen visar att Editorklassen använder sig av Symbolklassen.

I strukturen i figur 2.1 kan man se att både editerings- och styrningsfunktioner är strukturerade under interaktion. Interaktionen är den dialog som förs mellan användaren och programmet eller mellan en fil (kanal till en process) och programmet.

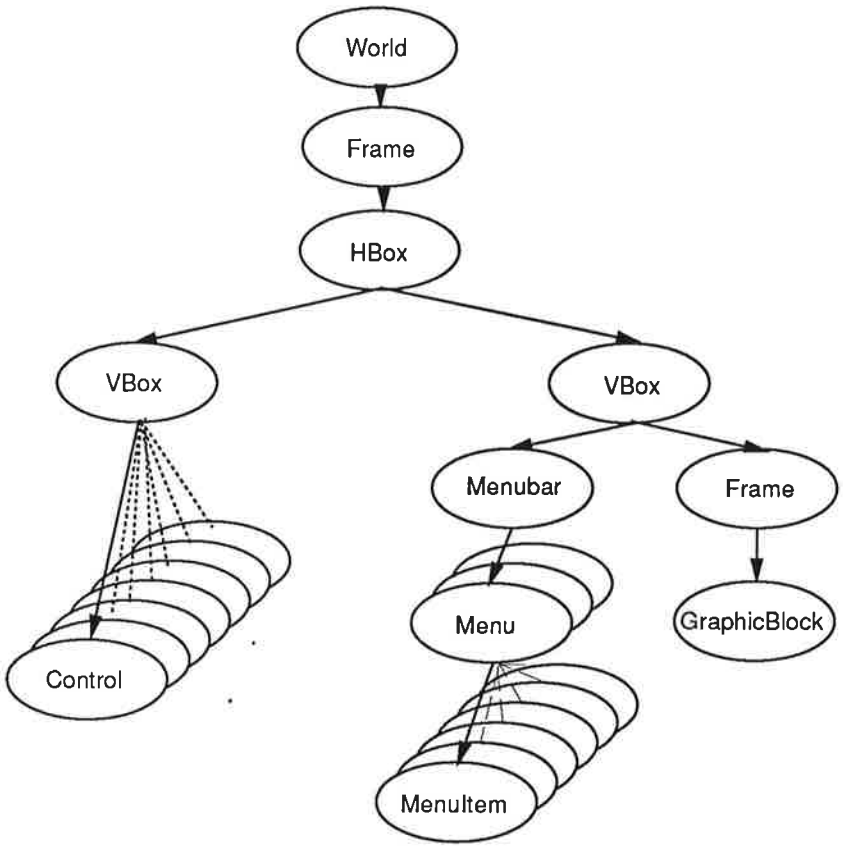
Alla klasser som har någon interaktiv verksamhet är ärvda ur en klass med Interactor som rot i klasshierarkin. Klasshierarkin för Editor och GrafcetControl visas i figur 2.2 och det framgår där att Editor är ärvd ur interaktorn GraphicBlock, som är designad för att hantera grafiska symboler och uppenbarar sig som en grafisk rityta. I GraphicBlock är alla grafiska strukturer och funktioner definierade medan alla underliggande strukturer som är konstruerade för editering och styrning definieras i Editor och GrafcetControl.

Styrklassen GrafcetControl ärver editorklassen Editor och det objekt som skapas i huvudprogrammet av klassen GrafcetControl kan erbjuda alla funktioner, som definierats i klasshierarkin.

Basklassen Interactor är definierad i InterViews och utgör grunden för hela programmet. Egenskaperna hos en interaktor beskrivs i InterViews referensmanual och den bör studeras för att förstå hur det här programmet fungerar i detalj.

Man kan se editering och styrning som två olika moder hos programmet. Beroende av vilken mod programmet befinner sig i kommer det att lyssna efter olika händelser. Vid editering kommer händelser genererade av mus och tangentbord att intressera programmet. En händelse detekteras kanske i en kommandoknapp och en funktion i editorn anropas. Om det t ex är en "move" funktion lägger sig då editorn i en slinga och lyssnar efter de händelser som sker på den grafiska ytan. Vid styrning anmäls även intresse för tidshändelser och inkanalhändelser, som används för synkronisering med omvärlden.

En uppsättning med symbolklasser enligt grafcetnormen deklareras, och det är de som editeras i editorn. Jag deklarerar också en uppsättning logikklasser som representerar logiska operatorer. De används för att beskriva de logiska villkor som kan finnas på en övergång. Symbol- och logikklasserna är två av många klasser som definieras för att Editor- respektive GrafcetControlklassen ska fungera. De och övriga klasser som Editor och GrafcetControl använder förklaras mer ingående i de kapitel där Editor respektive GrafcetControl behandlas.



Figur 2.3 Gränssnittets hierarki.

Knappar (Control, MenuItem) finns deklarerade i InterViews och de är "interaktorer". När de registrerar en händelse anropas en virtuell funktion som specialiseras för varje kommando. Den enda uppgiften jag har gett de här virtuella funktionerna är att anropa en funktion definierad i Editor eller GrafcetControl.

2.3 Initiering

Vid initieringen skapas alla objekt som tillsammans utgör gränssnittet mot användaren. Menuer, knappar och den grafiska ytan komponeras i boxar och ramar till en struktur enligt figur 2.3 och det resulterar i ett gränssnitt som i figur 1.1 i inledningen.

När alla objekt i gränssnittet är skapade och sorterade, anropas GrafcetControls Run-funktion. Den är definierad i basklassen Interactor och har till uppgift att läsa händelser från fönstersystemet och skicka händelserna till rätt interaktor.

3. Editering av Grafcet funktionsdiagram

3.1 Klasser av Grafcet symbolerna

Vid editeringen av funktionsdiagram manipuleras symbolobjekt i editorn. Innan jag börjar förklaringen av editorn kommer därför en genomgång av de standardiserade symbolerna och den symbollista som symbolerna är lagrade i.

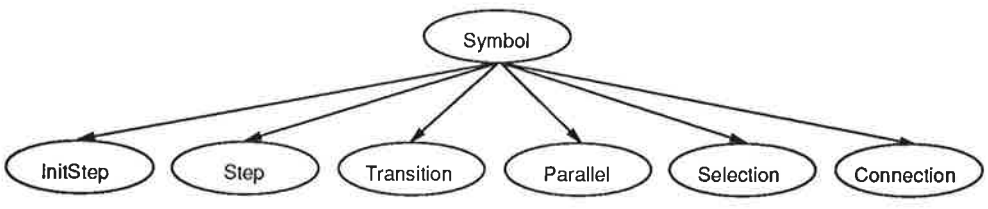
Symbolernas basklass, Symbol

Varje symboltyp i Grafcet-normen representeras av en symbolklass med egenskaper enligt normen. Varje objekt av en given symbolklass innehåller nödvändig information om symbolen samt funktioner för att manipulera den på ett sätt som motsvarar normen.

Trots att de olika symbolklasserna har olika egenskaper ärvs de ur samma basklass Symbol, se figur 3.1. Det gör att hanteringen av symbolerna kan göras på ett smidigt sätt, man kan t ex referera till dem som generella symboler och spara dem tillsammans i samma lista. Denna egenskap kallas polymorfism.

Gemensamma attribut hos symbolerna är:

- Symboltyp.
- Identitet, så att varje symbol blir unik.
- Pekare på de närmaste grannarna, in- och ut-symboler, och förbindelselinjer till de närmaste grannarna.
- Antal symbolobjekt på in- och utgången.
- Nästa och föregående symbolobjekt i symbollistan.
- Information om det grafiska objektets position och omfång.
- Pekare på det tillhörande grafiska objektet.



Figur 3.1 Symbolernas klasshierarki.

Vad som ska definieras i basklassen eller i någon av symbolklasserna får man avgöra med lite sunt förnuft. Basklassen t ex innehåller en del information som inte är nödvändig för alla symboltyper, t ex finns det tre in och utgångar trots att endast två av symbolklasserna, Parallel och Selection, använder så många.

Valet av antalet in- och utgångar gör också att det går bra att använda basklassen om man vill definiera Petrinät symboler. Det kan vara bra eftersom Petrinät och Grafcet funktionsdiagram är släktingar och programmet, kan efter en del modifieringar fungera som en Petrinät editor.

Varje symbolklass har naturligtvis funktioner tillgängliga för att manipulera alla ovanstående attribut. Hur de är deklarerade förklaras i nästa stycke.

Virtuella funktioner

För att symbolerna ska kunna hanteras på ett generellt sätt av editorn bör funktioner med samma innebörd kunna specialiseras i de olika symbolklasserna. Det löser man genom att deklarera funktionen "virtuell" i basklassen, den kan då definieras om hos den ärvande klassen för att få de egenskaper som är speciella för symbolen. Editorn behöver då inte veta vilken symboltyp, som eventuellt hanteras när en funktion anropas, rätt algoritm kommer ända att bli anropad. Följande funktioner definieras virtuella i basklassen

- Skapa och returnera det grafiska objekt som är relaterat till symbolen. Om objektet är sammansatt av flera grafiska objekt, samlas de i en bild innan de returneras.
- Justera vidden på det grafiska objektet.
- Editera text.
- Returnera omfång.
- Ändra omfång.
- Returnera brytpunkter hos förbindningslinje.
- Returnera objekttyp.

Funktioner som endast definieras i basklassen

Basklassen är skapad för att vara så generell att många funktioner kan vara gemensamma för de olika symbolklasserna. Här kommer några exempel på de funktioner som inte behöver definieras om i de ärvande symbolklasserna.

- Förbind ingång eller utgång.
- Ta bort förbindning på in eller utgång.
- Returnera antal in eller utgångar.
- Returnera symbol som är ansluten till in eller utgång.
- Returnera förbindelselinje som är ansluten till in eller utgång.
- Returnera eller sätt nästa symbol i listan.

- Returnera eller sätt föregående symbol i listan.
- Returnera typ.
- Returnera eller sätt identitet.
- Returnera det relaterade grafiska objektet.
- Returnera eller sätt centrum på det grafiska objektet.

Förbindningslinjer

En förbindningslinje är egentligen ingen symbol, men eftersom den måste kunna lagras och manipuleras definieras en symbolklass för den. Förbindninglinjeklassen är den klass som skiljer sig mest från de andra symbolklasserna, eftersom den endast kommer ihåg vilken position och vilka brytpunkter den har. Övriga symboler håller även reda på sin närmaste omgivning i funktionsdiagrammet.

Symbollistan

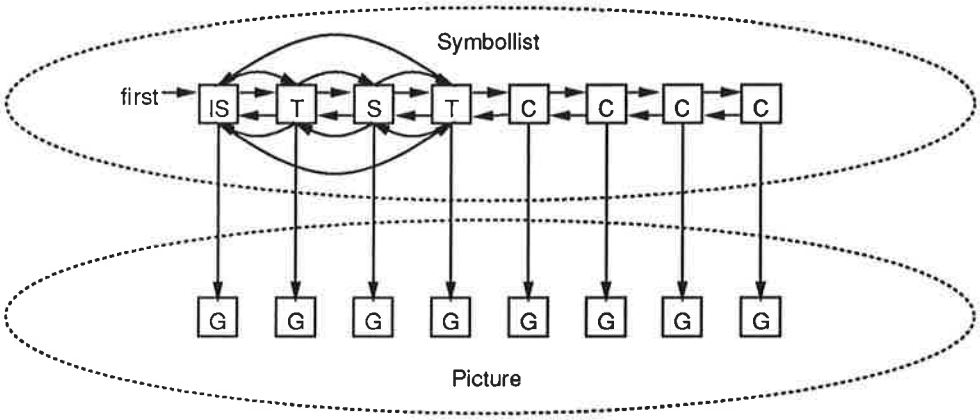
I editorn sparas symbolerna i en lista. Listan är definierad som en klass och den kan endast spara symboler. Följande listhanteringsfunktioner finns tillgängliga:

- Skapa listan.
- Returnera pekare på den första symbolen i listan.
- Returnera symbol av viss symboltyp och identitet.
- Returnera symbol som träffas av ett musklick.
- Sätt in en symbol i listan.
- Ta ut en symbol ur listan.
- Rensa listan från symboler.

3.2 Klassen Editor

I klassen Editor är alla editeringsfunktioner och filhanteringsfunktioner definierade. Funktionerna anropas från de olika kommandoknapparna som är placerade antingen i paletten eller i någon av menuerna. I funktionerna manipuleras symbolerna som beskrevs i föregående kapitel. Editorn ärver InterViewsklassen GraphicBlock och är en interaktor som är specialiserad för att hantera grafikobjekt. Editorn innehåller följande information

- Symbollistan som beskrevs ovan.
- En bild där alla grafiska objekt är samlade.
- Namnet på det aktuella funktionsdiagrammet.



Figur 3.2 Symbolerna i symbollistan pekar på varandra dels i form av en dubbellänkad lista och dels för att veta vilka de är förbundna med. Varje symbol, förutom förbindningarna, pekar också på på den förbindning som är ansluten till en förbunden granne men bilden skulle bli för otydlig med fler pilar.

- Är det aktuella funktionsdiagrammet kompilerat?
- Aktuell mod, styr eller editera.
- Finns initialsteg?
- Antal steg, övergångar och förbindningar.
- Antal parallella delningar och sammanföringar.
- Antal alternativa delningar och sammanföringar.

I bilden sparas alla grafiska objekt som utgör den synliga delen av funktionsdiagrammet och i symbollistan sparas alla symboler, som utgör den underliggande strukturen hos funktionsdiagrammet. I figur 3.2 visas hur bilden och symbollistan är strukturerade. Lägg märke till att symbolerna i symbollistan pekar på varandra dels i form av en dubbellänkad lista och dels för att veta vilka de är förbundna med. Varje symbol, förutom förbindningarna, pekar också på den förbindning som är ansluten till en förbunden granne men bilden skulle bli för otydlig om fler pilar skulle ha ritats in.

3.3 Editeringsfunktioner

Alla editeringsfunktioner är definierade i editorn. Förutom de funktioner som är direkt knutna till ett kommando finns det en uppsättning med hjälpfunktioner i editorn. Jag kommer här att förklara algoritmerna i funktionerna. En gemensam egenskap för de funktioner som detekterar händelser på den grafiska ytan är att de avslutas om musklick detekteras utanför den grafiska ytan.

SlideRect

För att kunna positionera figurer används rörliga objekt, sliders. I editorn använder jag en rörlig rektangel som när den anropas ges måttet av den symbol som ska positioneras.

När SlideRect anropas lägger den sig i en loop och tolkar de händelser som sker med musen på den grafiska ytan. Vid första klicket skapas en "rubberrect" och vid rörelsehändelser följer rektangeln med musen genom att raderas och återskapas på en ny position. När nästa musklick detekteras avslutas funktionen och positionen returneras som argument.

AdjustX och AdjustY

Adjust funktionerna fungerar som magneter på symboler som skapas. En symbols mittpunkt dras till en tiotal kordinat både i x- och y-led. Användaren behöver då inte ha fingertoppskänsla vid positioneringen av symboler.

Skapa symboler

En symbol skapas på följande vis

- Skapa symbolen
- Positionera symbolen med SlideRect
- Låt symbolen veta vilken position den ska ha
- Skapa symbolens grafiska objekt och lagra det i editorns bild
- Lagra symbolen i symbollistan

HitSymbol

HitSymbol är en viktig funktion som i en loop detekterar händelser på den grafiska ytan. När en musknapp trycks ner, undersöks alla symboler i symbollistan för att se om någon symbol fanns på den position, som musen hade vid nedtryckningen. Om ingen symbol fanns på positionen returneras nil.

GrowConnection

När en förbindelse skapas mellan två symboler består den av en multiline. Linjen skapas i en loop där händelser på den grafiska ytan detekteras och tolkas för att hitta brytpunkter och symbolanslutningar. Loopen avslutas när två symboler blivit träffade av musklickningar, och de två symbolerna kommer att följa med som argument tillsammans med linjens brytpunkter från funktionen.

Connect

GrowConnection anropas och när den sedan svarar med de två symbolerna som ska förbindas anropas ett objekt som kollar syntaxen. Om syntaxen är riktig, skapas ett symbolobjekt som beskriver förbindelselinjen, och de båda symbolerna delges om förbindelsen. Förbindningens ändpunkter justeras för att passa de förbundna symbolerna och hålla funktionsdiagrammet prydligt. När förbindelseobjektet skapas, lagras det i symbollistan precis som de andra symbolerna och det grafiska objektet skapas också och lagras i editorns bild.

Editeringshjälp mot felaktiga förbindningar

I förklaringen till hur symboler förbinds ovan nämnde jag att syntaxen kontrollerades innan förbindningen skapades. Det görs i ett regelverk som undersöker om den nya förbindelsen till den aktuella symbolen kan existera tillsammans

med tidigare förbindningar. Användaren får ingen chans att förbinda på ett felaktigt sätt, utan en dialogruta talar om vilket fel som gjorts, och editorn ignorerar sedan förbindelsen.

AdjustWidth

När parallella eller alternativa förgreningar förbinds med andra symboler justeras vidden på dem så att funktionsdiagrammet håller sig snyggt och vinkelrätt. Förbindelselinjens ändpunkter justeras samtidigt för att passa de symboler som den har förbundit.

Disconnect

Disconnect tar bort alla förbindelser till symbolens grannar samt delger grannarna att symbolen raderats eller flyttats.

Move

En symbol väljs med HitSymbol. Symbolens förbindningar tas bort enligt Disconnect. Symbolens grafiska objekt raderas och en SlideRect placeras på samma ställe. Ett nytt grafiskt objekt skapas på den position som användaren väljer med musen.

Delete

En symbol väljs med HitSymbol. Symbolens föbindningar tas bort med Disconnect och symbolen tas bort ur symbollistan och bilden. Symbolen raderas.

Text

En symbol väljs med HitSymbol och symbolens EditText funktion anropas. EditText är en funktion som deklareras om och specialiseras hos de ärvande klasserna. De enda klasserna som använder text är Step, InitStep samt Transition, och hos dem definieras EditText till att anropa lämpliga dialogrutor som hanterar den text som är intressant för klassen. När all ny textinformation är inskriven uppdateras klassens grafiska objekt.

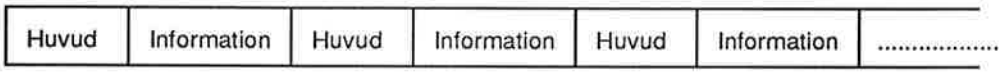
3.4 Lagring av funktionsdiagram

När information om ett funktionsdiagram ska lagras på en fil måste all information göras om till tecken och heltal, objekt kan ej sparas på binär form på en fil. Detta är ett generellt problem med C++. Ett språk skapas därför som talar om hur varje symbol ska skrivas eller läsas på filen.

Format

I figur 3.3 visas hur symboler ligger lagrade på en fil. Varje symbol består av ett huvud och en informationsdel. I huvudet ligger symbolens identitet, och när huvudet är inläst vet man därför, hur informationsdelen ska läsas och tolkas. All information som en symbol har om sin position, text och grannar mm ligger som tecken och heltal i informationsdelen. I bilaga A beskrivs hur en fil, där ett funktionsdiagram ligger lagrat ser ut, och hur de olika symbolerna representeras.

Fil med funktionsdiagram



Figur 3.3 Symbolerna i filen består av ett huvud och en informationsdel. När huvudet är inläst vet editorn hur informationsdelen ska läsas och tolkas.

Skriv på en fil

Varje symbolklass har en funktion "write on file" som skriver symbolens huvud och informationsdel enligt ovan. För att lagra ett funktionsdiagram på en fil anropar alla symboler i symbollistan "write on file."

Inläsning i två varv

När en fil laddas in, blir den avläst två gånger. I det första varvet skapas alla symboler och fylls med information om vilka de är och vilken position de ska ha, de sparas i symbollistan och dess grafiska representation i bilden. Varje symbol har en konstruktor med ett filnamn som argument, den används endast när en fil laddas in. Under det andra varvet skapar symbolerna ett intresse för sin omgivning och pekar på sina grannar och förbindelserna till grannarna.

4. Generering av styrfunktioner ur funktionsdiagram

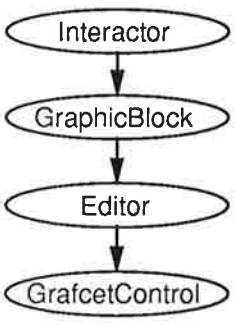
GrafcetControl är den klass som genererar matriser och logik ur den information som är uppbyggd i editorn och lagrad i symbollistan. GrafcetControl är ärvd ur klassen Editor och är med andra ord en interaktor. Det objekt som skapas av typen GrafcetControl innehåller alla funktioner som är definierade hos fäderna dvs Editor, GraphicBlock och Interactor.

Objektet av GrafcetControl arbetar i olika moder, antingen i styrmod då funktionerna definierade i GrafcetControl används eller i editeringsmod då funktionerna i Editor används. En flagga sätts som talar om vilken mod vi arbetar i och bestämmer vilka funktioner, som är tillgängliga. När moden ändras, kommer även intresset för händelser att ändras, eftersom tids- och inkanalhändelser måste kunna detekteras i styrmoden.

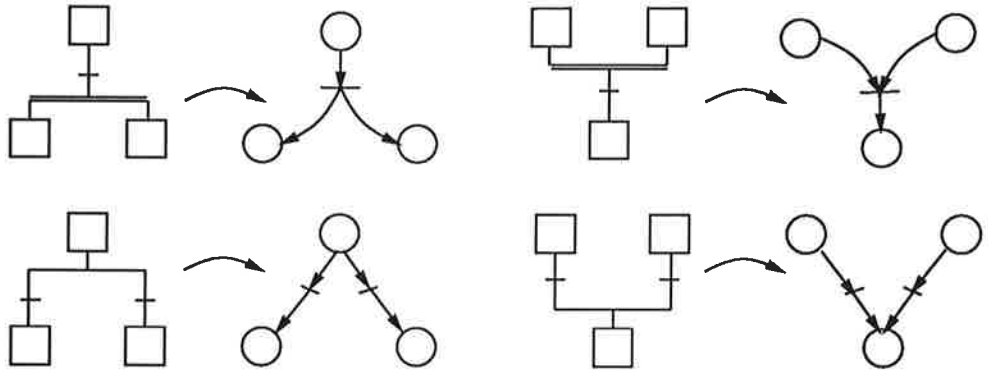
För att fungera som styrsystem behöver GrafcetControl följande information:

- Händelsematris.
- Tillgänglighetsmatris.
- Vektor med steg.
- Vektor med övergångar.
- Tabell med variabler.
- Tabell med order.
- Tidtabell
- Vektorer med information om stegen.

Hur de olika strukturerna fungerar, används och byggs upp förklaras i efterföljande stycken.



Figur 4.1 Klasshierarkin ovanför GrafcetControl.



Figur 4.2 Exempel på hur funktionsdiagrammet behandlas som ett Petrinät vid beräkningen av händelse och tillgänglighetsmatris.

4.1 Relationsmatriser och tillgänglighet

För att kunna använda funktionsdiagrammet till styrning omräknas listan till matrisform. Här har jag använt en del ideer, som jag har fått när jag har läst om Petrinät (Murata, 1989). En händelsematris skapas som talar om hur stafettpinnar ska förflyttas i funktionsdiagrammet när händelser inträffar och en tillgänglighetsmatris skapas som talar om om en övergång är tillgänglig. En övergång måste vara både tillgänglig och sann för att en tillståndsändring ska ske. För att kunna skapa händelsematris och tillgänglighetsmatris tänker jag funktionsdiagrammet som ett Petrinät. Steg och övergångar kan då ha mer än en in/utgång och symbolerna för parallell och alternativ förgrening tas bort. Figur 4.2 visar hur Grafcet funktioner behandlas som Petrinät funktioner.

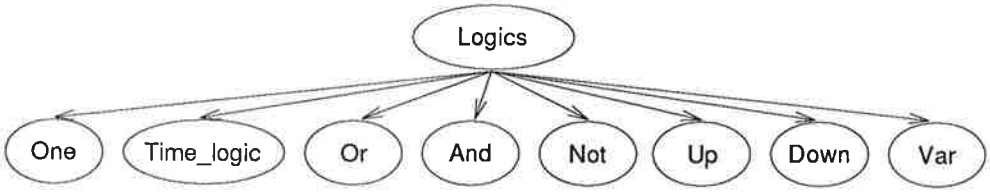
Händelsematrisen och tillgänglighetsmatrisen skapas med utgångspunkt från övergångarna, $t_0 \ldots t_m$. Symbollistan söks igenom och varje gång en övergång hittas undersöks vilka steg som finns på dess in- och utgångar. Elementen i matriserna formas sedan därefter, om tex övergång 3 föregås av steg 5 blir elementet på rad 5 och kolonn 3 minus ett.

Ekvation 4.1 visar hur uppdateringen av tillstånden $s_0 \ldots s_n$ beräknas. När en övergång blir sann kommer varje tillstånd som beror på övergången att räknas om och stafettpinnar byts enligt elementen i händelsematrisen som kan vara 0, 1 eller -1.

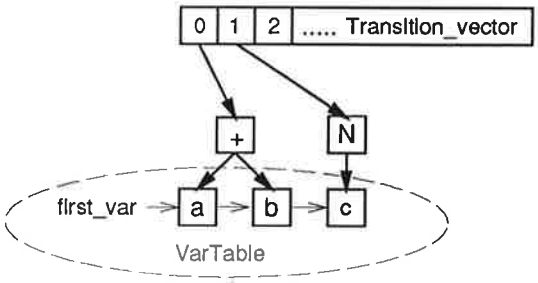
$$\begin{pmatrix} s_0 \\ s_1 \\ \vdots \\ s_n \end{pmatrix} = \begin{pmatrix} s_0 \\ s_1 \\ \vdots \\ s_n \end{pmatrix} + \begin{pmatrix} a_{00} & a_{01} & \cdots & a_{0n} \\ a_{10} & a_{11} & \cdots & a_{1n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m0} & a_{m1} & \cdots & a_{mn} \end{pmatrix} \begin{pmatrix} t_0 \\ t_1 \\ \vdots \\ t_m \end{pmatrix}. \qquad (4.1)$$

Förflyttning av stafettpinnar

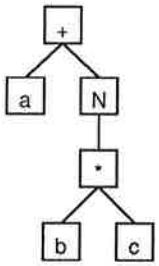
När uppdateringen av stafettpinnar sker i funktionsdiagrammet, använder jag en ritfunktion som bara uppdaterar en begränsad del av det grafiska blocket. Om hela det grafiska blocket uppdateras vid varje ändring, kommer funktionsdiagrammet att blinka vid en uppdateringen, eftersom den uppdaterade grafiken raderas innan den ritas om.



Figur 4.3 Klasshierarkin hos logikklasserna.



Figur 4.4 Den logiska relationen mellan ingångar och övergångar hanteras med en struktur uppbyggd av logikobjekt.



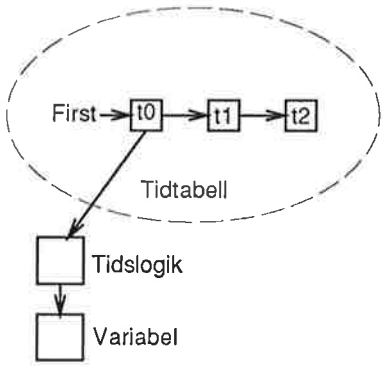
Figur 4.5 Struktur för det logiska uttrycket $a + NOT(b * c)$.

4.2 Hantering av logiken i övergångarna

För att hantera logik inkluderas logikbiblioteket där alla logikklasserna är deklarerade. Alla logiska operatorer ärvs ur basklassen Logics enligt figur 4.3. I figuren är alla tillgängliga operatorer med. Operatorerna anges med C-syntax.

Sambandet mellan inportarna och övergångarna beskrivs av ett logiskt villkor och ur det villkoret byggs en trädstruktur med logikobjekt upp för varje övergång enligt figur 4.4. I figuren är de logiska villkoren inritade för två övergångar med de logiska villkoren $NOT(c)$ och $a + b$. Strukturerna är uppbyggda mellan en vektor och en lista. Vektorn som representerar övergångarna innehåller pekare på logiska objekt och de logiska variablerna som utgör trädstrukturens löv samlas i en lista.

När trädstrukturen skapas anropas en rekursiv funktion som söker igenom villkorssträngen för att hitta den logiska operand som ska bli trädets fader. Den rekursiva funktionen returnerar en pekare på ett objekt som representerar den hittade operandtypen. De delsträngar som finns till höger och vänster om operanden genomsöks på samma sätt till löven i trädet hittats. Löven är de variabler som använts i det logiska uttrycket. Figur 4.5 visar hur trädstrukturen för $a + NOT(b * c)$ ser ut.



Figur 4.6 I tidtabellen sparas objekt som håller reda på tiden för nästa tidshändelse samt vilken tidslogik som ska updateras vid tidhändelsen.

4.3 Tidsberoende övergångar

Tidsberoende övergångar behandlas på ett speciellt sätt. Den logiska strukturen byggs upp på samma sätt som hos de övriga logiska operanderna, men när man frågar om en övergång är sann är det skillnad. När en flank hos det logiska uttrycket till den tidsberoende övergången detekteras, beräknas den tid då övergången ska ändra tillstånd. Tiden sparas som ett tidsobjekt i en tidtabell där tidsobjekt sorteras och sparas, se figur 4.6.

Tidtabellen kontrolleras vid varje händelse och tiden till nästa tidshändelse används för att beställa en tidshändelse. När tiden för det första tidsobjektet i listan uppnåtts, kommer en tidshändelse, och det logiska uttryck som beställde tidshändelsen uppdateras. Tidsobjektet tas ur listan och raderas.

Tid sparas som flyttal för att få en bra tidsmässig upplösning samt för att kunna hantera de stora tal som returneras av realtidsklockan. Realtidsklockan har tiden sparad i sekunder och millisekunder sedan 1970.

4.4 Flanktriggade övergångar

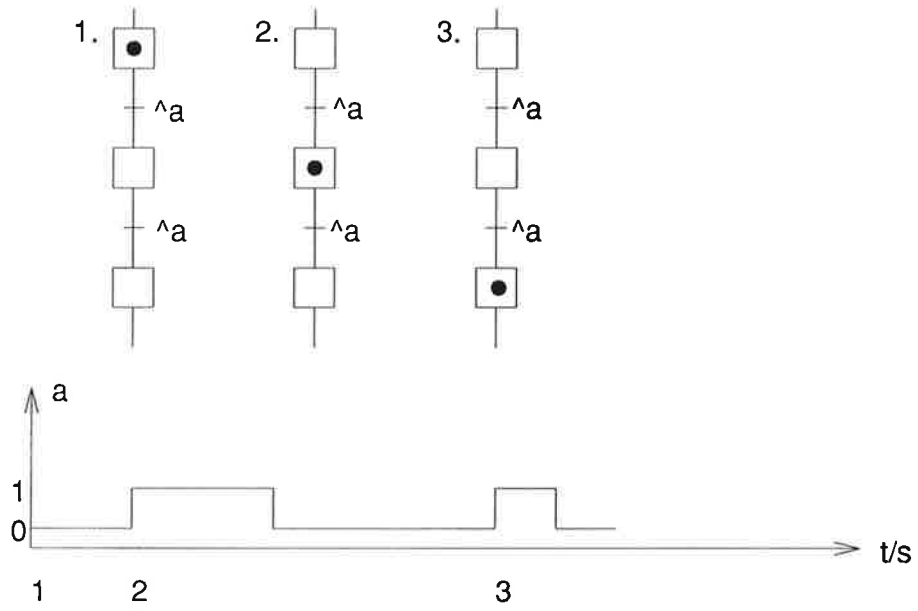
När två efterföljande övergångar flanktriggas på samma variabel enligt figur 4.7 ska endast den tillgängliga övergången bli sann. För att klara av det har jag blivit tvungen att kompromissa lite. Flank och tidsberoende övergångar kan nu endast trigga på en ensam variabel och inte på ett logiskt uttryck bestående av flera variabler.

Jag har löst det här problemet genom att först detektera alla övergångar som är sanna. Innan funktionsdiagrammet uppdateras låter jag sedan alla flanktriggande övergångar passera den sist detekterade flanken.

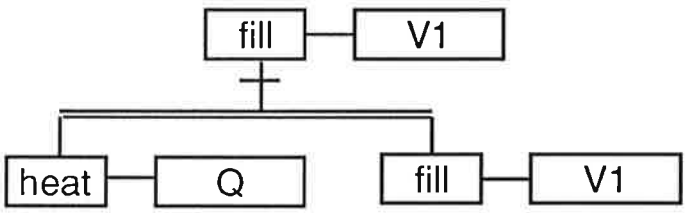
Med den här metoden kan ävergångar i olika funktionsdiagram eller i parallella förgreningar detektera samma flank hos en variabel.

4.5 Hantering av order från aktiva steg

Vad gäller de order som ska utföras när steg blir aktiva finns utvecklingsmöjligheter i programmet. För närvarande bearbetas endast den variabel som anges



Figur 4.7 När flera flanktriggade övergångar triggar på samma variabel ska endast de aktiva övergången detektera att flanken inträffat.



Figur 4.8 Eftersom samma order kan skickas av efterföljande steg är det viktigt att inga order skickas innan funktionsdiagrammets tillstånd blivit stationärt.

som "action" i dialogrutan för steg. När en inkanal- eller tidshändelse detekteras och ett eller flera steg ändrar tillstånd uppdateras en tabell som innehåller alla order i funktionsdiagrammet. När funktionsdiagrammet har slutat ändra tillstånd efter den inkommande händelsen skickas alla order som har ändrat tillstånd på en fil tillsammans med sitt tillstånd. Om order skickas direkt när steget ändrar tillstånd kan problem uppstå i funktionsdiagram likt det i figur 4.8. I det fallet ska inte v1 stänga när den parallella förgreningen görs, utan vara öppen hela tiden. Tillståndet hos ordern får ej heller bli beroende av vilket steg som undersöks först.

I Grafcet finns möjligheter att ytterligare precisera de order som ska utföras när steg blir aktiva. Detaljerade kommandon kan anges tex (S)tored, (D)elayed, (C)onditional, (L)imited eller (P)ulse. Om detaljerade kommandon ges måste även beskrivningar ges för det speciella kommandon. Ett Lagrat kommando (S) måste veta om ett start/stop ska ske och ett tidsberoende kommando måste veta tiden. De här preciseringarna har jag inte behandlat i programmet.

För att lösa de här problemen kan samma logik och tidsklasser användas som tidigare beskrevs till övergångarna och dess variabler. En motsvarande sambandsstruktur måste skapas mellan en steg- och ordervektor som mellan variabeltabellen och övergångsvektorn.

4.6 Synkronisering och kommunikation

När funktionsdiagrammet är klart och all styrinformation beräknad kan vi använda programmet för att styra någon process. Kommunikationen till en eventuell process (Där variabler relateras till portar i en I/O-enhet) ska ske via en fil, men eftersom programmet endast är i simuleringsstadiet simuleras en process genom att variabelnamnet och dess tillstånd skrivs på tangentbordet av användaren. Utskrifter som ska sändas på en fil skrivs temporärt i ett fönster.

Go

Kommandot Go under Sim menyn sätter en flagga som talar om att vi styr samt att vi är intresserade av tidshändelser och insignalhändelser. Stafettpinnar placeras i alla initialsteg. Eftersom inga andra interaktorer i programmet är intresserade av de händelserna kommer de att skickas till grafcetcontrols Handle funktion.

Handle och Calc

Då en händelse inträffar och vår Handle-funktion aktiveras frågar Handle-funktionen om det var en inkanal- eller tidshändelse. Om det var en tidshändelse tas det första tidsobjektet ut ur tidtabellen, det tidsberoende logiska objektet uppdateras och reglerloopen snurrar sedan ett varv genom att funktionen Calc anropas. Om det var en inkanalhändelse som inträffade läser vi in variabelnamnet och dess tillstånd, variabellistan uppdateras sedan och Calc anropas.

Reglerloopen Calc fungerar enligt följande:

- Undersök alla övergångars logiska villkor och lagra identiteten hos de övergångar som ska avfyras.
- Låt alla icke tillgängliga övergångar passera eventuella flanker.
- Om en övergång är tillgänglig och dess villkor är sant händer följande.
- Uppdatera gruppen med tillstånd enligt formel 4.1.
- Uppdatera stafettpinnarna så att de motsvarar tillstånden.
- Uppdatera variabellistan.
- Uppdatera tabellen med order för de steg som ändrat tillstånd.
- Skicka alla förändrade order på en kanal.

Det är viktigt att exekveringsordningen vid uppdateringen av styrsystemet följer ordningen enligt ovan.

Stop

Om kommandot Stop väljs slutar programmet att intressera sig för de ovan nämnda händelserna och programmet går tillbaka i editeringsmod. Alla stafettpinnar raderas.

5. Användarmanual

5.1 Start av programmet

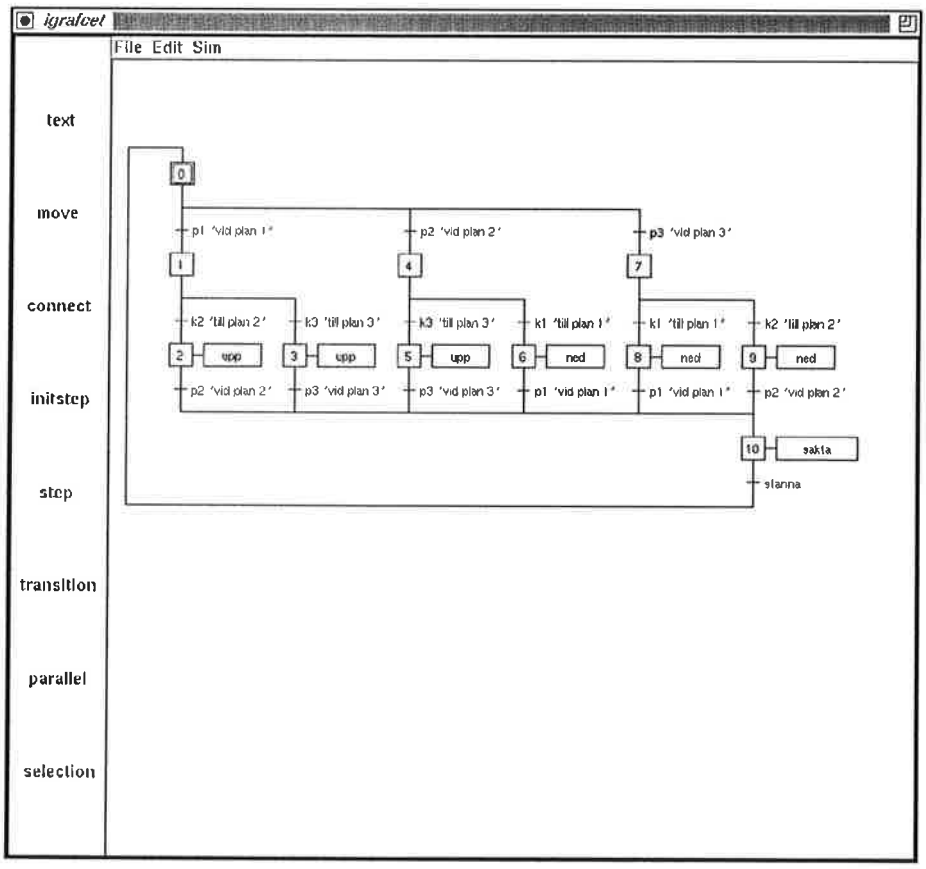
Bibliotek för funktionsdiagram

Innan programmet startas ska en katalog skapas på den plats där programmet ligger. I katalogen kommer funktionsdiagram att sparas på filer. Döp katalogen till gcdir.

Start

Skriv igrafcet på tangentbordet, programmet startas då och ett fönster likt det i figur 5.1 kommer då fram på skärmen.

Fönstret består av en palett med "knappar" som representerar de olika symbolerna samt ett par editeringskommandon(move, text och connect). Den fria



Figur 5.1 Igrafcets gränssnitt består av en grafisk yta samt en uppsättning knappar och menyer. Den som har använt ett ritprogram innan bör känna igen sig.

ytan är det grafiska blocket, det känner av mushändelser och där byggs funktionsdiagrammet upp. File, Edit och Sim är menyknappar som öppnar rullgardinsmenyer.

5.2 Editering av Grafcet funktionsbeskrivning

Editeringen sköts till stor del med musen, endast texteditering sker via tangentbordet.

Val av symboltyp eller editerings kommando

Klicka med musen på önskad symboltyp på paletten, eller klicka på connect, move eller text. Den valda knappen inverteras för att indikera att den är aktiv. Så länge knappen är inverterad kommer motsvarande uppgift att utföras under tiden användaren utför rätt händelsemönster med musen på den grafiska ytan.

Placering av symbol

Placera symbolen på den grafiska ytan genom att först klicka på ytan en gång. En rörlig symbol kommer då fram för att fixeras på den position musen trycks ned nästa gång. Om samma symboltyp ska placeras igen, upprepas ovanstående.

Förbindning av symboler, connect

Välj connect på paletten. Välj ut vilka symboler som ska förbindas. Dra förbindningen genom att först klicka på frånsymbolen och sedan på tillsymbolen. Tänk på riktningen då förbindningen dras, från → till.

Ändring av en symbols position, move

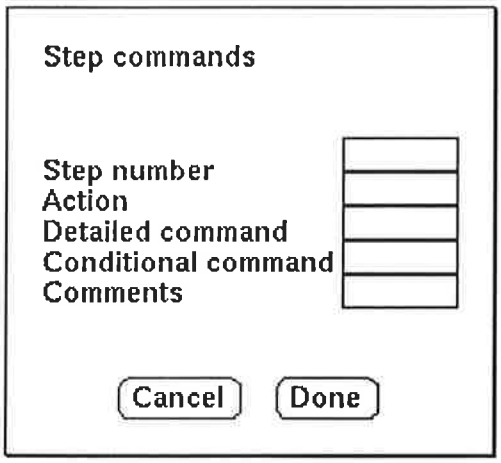
Välj move på paletten. Välj vilken symbol som ska flyttas genom att klicka på den. Förbindningarna till symbolen raderas nu automatiskt. Ge symbolen en ny position och fixera den med ett klick. Observera att förbindningslinjer ej kan flyttas, och de reagerar inte när man klickar på dem med musen.

Radering av symbol, delete

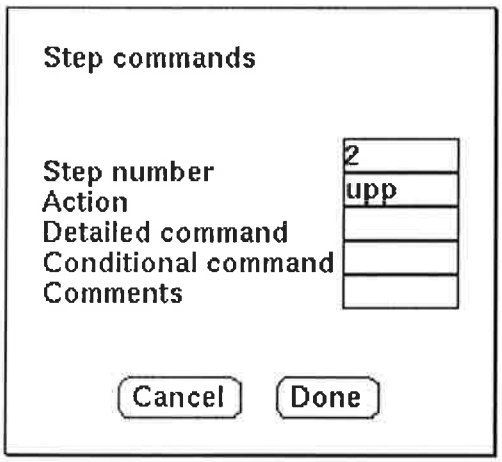
Välj delete i rullgardinsmenyn under Edit. Välj vilken symbol som ska raderas och klicka på den. En dialogruta visas nu så att en ångerchans ges. Svara cancel eller yes i dialogrutan. Observera att förbindningslinjer ej kan raderas med delete och de reagerar inte när man klickar på dem med musen. För att radera en förbindelselinje får man flytta på eller radera en symbol som är förbunden med linjen.

5.3 Texteditering av övergångar och steg

Välj text på paletten och klicka sedan på önskad symbol. En dialogruta visas där textinformation ska skrivas. Text kan endast knytas till steg och övergångar, om man väljer andra symboler vid textediteringen händer ingenting.



Figur 5.2 Dialogruta för texteditering av steg. Om inga order anges kan man ange kommentarer i stället. Om både order och kommentarer anges skivs endast order ut i funktionsdiagrammet.



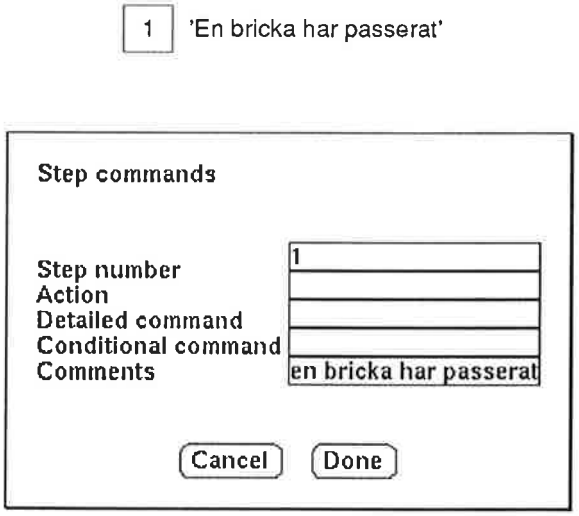
Figur 5.3 En dialogruta, som användes vid editeringen av ett steg i hissens funktionsdiagram, där en order skrivits.

Texteditering av steg

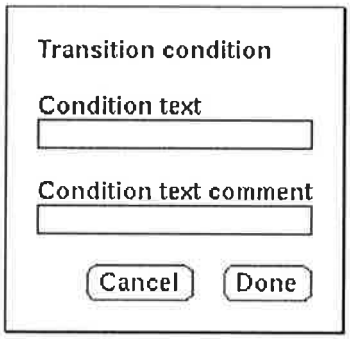
Om det var ett steg som valdes kommer en dialogruta enligt figur 5.2 fram på monitorn. I dialogrutan skrivs text in på den plats musen placeras på. Eftersom inte Detailed command och Conditional command behandlats, är det inte lönt att skriva något på de platserna.

Förutom identitet (Step number), kan order och kommentarer knytas till steg. Om både order och kommentarer anges till ett steg kommer endast ordern att skrivas ut. Order till stegen skrivs in i dialogrutan enligt figur 5.3. Dialogrutan i figur 5.3 resulterade i ett av stegen i funktionsdiagrammet till hissen se figur 5.1. När ett steg ändrar tillstånd skickas ordern på en fil tillsammans med stegets tillstånd.

Om man vill skriva en kommentar till ett steg görs det enligt figur 5.4, där även det grafiska resultatet visas överst. Texten i kommentarerna används inte av styrfunktionerna utan är endast till för förtydligande.



Figur 5.4 När en kommentar knyts till ett steg placeras den till höger om steget, överst i bild. Kommentaren påverkar inte styrfunktionerna utan är endats till för förtydligande.



Figur 5.5 Dialogruta för texteditering av övergångar.

Texteditering av övergångar

När ett steg ska textediteras erhålls en dialogruta enligt figur 5.5. På övergångarna ska det önskade logiska villkoret skrivas i rutan Condition text och kommentarer i rutan Comments. Det logiska uttrycket kan naturligtvis bestå av en ensam variabel. Observera att inga mellanslag får skrivas i den textsträng som beskriver det logiska uttrycket hos en övergång.

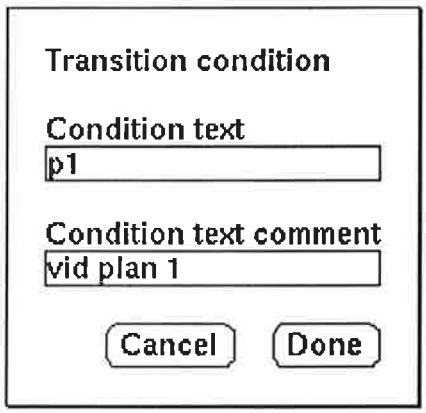
För att undvika konflikt mellan övergångar som följer en alternativ förgrening, ska övergångsvillkoren utesluta varandra enligt figur 5.1.

I figur 5.6 visas hur en dialogruta till en av "hissens" övergångar ser ut. Det grafiska resultatet hos övergången kan man se i figur 5.1, kommentaren skrivs till höger om det logiska uttrycket.

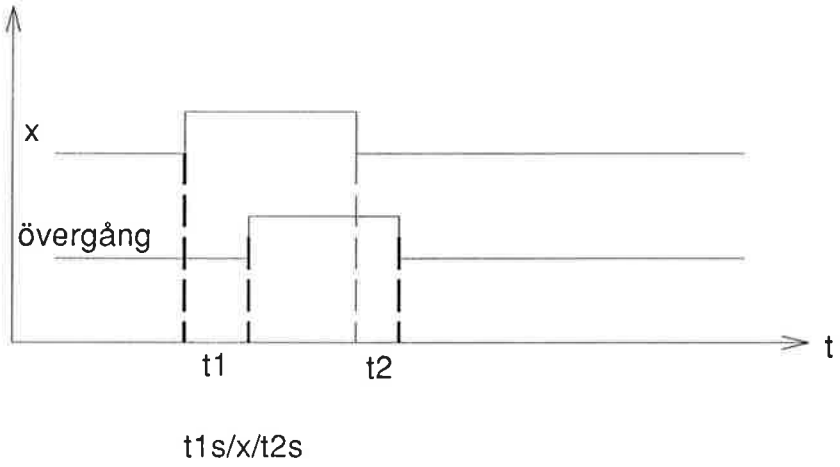
Logiska operandeder anges med C-syntax i det logiska uttrycket och de ävriga möjliga operanderna för upp- respektive nedflank samt för tidsberoende anges på följande vis.

Uppflank - $\wedge$

Nedflank - _



Figur 5.6 En dialogruta till en av "hissens" övergångar.



Figur 5.7 En övergång kan ha två tidsberoenden. Ett tidsberoende för uppflank, t_1, och ett för nedflank ,t_2.

Tidsberoende - tex 3s/x/2s där de två tiderna $t_1 = 3s$ och $t_2 = 2s$ är de fördröjningar som visas i figur 5.7.

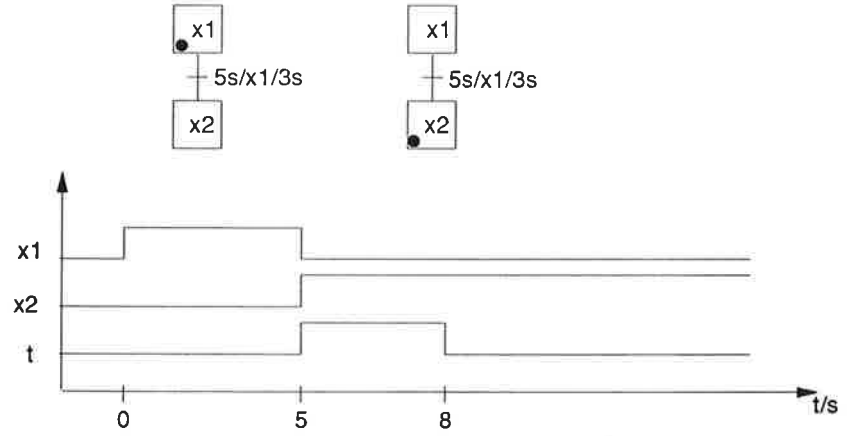
En övergångs tidsberoende används för att fördröja en variabel, men det kan även användas för att fördröja ett steg. När vi vill fördröja ett steg måste vi lyssna på stegets tillstånd och sedan lägga fördröjningen i den efterföljande övergången, enligt figur 5.8.

När all information angetts tryck Done, eller Cancel om ingenting ska utföras.

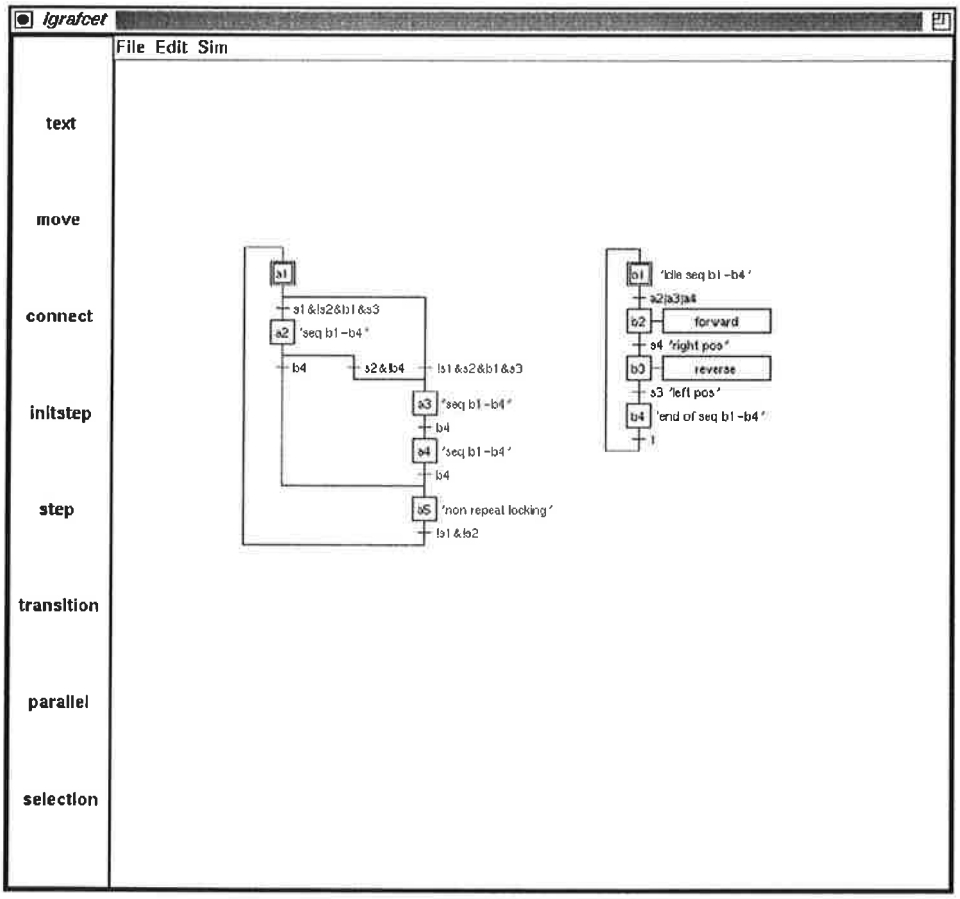
5.4 Parallella aktiviteter

Parallella aktiviteter kan ske på olika sätt i programmet. Antingen genom att använda de parallella förgreningarna eller sammanförningarna(samma symbol), som finns tillgängliga i paletten, eller genom att rita flera jämlöpande funktionsdiagram.

En anledning till att använda mer än ett funktionsdiagram kan vara för att definiera procedurer, som får ersätta de macrosteg som inte finns tillgängliga. För att detta ska vara möjligt måste övergångar kunna lyssna på stegs tillstånd.



Figur 5.8 Om en tidsberoende övergång synkroniseras av föregående steg, kan steget fördröjas.



Figur 5.9 Funktionsdiagrammet till höger fungerar som en procedur och "anropas" från det högra funktionsdiagrammet. Synkroniseringen mellan de båda sker genom att övergångarna triggar på strategiska stegs identitet.

Det görs genom att stegets identitet(Nr) anges som övergångens logiska variabel. Ett exempel på det visas i figur 5.9, där två funktionsdiagram är skapade som synkroniserar varandra via stegens identitet.

Om man vil kan även jämlöpande funktionsdiagram fungera helt oberoende av varandra.

5.5 Filhantering

Under "File" i menyn finns kommandon för filhantering. Spara eller läs t.ex.

New

Ge plats åt ett nytt funktionsdiagram, det gamla raderas.

Open

Välj ett funktionsdiagram i filväljaren. Tryck open eller C/R och funktionsdiagrammet blir inläst.

Save

Spara funktionsdiagrammet under det namn den har. Om inget namn angetts heter funktionsdiagrammet "nameless".

Save As

Spara funktionsdiagrammet under det namn som Ni anger i dialogrutan. Välj ett namn i fil menyn eller ange ett nytt. Tryck save eller C/R.

Quit

Quit avslutar programmet.

5.6 Simulering och styrning

Kommunikation

När programmet används för att styra något ska kommunikationen ske via en fil till en I/O-enhet. Varje förändring medför att variabelnamnet och dess tillstånd skickas till programmet. Vid simulering, som är det enda provade hittils, används standard kanalerna för all kommunikation.

Go

Go startar styrningen och initierar funktionsdiagrammet och stafettpinnar placeras alla initialsteg. För att simulera en händelse på en inport används standardinput, placera markören i samma fönster som programmet startades i och skriv variabelnamn, tillstånd (0 eller 1) och C/R. På standard output kommer på samma sätt variabelnamn och tillstånd hos utvariabler att skrivas ut. Vilka tillståmd som är aktiva syns i funktionsdiagrammet då de har en stafettpinne.

När programmet befinner sig i styrmod är inga editeringskommandon tillgängliga. För att åter kunna editera funktionsdiagrammet måste styrningen avslutas med stop.

Stop

Stop avslutar styrningen och återställer funktionsdiagrammet (raderar alla stafettpinnar).

6. Slutsatser

Jag har konstruerat ett program som kan editera Grafcet funktiondiagram och ur dem generera styrsystem. Det motsvarar de mål som jag hade när jag påbörjade examensarbetet. Naturligtvis kan programmet förbättras och utökas, men någonstans måste man sluta. Exempel på lämpliga utvidgningar i programmet vore att definiera macrosteg samt utöka möjligheterna att ytterligare påverka order hos steg.

Under examensarbetet har jag lärt mig hur man programmerar objektorienterat och använder ett bibliotek som InterViews för skapa sitt användargränssnitt. För att konstruera programmet har jag fått lösa många olika typer av problem och jag tycker att jag har lärt mig mycket. Ett viktigt problem, som inte är beroende av programspråket, är beräkningsordningen vid uppdateringen av styrsystemet. Eftersom jag inte kunde C++ när jag påbörjade examensarbetet, hade jag inte riktigt klart för mig hur jag skulle angripa problemet och strukturera mitt program. Strukturerna växte i stället fram efterhand som jag lärde mig att använda C++ och InterViews. I ett program som det här, som hanterar Grafcet, har det objektorienterade programmeringssättet kännts naturligt.

Det är synd att det inte fanns en bra exempelsamling i InterViews referensmanual när jag började examensarbetet. När man ska lära sig ett nytt programspråk eller hur ett bibliotek används, gör man det snabbast genom att studera exempel.

Om man kan tankesättet för den objektorienterade programmeringen och känner till vilka möjligheter som finns i programspråk och klassbibliotek när man börjar att studera sitt problem, kan man angripa det på ett effektivt sätt. Den objektorienterade programmeringen ger programmeraren möjligheten att kapsla in de olika delarna av ett program på ett naturligt sätt. För att kunna utnyttja det måste man studera sitt problem noga och strukturera det innan man börjar programmera.

7. Referenser

BAYARD, G (1989): *Funktionsdiagram (GRAFCET)*, Telemecanique Svenska AB, Box 503, 64200 Flen.

IEC (1988): "Preparation of function charts for control systems," International Electrotechnical Commission, Internationell Standard IEC 848.

LINTON, MARK A. , VLISSIDES, JOHN M. , CALDER, PAUL R (1989): "Composing User Interfaces with InterViews," *IEEE Computer, vol. 22, no. 2, February 1989.*

MURATA, T (1989): "Petri Nets: Properties, Analysis and Applications," *Proceedings of the IEEE, vol. 77, no. 4, April 1989.*

STROUSTRUP, B (1991): *The C++ Programming Language, second edition*, Addison-Wesley, Reading, Mass.

Bilaga A Språk för lagring av funktionsdiagram på fil

För att kunna lagra ett funktionsdiagram på en fil har jag skapat ett språk, som talar om hur varje Grafcetsymbol ska lagras. I filen ligger symbolerna ordnade radvis, det är därför ganska enkelt att läsa filen. Gemensamt för symbolerna är att de börjar med att skriva ett "huvud" som innehåller deras identitet, efter huvudet skrivs all övrig information tex position, hur symbolerna är förbundna mm i en informationsdel. Informationsdelen består av de attribut som behövs för att symbolen ska kunna återskapas samt att ett styrsystem ska kunna skapas ur funktionsdiagrammet.

Jag ska här beskriva språket utifrån ett exempel, där funktionsdiagrammet för hissens styrfunktion är utgångspunkt, och tar slumpvis några symboler för att förklara vad de olika tecknen i raderna representerar.

```
F37 6 210 -20 210 -42 -337 -42 -337 273 -290 273 -290 260
F36 2 210 0 210 -20
F35 2 210 40 210 20
F34 2 210 60 210 40
F33 2 210 80 210 60
F32 2 210 120 210 100
F31 2 210 140 210 120
F30 2 110 60 110 40
F29 2 110 80 110 60
F28 2 110 120 110 100
F27 2 110 140 110 120
F26 2 110 160 110 140
F25 2 10 60 10 40
F24 2 10 80 10 60
F23 2 10 120 10 100
F22 2 10 140 10 120
F21 2 -90 60 -90 40
F20 2 -90 80 -90 60
F19 2 -90 120 -90 100
F18 2 -190 60 -190 40
F17 2 -190 80 -190 60
F16 2 -290 60 -290 40
E4 -270 40 -290 210 6 A 13 16 A 11 18 A 9 21 A 2 25 A 1 30 A 0 34 1 B 10 35
B10 210 10 E 4 35 A 4 36 2 10 5 sakta 0 0 0
B9 -290 90 A 15 6 A 13 7 1 2 3 upp 0 0 0
E1 -270 140 -290 -190 1 B 8 3 2 A 15 5 A 12 12
A15 -290 120 E 1 5 B 9 6 2 k2 11 till plan 2
B8 -290 170 A 14 2 E 1 3 1 1 0 0 0 0
A14 -290 200 E 0 1 B 8 2 2 p1 10 vid plan 1
E0 -270 220 -290 110 1 C 7 0 3 A 14 1 A 8 8 A 7 9
C7 -290 250 A 4 37 E 0 0 1 0 0
B6 -190 90 A 12 13 A 11 17 1 3 3 upp 0 0 0
B5 -90 170 A 8 10 E 2 14 1 4 0 0 0 0
B4 -90 90 A 10 19 A 9 20 1 5 3 upp 0 0 0
B3 10 90 A 3 23 A 2 24 1 6 3 ned 0 0 0
B2 110 90 A 6 28 A 1 29 1 8 3 ned 0 0 0
B1 110 170 A 7 11 E 3 26 1 7 0 0 0 0
B0 210 90 A 5 32 A 0 33 1 9 3 ned 0 0 0
A13 -290 60 B 9 7 E 4 16 2 p2 10 vid plan 2
A12 -190 120 E 1 12 B 6 13 2 k3 11 till plan 3
A11 -190 60 B 6 17 E 4 18 2 p3 10 vid plan 3
A10 -90 120 E 2 15 B 4 19 2 k3 11 till plan 3
A9 -90 60 B 4 20 E 4 21 2 p3 10 vid plan 3
A8 -90 200 E 0 8 B 5 10 2 p2 10 vid plan 2
```

```
A7 110 200 E 0 9 B 1 11 2 p3 10 vid plan 3
A6 110 120 E 3 27 B 2 28 2 k1 11 till plan 1
A5 210 120 E 3 31 B 0 32 2 k2 11 till plan 2
A4 210 -20 B 10 36 C 7 37 6 stanna 0
A3 10 120 E 2 22 B 3 23 2 k1 11 till plan 1
A2 10 60 B 3 24 E 4 25 2 p1 10 vid plan 1
A1 110 60 B 2 29 E 4 30 2 p1 10 vid plan 1
A0 210 60 B 0 33 E 4 34 2 p2 10 vid plan 2
F0 2 -290 240 -290 220
F1 2 -290 220 -290 200
F2 2 -290 200 -290 180
F3 2 -290 160 -290 140
F5 2 -290 140 -290 120
F6 2 -290 120 -290 100
F7 2 -290 80 -290 60
F8 2 -90 220 -90 200
F9 2 110 220 110 200
F10 2 -90 200 -90 180
F11 2 110 200 110 180
F12 2 -190 140 -190 120
F13 2 -190 120 -190 100
E2 -70 140 -90 10 1 B 5 14 2 A 10 15 A 3 22
E3 130 140 110 210 1 B 1 26 2 A 6 27 A 5 31
F14 2 -90 160 -90 140
F15 2 -90 140 -90 120
```

(A) övergång, (B) steg och (C) initialsteg

Den sista raden på föregående sida representerar en övergång. I början på raden finns övergångens huvud, A7. Det innebär att det här är övergång nummer sju. Identiteten sju finns bara till för att göra övergången unik och saknar betydelse i övrigt.

Efter huvudet kommer informationsdelen 110 200 E 0 9 B 1 11 2 p3 10 vid plan 3. 110 200 är övergångens position på den grafiska ytan. E 0 9 betyder att en alternativ förgrening med identitet 0 finns på övergångens ingång och den är förbunden till övergången via förbindelselinje 9. B 1 11 betyder att övergångens utgång består av ett steg med identiteten 1 och det är förbundet via förbindelselinje 11. De efterföljande tecknen representerar den text som är knuten till övergången. All text består av en siffra som talar om hur lång textsträngen är följt av textsträngen.

Formatet för steg och initialsteg påminner om övergångarnas format, den enda skillnaden är antalet textsträngar i slutet av raden.

(D) parallell och (E) alternativ förgrening eller sammanföring

Raden som börjar med E0 representerar en alternativ förgrening. Huvudet följs av symbolens position och omfång, x(-270), y(220), vänster(-290) och höger(110). Därefter följer 1 C 7 0, vilket betyder att symbolen har en ingång av typen initialsteg med identitet 7 förbunden med linje 0. På samma sätt följer information om symbolens utgångar, 3 A 14 1 A 8 8 A 7 9, betyder att tre övergångar med given identitet och förbundna via givna förbindningslinjer är anslutna till symbolens utgång. En parallell förgrening representeras på samma sätt, men med signalementet D i huvudet.

(F) förbindelselinje

En förbindningslinje tex den sista i filen ovan, innehåller förutom sitt huvud F15 ett tal 2 som anger antalet brytpunkter följt av brytpynkternas koordinater -90 140 och -90 120.

Bilaga B Programlistor

Här följer nu programlistor med de "headerfiler" och källkodsfiler som jag har skrivit. Det råder blandade meningar om de ska följa med i programdokumentationen, men eftersom jag vill ha examensarbetet samlat på en plats tar jag med dem som bilaga.

DESCRIPTION

Symbol is the base class for all the Grafcet symbol classes. The symbols are represented both by some graphic information, that makes it possible to redraw them, and an underlying structure with information nessesary to connect them and to simulate the graph.

Author: Sven Jonasson

CODE

```
enum SymbolType { NoType, TransitionType,
                  InitStepType, StepType,
                  ParallelType, SelectionType,
                  ConnectionType };
```

CLASS Symbol

Public members

Symbol();

Create a symbol.

void ConnectInput(Symbol* from, Symbol* connection);
void ConnectOutput(Symbol* to, Symbol* connection);

When a connectionline is drawn done the symbol points at the connected symbol and the connecting line.

void DisconnectInput(Symbol*);
void DisconnectOutput(Symbol*);

Disconnect the symbol and the connectionline to it.

void DisconnectFirstInput();
void DisconnectFirstOutput();

Disconnect the first of the three input or output pointers and decrease the input/output counter.

virtual void EditText();

Edit text. Implemented in the Grafcet classes that handles text.

Graphic* GetGraphic();
void SetGraphic(Graphic* gr);

Return the symbols graphic representation.

virtual Graphic* CreateGraphic();

Create the symbols graphic that represents the Grafcet class in question. Implemented in the Grafcet classes.

virtual Graphic* GetAdjusted(Coord);

Returns the adjusted graphic representation of a symbol. The width can only be adjusted in multi input/output classes like Parallel and Selection.

Symbol* GetNext();
void SetNext(Symbol*);
Symbol* GetPre();
void SetPre(Symbol*);

Return or set a following symbol sorted in the symbollist.

SymbolType GetType();
void SetId(int);
int GetId();

Return or set this symbols id number or return it's type..

Symbol* GetInput();
Symbol* GetInput(int);
Symbol* GetInputConnection();
Symbol* GetInputConnection(int);
Symbol* GetOutput();

Symbol* GetOutput(int);
Symbol* GetOutputConnection();
Symbol* GetOutputConnection(int);
int NrOfInputs();
int NrOfOutputs();
Returns information about the symbols that is connected to this symbol or returns how many they are.

virtual Coord GetTop();
virtual Coord GetBottom();
virtual void GetBounds(Coord&,Coord&,Coord&,Coord&);
virtual void SetBounds(Coord,Coord,Coord,Coord);
Coord GetX();
Coord GetY();
Coord GetLeft();
Coord GetRight();
void SetX(Coord xx);
void SetY(Coord yy);
void SetLeft(Coord l);
void SetRight(Coord r);
Function to handle this symbols position and witdh.

virtual void WriteOnFile(ofstream&);
Writes all information about the symbol on a file. This function is reimplemented in every Grafcet class.

virtual void GetVertices(Coord *&x, Coord *&y, int &n);
Returns the vertices on a connectionline.

virtual Transition* asTransition();
virtual Step* asStep();
virtual InitStep* asInitStep();

Protected members

SymbolType type;
int id;

Symbol* inputconnection[10];
Symbol* input[10];
int nr_of_inputs;
Symbol* outputconnection[10];
Symbol* output[10];
int nr_of_outputs;
The input points at the previous symbol in the function chart and the inputconnection points at the connection between the two symbols. To make the implementation general, all symbols have ten inputs and outputs. Therefore it's easy to change some implementation making a PetriNet editor.

Graphic *g;
Pointer to the picture that represents this symbol. If the graphic is a composite graphic object it is sorted in a picture.

Symbol* nSymbol;
Symbol* pSymbol;
Pointer to the next and previous symbol in the symbollist.

Coord left;
Coord right;
Coord x;
Coord y;

Information about the symbols position and width.

CLASS Symbollist

Symbol* first;

Public members

Symbollist();
Symbol* First();
Returns the first symbol in the list.

Symbol* GetSymbol(const Event&);
Symbol* GetSymbol(SymbolType, int);
Points at the symbol hit by a event. Nil if there was no hit.

void Into(Symbol*);
void Out(Symbol*);
Takes a symbol in or out of the list. When a symbol is taken out it is deleted.

void ClearList(Picture*);

CLASS Transition

Base class

public Symbol

Public members

Transition(int);
Transition(ifstream&);
void EditText();
Graphic* CreateGraphic();
void GetBounds(Coord&,Coord&,Coord&,Coord&);
void SetBounds(Coord,Coord,Coord,Coord);
Transition* asTransition();
Graphic* GetAdjusted(Coord);
void WriteOnFile(ofstream&);
char* transitioncondition[2];
0: Condition 1: Comment

CLASS Step

Base class

public Symbol

Public members

Step(int);
Step(ifstream&);
Coord GetBottom();
Coord GetTop();
void EditText();
Graphic* CreateGraphic();
void GetBounds(Coord&,Coord&,Coord&,Coord&);
void SetBounds(Coord,Coord,Coord,Coord);
Graphic* GetAdjusted(Coord);
void WriteOnFile(ofstream&);
Step* asStep();
char* step_text[5];
0: Step number. 1: Action. 2: Detailed command. 3: Conditional command. 4: Comment

CLASS InitStep

Base class

public Symbol

Public members

InitStep(int);
InitStep(ifstream&);
Coord GetBottom();
Coord GetTop();
void EditText();
Graphic* CreateGraphic();
void GetBounds(Coord&,Coord&,Coord&,Coord&);
void SetBounds(Coord,Coord,Coord,Coord);
Graphic* GetAdjusted(Coord);
void WriteOnFile(ofstream&);
InitStep* asInitStep();
char* initstep_text[2];
0: nr 1: comments

CLASS Parallel

Base class

public Symbol

Public members

Parallel(int);
Parallel(ifstream&);
Coord GetTop();
Coord GetBottom();
Graphic* CreateGraphic();
void GetBounds(Coord&,Coord&,Coord&,Coord&);
void SetBounds(Coord,Coord,Coord,Coord);
Graphic* GetAdjusted(Coord=999);
void WriteOnFile(ofstream&);

CLASS Selection

Base class

public Symbol

Public members

Selection(int);
Selection(ifstream&);
Graphic* CreateGraphic();
void GetBounds(Coord&,Coord&,Coord&,Coord&);
void SetBounds(Coord,Coord,Coord,Coord);
Graphic* GetAdjusted(Coord=999);
void WriteOnFile(ofstream&);

CLASS Connection

Base class

public Symbol

Public members

Connection(int nr);
Connection(ifstream&);
Graphic* CreateGraphic();
void GetVertices(Coord *&x, Coord *&y, int &n);
void SetVertices(Coord*,Coord*,int);
void WriteOnFile(ofstream&);

Private members

int n;
Coord vx[20];
Coord vy[20];
Vertices of the connection.

CLASS DECLARATIONS

class Transition
class Step
class InitStep
class Parallel
class Selection
class Connection

EXTERNAL DECLARATIONS

```
const Coord PAD_SIZE;
PFont* font;
```

DEFINED MACROS

GRAPHSYMBOL_H

INCLUDED FILES

InterViews/defs.h
InterViews/event.h
InterViews/graphic.h
InterViews/Graphic/base.h
InterViews/Graphic/picture.h
fstream.h

DESCRIPTION

This is the definition of Editor that supports the editing of Grafcet function charts. A function chart consists of both an underlying structure and a visible graphic structure. When a function chart is being edited the underlying information is stored in Grafcet symbol objects that is sorted in the symbollist. The graphic objects is saved in a picture.

A langage for saving and reading function charts on files is created and used in the read and save functions.

Author: Sven Jonasson

CLASS Editor

Base class

public GraphicBlock

Public members

Editor();

void WriteText();

Pick a symbol and edit text to it.

void MakeStep();
void MakeInitStep();
void MakeTransition();
void MakeParallel();
void MakeSelection();

Create a Grafcet symbol and place it with the mouse.

void MakeConnection();

Connect two symbols and draw a connection line.

void MoveSymbol();
void DeleteSymbol();

void New();
void SaveGraphAs();
void SaveGraph();
void ReadGraph();
void ReadIt();

boolean isEditmode();

Protected members

virtual void Handle(Event&);
boolean SlideRect(Coord& x, Coord& y,
int w, int h, Coord offx, Coord offy,
boolean move_command = false);

Creates a rubberrect with given bounds. The rubberrect can then be positioned with the mouse and when a downevent is detected the rubberrects position is returned.

boolean GrowConnection(Symbol *&from, Symbol *&to,
Coord *&x, Coord *&y, int& n);

Grows a connectionline until two Grafcet symbols is choosen. The two symbols and the connection lines vertices is then returned.

boolean Overlap(Symbol*, Symbol*);

Returns true if two symbols (parallel or selection) overlap horizontal.

void AdjustConnection(Symbol *from, Symbol *to,
Coord *&x, Coord *&y, int n);

Adjusts the end vertices on a connection line to fit the connected symbols.

void Disconnect(Symbol*);
Disconnects the symbol from it's neighbours and delets the connection lines to it.

boolean HitSymbol(Symbol *&s);
Pick a symbol with mouse. True is returned if the function picked a symbol.

boolean ChooseFile(char*,char*);
Display a filechooser and save the file name in "being_edited".

int transitionnr;
int stepnr;
int parallelnr;
int selectionnr;
int connectionnr;
Counts the symbols so they can get a identity.

Picture* p;
Symbollist* sl;
All the graphic objects is sorted in the picture and all the underlying information is stored in the Grafcet symbols in the symbollist.

boolean compiled;
True if the function chart compiled.

boolean go;
The mode we are in, controlmode if go else editmode.

char* being_edited;
The name of the function chart being edited.

CLASS DECLARATIONS
class Symbol
class Symbollist

DEFINED MACROS
EDITOR_H

INCLUDED FILES
InterViews/Graphic/ellipses.h
InterViews/graphic.h
InterViews/Graphic/grblock.h
InterViews/Graphic/picture.h
symbol.H

DESCRIPTION

The GrafcetControl class converts a functionchart to a control system. Matrixes and other logigal structures is created from the information build in the editor.

Author: Sven Jonasson

CLASS GrafcetControl

Base class

public Editor

Public members

GrafcetControl();

void Go();

If the functionchart is compiled the controlsystem shows interest in channel events. The functionchart is initialized by means of tokens being placed in all initial steps and the set of steps has got ones in all the steps that is initsteps.

void Stop();

Ignore timer and channel events and erase all tokens from the function chart.

void Handle(Event&);

Mask the comming events(if timer or channel events) and call some control functions.

void Calc();

This is the update function in the control system. For each transition that is enable and true i calculate the new set of steps and update the tokens and sends order on a channel.

boolean CheckSyntax();

Check the functioncharts syntax. Are all symbols connected and do they carry information enough to be part of the control system.

void MakeRelation();

Create the matrix that describe the movement of active steps and the matrix that knows what steps to check for a transition to be enable.

void MakeConditions();

Create the structure for the logical conditions on the transitions.

boolean isEnable(int transitionnr);

Is this transition enable.

int A(int i, int j);

Return the element in row i and col j in the incidence matrix.

Private members

Transition_vector* transition_vector;

The set of transitions.

boolean fire[100];

The transitions to fire in this loop.

VarTable* var_table;

All variables that the transitions depends on.

TimeTable* time_table;

The timetable that stores all the times for timer events to expire and what timer logic to update then.

boolean no_timer;

Is a timer ticking.

Time_logic* cur_time_logic;

The time logic to update when the next timer event arrives.

int step[100];
boolean step_changed[100];
char* step_name[100];
char* step_action[100];
The set of steps and information about them.

ActionTable* action_table;
A table with all the step actions in the functionchart. Actions that changed because of the last event is send as output.

Graphic* token[100];
Houndred tokens to place when the related step turns active.

int what[100][10];
int a[100][100];
The incidense matrix and the WHAT steps must be active for this transition to be enable matrix.

int nr_of_transitions;
int nr_of_steps;
int nr_of_initsteps;

int initkey[10];
Finds the initsteps among the steps.

CLASS DECLARATIONS
class Editor

DEFINED MACROS
GCCONTROL_H

INCLUDED FILES
comp.H
editor.H
InterViews/defs.h
InterViews/Graphic/ellipses.h

DESCRIPTION

The logical expression of a transition is build in a tree structure of objects where each object represents a logical operand or input variable.

The transition points at the root of the tree, and the input variables is represented as the trees leaves When the transition condition is tested, all group levels below the root is tested.

Author: Sven Jonasson

CLASS Logics

Base class for logics. Logics are derived from this class. Therefore they can be refered to as Logics.

Public members

Logics();

virtual boolean Is();

Returns if the Logics condition is troe or false.

virtual void Update(int);

Call this funktion when the input changes. When Update is called the whole structure below in the logic tree is updated, this is done by a recursive function.

virtual void Look_at_flanklogic(Logics*);

Let the variable look back on the root that is sensitive to edges.

virtual void Reset();

Reset the state of the logic.

CLASS One

Base class

public Logics

Logic that always is true.

Public members

One();
void Reset();
boolean Is();

CLASS And

Base class

public Logics

Public members

And(char* lstr,char* rstr,VarTable* vt,TimeTable* tt);
void Reset();
boolean Is();

Private members

Logics* leftlogic;
Logics* rightlogic;

CLASS Or

Base class

public Logics

Public members

Or(char* lstr, char* rstr,VarTable* vt,TimeTable* tt);
void Reset();
boolean Is();

Private members

Logics* leftlogic;
Logics* rightlogic;

CLASS Not

Base class

public Logics

Public members

Not(char* str,VarTable* vt,TimeTable* tt);
boolean Is();
void Reset();

Private members

Logics* logic;

CLASS Time_logic

Base class

public Logics

Logic class for time dependent transitions.

Public members

Time_logic(char*,char*,char*,VarTable*,TimeTable*);
boolean Is();
void Reset();
void Update(int state);
void Time_update();

Private members

boolean mystate;
boolean newlogicstate;
boolean oldlogicstate;
boolean timer_ticking;
int t1,t2;
Logics* logic;
TimeTable* time_table;

CLASS Up

Base class

public Logics

Public members

Up(char* str,VarTable* vt,TimeTable* tt);
boolean Is();
void Update(int);
void Reset();

Private members

Logics* logic;
boolean oldlogic;
boolean newlogic;

CLASS Down

Base class

public Logics

Public members

Down(char* str,VarTable* vt,TimeTable* tt);
boolean Is();
void Update(int);
void Reset();

Private members

Logics* logic;
boolean oldlogic;
boolean newlogic;

CLASS Var

Base class

public Logics

Variable, bottom level of the logic structure.

Public members

```
Var(char* str);
void Reset();
boolean Is();
void Update(int state);
char* GetName();
void SetNext(Var*);
Var* GetNext();
void Look_at_flanklogic(Logics*);
```

Private members

```
boolean is;
Var* next;
char* name;
Logics* flanklogics[10];
int nr_of_flanklogics;
```

DESCRIPTION

The Timer and TimeTable classes is needed to handle timedependent transitions and steps. So far i have only treated timedependence on transitions.

CLASS Timer

Class that stores the time for a timer event.

Public members

```
Timer(double,Time_logic*);
double GetTime();
Time_logic* GetTime_logic();
~Timer();
Timer* GetNext();
Timer* GetPrev();
void SetNext(Timer*);
void SetPrev(Timer*);
```

Private members

```
double time;
Time_logic* tl;
Timer* nTimer;
Timer* pTimer;
```

CLASS TimeTable

The timetable sorts the timers in a list. Next time to expire is the first timer in the list.

Public members

```
TimeTable();
~TimeTable();
void InsertTimer(int,Time_logic*);
boolean Empty();
double GetTime();
void FirstOut();
double GetNextDelay();
Time_logic* GetTime_logic_to_update();
void Reset();
```

Private members

Timer* first;

CLASS VarTable

Array of lists that contents pointers to Var objects. When a input is changed it updates the var objects referred to in the list.

Public members

VarTable();
Var* Exist(char* var_name); //returns variable if it exist else nil.
void Insert(Var*);
void Update(char* var_name,int state);
void Reset();

Private members

int nr_of_vars;
Var* first_var;

DESCRIPTION

Each element in the transition vector is related to a transition. It points at the structure that represents the transition condition.

Build_inputs creates the condition structure for all transitions.

CLASS Transition_vector

Public members

Transition_vector(int, Symbollist*, VarTable*, TimeTable*);
~Transition_vector();
boolean Is(int);

Private members

Logics* transition_vector[100];
int nr_of_transitions;

CLASS DECLARATIONS

class Transition
class TimeTable
class VarTable

DEFINED MACROS

COMP_H

INCLUDED FILES

symbol.H
InterViews/menu.h

DESCRIPTION

Controls to make symbols and for some edit commands. The controls are derived from InterViews Control class and all i had to do to specialice my own control classes was to redefine the virtual Do function and pass a interactor with the wanted visual information to the Controls constructor.

Author: Sven Jonasson

CLASS Make_Transition

Base class

public Control

Public members

Make_Transition();
void Do();
int nr;

CLASS Make_Step

Base class

public Control

Public members

Make_Step();
void Do();
int nr;

CLASS Make_InitStep

Base class

public Control

Public members

Make_InitStep();
void Do();
boolean init_exist;
int nr;

CLASS Make_Parallel

Base class

public Control

Public members

Make_Parallel();
void Do();
int nr;

CLASS Make_Selection

Base class

public Control

Public members

Make_Selection();
void Do();
int nr;

CLASS Make_Connection

Base class

public Control

Public members

Make_Connection();
void Do();
int nr;

makesymbol.H (C++)

CLASS MakeMove

Base class

public Control

Public members

MakeMove();
void Do();

CLASS MakeText

Base class

public Control

Public members

MakeText();
void Do();

EXTERNAL DECLARATIONS

GrafcetControl* gccontrol;

DEFINED MACROS

MAKESYMBOL_H

INCLUDED FILES

grafcetcontrol.H
InterViews/control.h

DESCRIPTION

GrowingConnection is used to draw a GrowingMultiLine when two symbols are connected. The connected symbols and information about the multiline passed as argument in the GetCurrent function.

Author: Sven Jonasson

CLASS GrowingConnection

Public members

GrowingConnection(Painter* p,Canvas* c);

void Track(Coord x, Coord y);

Tracks the rubberband line.

void Append(Coord x, Coord y);

Appends a vertix ending at (x, y) to the rubberband line.

void Attach(Symbol* s, Coord x, Coord y);

Attaches an end point to a symbol and updates the rubberband line by calling Append.

void GetCurrent(Symbol *&from, Symbol *&to,
Coord *&x, Coord *&y, int& n);

Returns the vertices of the multiline and erases the GrowingMultiLine. Also returns the two connected symbols.

boolean Valid();

Returns true when two symbols are connected.

Private members

Coord vx[10];
Coord vy[10];
GrowingMultiLine ml;
Symbol* conn[2];
unsigned n_conn;

DEFINED MACROS

GROWINGCONNECTION_H

INCLUDED FILES

InterViews/canvas.h
InterViews/painter.h
InterViews/rubband.h
InterViews/rubcurve.h
InterViews/rubgroup.h
InterViews/rubline.h
InterViews/rubverts.h
symbol.H

DESCRIPTION

Some functions to manipulate strings.

Author: Sven Jonasson

CODE

boolean Match(char* substr, char* str);

Returns true if the substring is inside the string.

char* cpy(const char* cstr);

Copies a const string to a string.

boolean StrEqual(char*,char*);

Returns true if two strings are equal.

char* cpy(char* str, int strstart, int strstop);

Copies a part of a string to a substring (start copy on strstart and stop on strstop).

char* ParentheseGroup(char* str);

If a string is a surrounded with matching parentheses they are removed and the string is returned without them.

DEFINED MACROS

MYSTRING_H

INCLUDED FILES

InterViews/defs.h

DESCRIPTION

Dialogboxes to edit text in Steps, InitSteps and Transitions. The Dialogboxes are derived from Dag Brick's StdDialog and made to fit these Grafcet symbols.

Author: Sven Jonasson

CLASS TransitionDialog

Base class

public StdDialog

Public members

```
TransitionDialog(char* transitioncondition[2]);
~TransitionDialog();
const char* Text(int);
```

Protected members

```
StringEditor* se[2];
ButtonState edit_state0;
ButtonState edit_state1;
```

CLASS InitStepDialog

Base class

public StdDialog

Public members

```
InitStepDialog(char* initstep_text[2]);
~InitStepDialog();
const char* Text(int);
```

Protected members

```
StringEditor* se[2];
ButtonState edit_state0;
ButtonState edit_state1;
```

CLASS StepDialog

Base class

public StdDialog

Public members

```
StepDialog(char* step_text[4]);
~StepDialog();
const char* Text(int);
```

Protected members

```
StringEditor* se[5];
ButtonState edit_state0;
ButtonState edit_state1;
ButtonState edit_state2;
ButtonState edit_state3;
ButtonState edit_state4;
```

DEFINED MACROS

GCDIALOG_H

INCLUDED FILES

stddialog.H

DESCRIPTION

Commands to be inserted in menus. The only thing that the commands is doing is call functions in Editor or GrafcetControl.

Author: Sven Jonasson

CLASS QuitCommand

Base class

public MenuItem

Public members

QuitCommand();
void Do();

CLASS ReadCommand

Base class

public MenuItem

Public members

ReadCommand();
void Do();

CLASS NewCommand

Base class

public MenuItem

Public members

NewCommand();
void Do();

CLASS SaveCommand

Base class

public MenuItem

Public members

SaveCommand();
void Do();

CLASS SaveAsCommand

Base class

public MenuItem

Public members

SaveAsCommand();
void Do();

CLASS DeleteCommand

Base class

public MenuItem

Public members

DeleteCommand();
void Do();

CLASS RelationCommand

Base class

public MenuItem

Public members

RelationCommand();
void Do();

CLASS StopCommand

Base class

public MenuItem

Public members

StopCommand();
void Do();

CLASS GoCommand

Base class

public MenuItem

Public members

GoCommand();
void Do();

EXTERNAL DECLARATIONS

GrafcetControl* gccontrol;

DEFINED MACROS

COMMAND_H

INCLUDED FILES

grafcetcontrol.H
InterViews/control.h
InterViews/menu.h

```
#include <command.H>
#include <comp.H>
#include <editor.H>
#include <grafcetcontrol.H>
#include <grafcetnorm.H>
#include <symbol.H>
#include <makesymbol.H>

#include <InterViews/box.h>
#include <InterViews/frame.h>
#include <InterViews/glue.h>
#include <InterViews/Graphic/ppaint.h>
#include <InterViews/menu.h>
#include <InterViews/painter.h>
#include <InterViews/world.h>

// Global variabels.
GrafcetControl* gccontrol;
World* InterViews_world;
HBox* hbox;
const Coord PAD_SIZE = 350;
PFont* font;


PropertyData props[] = {
  {"*font", "*helvetica-bold-r*14*"},
  { nil }
};


OptionDesc options[] = {
  { nil }
};


void main (int argc, char** argv) {

  InitPPaint();

  InterViews_world = new World("pgm", props, options, argc, argv);

  font = new PFont("*helvetica-medium-r*8*");

  VBox* vbox1 = new VBox;
  VBox* vbox2 = new VBox;

  gccontrol = new GrafcetControl;

  MenuBar* menubar = new MenuBar;
  PulldownMenu* menu1 = new PulldownMenu("File ");

  menu1->Include(new NewCommand);
  menu1->Include(new ReadCommand);
  menu1->Include(new SaveCommand);
  menu1->Include(new SaveAsCommand);
  menu1->Include(new QuitCommand);
  menubar->Include(menu1);

  PulldownMenu* menu2 = new PulldownMenu("Edit ");

  menu2->Include(new DeleteCommand);
  menubar->Include(menu2);

  PulldownMenu* menu3 = new PulldownMenu("Sim ");
  menu3->Include(new StopCommand);
  menu3->Include(new GoCommand);
  menubar->Include(menu3);

  vbox2->Insert(menubar);
  vbox2->Insert(new Frame(gccontrol));

  vbox1->Insert(new VGlue(round(.5*cm)));
  vbox1->Insert(new MakeText);
  vbox1->Insert(new VGlue(round(.5*cm)));
  vbox1->Insert(new MakeMove);
  vbox1->Insert(new VGlue(round(.5*cm)));
  vbox1->Insert(new Make_Connection());
  vbox1->Insert(new VGlue(round(.5*cm)));
  vbox1->Insert(new Make_InitStep());
  vbox1->Insert(new VGlue(round(.5*cm)));
  vbox1->Insert(new Make_Step());
  vbox1->Insert(new VGlue(round(.5*cm)));
  vbox1->Insert(new Make_Transition());
  vbox1->Insert(new VGlue(round(.5*cm)));
  vbox1->Insert(new Make_Parallel());
  vbox1->Insert(new VGlue(round(.5*cm)));
  vbox1->Insert(new Make_Selection());
  vbox1->Insert(new VGlue(round(.5*cm)));

  hbox = new HBox;
  hbox->Insert(new Frame(vbox1));
  hbox->Insert(vbox2);

  InterViews_world->InsertApplication(new Frame(hbox));
  gccontrol->Run();
}
```

```
#include <gcdialog.H>
#include <symbol.H>
#include <mystring.H>
#include <stddialog.H>
#include <type2sign.H>

#include <fstream.h>
#include <stream.h>
#include <string.h>

#include <InterViews/canvas.h>
#include <InterViews/painter.h>
#include <InterViews/graphic.h>

#include <InterViews/Graphic/label.h>
#include <InterViews/Graphic/lines.h>
#include <InterViews/Graphic/picture.h>
#include <InterViews/Graphic/polygons.h>
////////////////////////////////////////////////////////
Symbol :: Symbol()
{
  type = NoType; // Symboltype
  id = 0;
  x = 0;
  y = 0;
  nr_of_inputs = 0;
  nr_of_outputs = 0;
  nSymbol = nil; /* next symbol in the symbollist */
  pSymbol = nil; /* previous symbol in the symbollist */
}


void Symbol :: ConnectInput(Symbol* from, Symbol* connection)
{
  input[nr_of_inputs] = from;
  inputconnection[nr_of_inputs] = connection;
  nr_of_inputs++;
}
void Symbol :: ConnectOutput(Symbol* to, Symbol* connection)
{
  output[nr_of_outputs] = to;
  outputconnection[nr_of_outputs] = connection;
  nr_of_outputs++;
}



// When we disconnect a symbol, the two pointers
// to the symbol and it's connection, are set to nil.
// The last active pointer takes the index of the one
// that was deactivated.
void Symbol :: DisconnectInput(Symbol* s)
{
  int nr = 0;
  boolean found = false;
  while (nr_of_inputs > nr && !found ) {
    if ( s == input[nr] ) {
      found = true;
      input[nr] = input[nr_of_inputs - 1];
      inputconnection[nr] = inputconnection[nr_of_inputs - 1];
      input[nr_of_inputs - 1] = nil;
      inputconnection[nr_of_inputs - 1] = nil;
      nr_of_inputs--;
    }
    else {
      nr++;
    }
  }
}


void Symbol :: DisconnectOutput(Symbol* s)
{
  int nr = 0;
  boolean found = false;
  while (nr_of_outputs > nr && !found ) {
    if ( s == output[nr] ) {
      found = true;
      output[nr] = output[nr_of_outputs - 1];
      outputconnection[nr] = outputconnection[nr_of_outputs - 1];
      output[nr_of_outputs - 1] = nil;
      outputconnection[nr_of_outputs - 1] = nil;
      nr_of_outputs--;
    }
    else {
      nr++;
    }
  }
}


void Symbol :: DisconnectFirstInput()
{
  input[0] = input[nr_of_inputs - 1];
  inputconnection[0] = inputconnection[nr_of_inputs - 1];
  input[nr_of_inputs - 1] = nil;
  inputconnection[nr_of_inputs - 1] = nil;
  nr_of_inputs--;
}


void Symbol :: DisconnectFirstOutput()
{
  outputconnection[0] = outputconnection[nr_of_outputs - 1];
  output[0] = output[nr_of_outputs - 1];
  outputconnection[nr_of_outputs - 1] = nil;
  output[nr_of_outputs - 1] = nil;
  nr_of_outputs--;
}


void Symbol :: EditText()
{} // Redefined in the inheriting classes step, transition etc.


// Returns the graphic related to the symbol.
Graphic* Symbol :: GetGraphic() { return g; }
void Symbol :: SetGraphic(Graphic* gr) { g = gr; }


Graphic* Symbol :: CreateGraphic()
{
  cerr << "Symbol::CreateGraphic()" << endl;
  return nil;
}


Graphic* Symbol :: GetAdjusted(Coord)
{
  cerr << "Symbol::GetAdjusted(int)\n";
```

```
  return nil;
}


// Return or set the neighbour symbol in the symbollist.
// These four funktions are used in the symbollist class.
Symbol* Symbol :: GetNext() { return nSymbol; }
Symbol* Symbol :: GetPre() { return pSymbol; }
void Symbol :: SetNext(Symbol* s) { nSymbol = s; }
void Symbol :: SetPre(Symbol* s) { pSymbol = s; }


SymbolType Symbol :: GetType() { return type; }


int Symbol :: GetId() { return id; }
void Symbol :: SetId(int i) { id = i; }


//Returns a connected symbol, if there is one.
Symbol* Symbol :: GetOutput() { return output[0]; }
Symbol* Symbol :: GetOutput(int i) { return output[i]; }
Symbol* Symbol :: GetInput() { return input[0]; }
Symbol* Symbol :: GetInput(int i) { return input[i]; }



// Returns the related connection.
Symbol* Symbol :: GetOutputConnection() { return outputconnection[0]; }
Symbol* Symbol :: GetOutputConnection(int i) { return outputconnection[i]; }
Symbol* Symbol :: GetInputConnection() { return inputconnection[0]; }
Symbol* Symbol :: GetInputConnection(int i) { return inputconnection[i]; }



// Returns how many inputs or outputs there is.
int Symbol :: NrOfInputs() { return nr_of_inputs; }
int Symbol :: NrOfOutputs() { return nr_of_outputs; }


Coord Symbol :: GetX() { return x ;}
Coord Symbol :: GetY() { return y; }
Coord Symbol :: GetBottom() { return y ;}
Coord Symbol :: GetTop() { return y ;}

Coord Symbol :: GetLeft() { return left;}
Coord Symbol :: GetRight() { return right;}

void Symbol :: SetX(Coord xx) { x = xx;}
void Symbol :: SetY(Coord yy) { y = yy;}
void Symbol :: SetLeft(Coord l) { left = l;}
void Symbol :: SetRight(Coord r) { right = r;}

void Symbol :: GetVertices(Coord *&, Coord *&, int &) {}
void Symbol :: GetBounds(Coord&,Coord&,Coord&,Coord&) {}
void Symbol :: SetBounds(Coord,Coord,Coord,Coord) {}

void Symbol :: WriteOnFile(ofstream&) {}


Transition* Symbol :: asTransition()
{
  cerr << "error";
  return nil;
}
```

```
Step* Symbol :: asStep()
{
  cerr << "error";
  return nil;
}
InitStep* Symbol :: asInitStep()
{
  cerr << "error";
  return nil;
}


////////////////////////////////////////////////////////////////
Symbollist :: Symbollist()
{
  first = nil;
}


Symbol* Symbollist :: First()
{
  return first;
}


void Symbollist :: Into(Symbol* s)
{
  s->SetNext(first);
  if ( first != nil )
    first->SetPre(s);
  first = s;
}


void Symbollist :: Out(Symbol* s)
{
  // Find the symbol. Take it out of the list and
  // delete it.
  Symbol* tmp = first;
  boolean found = false;
  while ( !found && tmp != nil ) {

    if( s == tmp ) {
      found = true;
      if ( tmp->GetNext() != nil )
        // Not the last symbol in the symbollist.
        tmp->GetNext()->SetPre(tmp->GetPre());
      if ( tmp->GetPre() != nil )
        tmp->GetPre()->SetNext(tmp->GetNext());
      else
        first = tmp->GetNext();
    }
    else
      tmp = tmp->GetNext();
  }
  if (tmp != nil)
    delete tmp;
}


Symbol* Symbollist :: GetSymbol(const Event& e)

{
  boolean found = false;
  Symbol* tmp = first;
```

```
  BoxObj bo(e.x - 2, e.y - 2, e.x + 2, e.y + 2);
  while ( tmp != nil && !found ) {
    if ( tmp->GetGraphic()->Intersects(bo) )
      found = true;
    else
      tmp = tmp->GetNext();
  }
  return tmp;
}


Symbol* Symbollist :: GetSymbol(SymbolType st, int ident)
{
  boolean found = false;
  Symbol* tmp = first;
  while ( tmp != nil && !found ) {
    if ( tmp->GetType() == st && tmp->GetId() == ident )
      found = true;
    else
      tmp = tmp->GetNext();
  }
  return tmp;
}


void Symbollist :: ClearList(Picture* pict)
{
  Symbol* tmp = First();
  while ( tmp != nil ) {
    pict->Remove( tmp->GetGraphic() );
    Out(tmp);
    delete tmp;
    tmp = First();
  }
}


////////////////////////////////////////////////////////////////////
// Helps the graphbuilder to get the symbols
// right.

Coord AdjustX(Coord x)
{
  x % 20  > 10 ? x = x + x % 20 : x = x - x % 20;
  return x;
}

Coord AdjustY(Coord y)
{
  y % 10  > 5 ? y = y + y % 10 : y = y - y % 10;
  return y;
}



////////////////////////////////////////////////////////////
Step :: Step(int nr) : Symbol()
{
  type = StepType;
  id = nr;
  for (int i = 0; i < 5; ++i)
    step_text[i] = "\0";
}
```

```
Step :: Step(ifstream& ifstr) : Symbol()
{
  nr_of_inputs = 0;
  nr_of_outputs = 0;
  type = StepType;
  int texttest;
  char osign,isign; //
  int iid,oid,icid,ocid;
  ifstr >> id >> x >> y;
  ifstr >> isign >> iid >> icid;
  ifstr >> osign >> oid >> ocid;
  for (int i=0;i<5;++i) {
    ifstr >> texttest;
    if ( texttest > 0 ) {
      ifstr.get();              // skip space after length
      step_text[i] = new char[texttest+1];
      ifstr.get(step_text[i], texttest+1, '\0');
    }
    else
      step_text[i] = "\0";
  }
}


Coord Step :: GetBottom() { return y-10; }
Coord Step :: GetTop() { return y+10; }

void Step :: EditText()
{
  StepDialog* sd = new StepDialog(step_text);
  int state = sd->Display();
  if (state == 2)
    for (int i=0;i<5;++i) {
      step_text[i] = cpy(sd->Text(i));
    }
  delete sd;
}


Graphic* Step :: CreateGraphic()
{
  Coord xdiff = 0;
  Picture* pict = new Picture();
  Rect* r0 = new Rect(x - 15, y - 10, x + 15, y + 10);
  r0->SetBrush(psingle);
  pict->Append(r0);
  if ( strlen(step_text[0]) > 0 ) {
    Label* l0 = new Label(step_text[0],strlen(step_text[0]));
    l0->SetFont(font);
    r0->Align(Center,l0,Center);
    pict->Append(l0);
  }

  if ( strlen(step_text[1]) > 0 ) {
    if ( strlen(step_text[2]) > 0 ) {
      xdiff = xdiff + 20;
      Rect* r1 = new Rect(x + 25, y - 10, x + 45, y + 10);
      Label* l1 = new Label(step_text[2],strlen(step_text[2]));
      r1->SetBrush(psingle);
      l1->SetFont(font);
      r1->Align(Center,l1,Center);
      pict->Append(r1);
      pict->Append(l1);
    }
```

```
    Line* line1 = new Line(x + 15, y, x + 25, y);
    line1->SetBrush(psingle);
    pict->Append(line1);
    Rect* r2 = new Rect(x + 25 + xdiff, y - 10,
                        x + 40 + strlen(step_text[1])*8 + xdiff, y + 10);
    Label* l2 = new Label(step_text[1],strlen(step_text[1]));
    l2->SetFont(font);
    r2->SetBrush(psingle);
    r2->Align(Center,l2,Center);
    pict->Append(r2);
    pict->Append(l2);
  }
  else if ( strlen(step_text[4]) > 0 ) {
    Label* l3 = new Label(" '",2);
    Label* l4 = new Label(step_text[4],
                          strlen(step_text[4]));
    Label* l5 = new Label("'",1);
    l3->SetFont(font);
    l4->SetFont(font);
    l5->SetFont(font);
    r0->Align(CenterRight,l3,CenterLeft);
    l3->Align(CenterRight,l4,CenterLeft);
    l4->Align(CenterRight,l5,CenterLeft);
    pict->Append(l3);
    pict->Append(l4);
    pict->Append(l5);
  }

  g = pict;
  return pict;
}


Graphic* Step :: GetAdjusted(Coord)
{
  return CreateGraphic();
}


void Step :: GetBounds(Coord& xx,
                       Coord& yy,
                       Coord& w,
                       Coord& h)
{
  xx = x + PAD_SIZE;
  yy = y + PAD_SIZE;
  h = 20;
  w = 30;
}


void Step :: SetBounds(Coord xx,
                       Coord yy,
                       Coord ,
                       Coord )
{
  x = AdjustX(xx) - PAD_SIZE;
  y = AdjustY(yy) - PAD_SIZE;
}


void Step :: WriteOnFile(ofstream& curFile)
{
  curFile.put('B'); // B is the signalement for a step.
  curFile << id << " ";
  curFile << x << " " << y << " ";
  if ( nr_of_inputs > 0 ) {
       curFile << Type2Signalement(input[0]->GetType()) << " ";
       curFile << input[0]->GetId() << " ";
       curFile << inputconnection[0]->GetId() << " ";
     }
  else {
    curFile << 'G' << " ";
    curFile << 0 << " " << 0 << " ";
  }
  if ( nr_of_outputs > 0 ) {
    curFile << Type2Signalement(output[0]->GetType()) << " ";
    curFile << output[0]->GetId() << " ";
    curFile << outputconnection[0]->GetId() << " ";
  }
  else {
    curFile << 'G' << " ";
    curFile << 0 << " " << 0 << " ";
  }
  for (int i=0; i<5; ++i) {
    if ( strlen(step_text[i]) > 0 ) {
      curFile << strlen(step_text[i]) << " " << step_text[i] << " ";
    }
    else
      curFile << 0 << " ";
  }
  curFile << endl;
}


Step* Step :: asStep() { return this; }

////////////////////////////////////////////////////////////

InitStep :: InitStep(int nr) : Symbol()
{
  id = nr;
  type = InitStepType;
  initstep_text[0] = "\0"; // StepNr is the visual number.
  initstep_text[1] = "\0";
}

InitStep :: InitStep(ifstream& ifstr) : Symbol()
{
  nr_of_inputs = 0;
  nr_of_outputs = 0;
  type = InitStepType;
  int texttest;
  char osign,isign; //
  int iid,oid,icid,ocid;
  ifstr >> id >> x >> y;
  ifstr >> isign >> iid >> icid;
  ifstr >> osign >> oid >> ocid;
  for (int i=0;i<2;++i) {
    ifstr >> texttest;
    if ( texttest > 0 ) {
      ifstr.get();              // skip space after length
      initstep_text[i] = new char[texttest+1];
      ifstr.get(initstep_text[i], texttest+1, '\0');
    }
    else
      initstep_text[i] = "\0";
  }
}
```

```
Coord InitStep :: GetBottom() { return y-10; }
Coord InitStep :: GetTop() { return y+10; }


void InitStep :: EditText()
{
  InitStepDialog* isd = new InitStepDialog(initstep_text);
  int state = isd->Display();
  if (state == 2)
    for (int i=0;i<2;++i) {
      initstep_text[i] = cpy(isd->Text(i));
    }
  delete isd;
}


Graphic*  InitStep :: CreateGraphic()
{
  Rect* r1 = new Rect(x - 15, y - 10, x + 15, y + 10);
  Rect* r2 = new Rect(x - 13, y - 8, x + 13, y + 8);

  r1->SetBrush(psingle);
  r2->SetBrush(psingle);

  Picture* pict = new Picture();
  pict->Append(r1);
  pict->Append(r2);

  if ( strlen(initstep_text[0]) > 0 ) {
    Label* l = new Label(initstep_text[0],strlen(initstep_text[0]));
    l->SetFont(font);
    r1->Align(Center,l,Center);
    pict->Append(l);
  }
  if ( strlen(initstep_text[1]) > 0 ) {
    Label* l1 = new Label(initstep_text[1],strlen(initstep_text[1]));
    Label* l2 = new Label("  '",3);
    Label* l3 = new Label("'",1);
    l1->SetFont(font);
    l2->SetFont(font);
    l3->SetFont(font);
    r1->Align(CenterRight,l2,CenterLeft);
    l2->Align(CenterRight,l1,CenterLeft);
    l1->Align(CenterRight,l3,CenterLeft);
    pict->Append(l1);
    pict->Append(l2);
    pict->Append(l3);
  }
  g = pict;
  return pict;
}


Graphic* InitStep :: GetAdjusted(Coord)
{
  return CreateGraphic();
}


void InitStep :: GetBounds(Coord& xx,
                           Coord& yy,
                           Coord& w,
                           Coord& h)
{
  xx = x + PAD_SIZE;
  yy = y + PAD_SIZE;
  h = 20;
  w = 30;
}


void InitStep :: SetBounds(Coord xx,
                           Coord yy,
                           Coord ,
                           Coord )
{
  x = AdjustX(xx) - PAD_SIZE;
  y = AdjustY(yy) - PAD_SIZE;
}


void InitStep :: WriteOnFile(ofstream& curFile)
{
  curFile.put('C'); // C is the signalement for a initstep.
  curFile << id << " ";
  curFile << x << " " << y << " ";
  if ( nr_of_inputs > 0 ) {
    curFile << Type2Signalement(input[0]->GetType()) << " ";
    curFile << input[0]->GetId() << " ";
    curFile << inputconnection[0]->GetId() << " ";
  }
  else {
    curFile << 'G' << " ";
    curFile << 0 << " " << 0 << " ";
  }
  if ( nr_of_outputs > 0 ) {
    curFile << Type2Signalement(output[0]->GetType()) << " ";
    curFile << output[0]->GetId() << " ";
    curFile << outputconnection[0]->GetId() << " ";
  }
  else {
    curFile << 'G' << " ";
    curFile << 0 << " " << 0 << " ";
  }
  for (int i=0; i<2; ++i) {
    if ( strlen(initstep_text[i]) > 0 ) {
      curFile << strlen(initstep_text[i]) << " " << initstep_text[i] << " ";
    }
    else
      curFile << 0 << " ";
  }
  curFile << endl;
}


InitStep* InitStep :: asInitStep()
{
  return this;
}


////////////////////////////////////////////////////////////
Transition :: Transition(int nr) : Symbol()
{
  type = TransitionType;
  id = nr;
  x = 0;
  y = 0;
  transitioncondition[0] = "\0";
```

```
  transitioncondition[1] = "\0";
}


Transition :: Transition(ifstream& ifstr) : Symbol()
{
  nr_of_inputs = 0;
  nr_of_outputs = 0;
  type = TransitionType;
  int texttest;
  char osign,isign; //
  int iid,oid,icid,ocid;
  ifstr >> id >> x >> y;
  ifstr >> isign >> iid >> icid;
  ifstr >> osign >> oid >> ocid;
  for (int i=0; i<2; ++i) {
    ifstr >> texttest;
    if ( texttest > 0 ) {
      ifstr.get();              // skip space after length
      transitioncondition[i] = new char[texttest+1];
      ifstr.get(transitioncondition[i], texttest+1, '\0');
    }
    else
      transitioncondition[i] = "\0";
  }
}


void Transition :: EditText()
{
  TransitionDialog* td;
  td = new TransitionDialog(transitioncondition);
  int state = td->Display();
  if ( state == 2)
    for (int i=0; i<2; ++i)
      transitioncondition[i] = cpy(td->Text(i));
  delete td;
}


Graphic*  Transition :: CreateGraphic()
{
  Picture* pict = new Picture();
  Line* l = new Line(x - 5, y , x + 5, y);
  l->SetBrush(psingle);
  pict->Append(l);
  if ( strlen(transitioncondition[0]) > 0 ) {
    Label* lab1 = new Label(" ",1);
    Label* lab2 = new Label(transitioncondition[0],
                            strlen(transitioncondition[0]));
    lab1->SetFont(font);
    lab2->SetFont(font);
    l->Align(CenterRight,lab1,CenterLeft);
    lab1->Align(CenterRight,lab2,CenterLeft);
    pict->Append(lab1);
    pict->Append(lab2);
    if ( strlen(transitioncondition[1]) > 0 ) {
      Label* lab3 = new Label(" '",2);
      Label* lab4 = new Label(transitioncondition[1],
                              strlen(transitioncondition[1]));
      Label* lab5 = new Label("'",1);
      lab3->SetFont(font);
      lab4->SetFont(font);
      lab5->SetFont(font);
      lab2->Align(CenterRight,lab3,CenterLeft);
      lab3->Align(CenterRight,lab4,CenterLeft);
      lab4->Align(CenterRight,lab5,CenterLeft);
      pict->Append(lab3);
      pict->Append(lab4);
      pict->Append(lab5);
    }
  }
  g = pict;
  return pict;
}


Graphic* Transition :: GetAdjusted(Coord)
{
  return CreateGraphic();
}


void Transition :: GetBounds(Coord& xx,
                             Coord& yy,
                             Coord& w,
                             Coord& h)
{
  xx = x + PAD_SIZE;
  yy = y + PAD_SIZE;
  h = 0;
  w = 10;
}


void Transition :: SetBounds(Coord xx,
                             Coord yy,
                             Coord ,
                             Coord )
{
  x = AdjustX(xx) - PAD_SIZE;
  y = AdjustY(yy) - PAD_SIZE;
}


void Transition :: WriteOnFile(ofstream& curFile)
{
  curFile.put('A'); // A is the signalement for a transition.
  curFile << id << " ";
  curFile << x << " " << y << " ";
  if ( nr_of_inputs > 0 ) {
    curFile << Type2Signalement(input[0]->GetType()) << " ";
    curFile << input[0]->GetId() << " ";
    curFile << inputconnection[0]->GetId() << " ";
  }
  else {
    curFile << 'G' << " ";
    curFile << 0 << " " << 0 << " ";
  }
  if ( nr_of_outputs > 0 ) {
    curFile << Type2Signalement(output[0]->GetType()) << " ";
    curFile << output[0]->GetId() << " ";
    curFile << outputconnection[0]->GetId() << " ";
  }
  else {
    curFile << 'G' << " ";
    curFile << 0 << " " << 0 << " ";
  }
  for (int i=0; i<2; ++i) {
    if (  strlen(transitioncondition[i]) > 0 ) {
```

```
      curFile << strlen(transitioncondition[i])
        << " " << transitioncondition[i] << " ";
    }
    else
      curFile << 0 << " ";
  }
  curFile << endl;
}


Transition* Transition :: asTransition()
{
  return this;
}


////////////////////////////////////////////////////////
Parallel :: Parallel(int nr) : Symbol()
{
  type = ParallelType;
  id = nr;
  x = 0;
  y = 0;
  left = x - 30;
  right = x + 30;
}


Parallel :: Parallel(ifstream& ifstr) : Symbol()
{
  nr_of_inputs = 0;
  nr_of_outputs = 0;
  type = ParallelType;
  int iid,icid,oid,ocid,nr;
  char isign,osign;
  ifstr >> id >> x >> y >> left >> right;
  nr_of_inputs = 0;
  nr_of_outputs = 0;
  ifstr >> nr;
  for (int i=0; i<nr; ++i)
    ifstr >> isign >> iid >> icid;
  ifstr >> nr;
  for (i=0; i<nr; ++i)
    ifstr >> osign >> oid >> ocid;
}


Coord Parallel :: GetTop() { return y+1; }
Coord Parallel :: GetBottom() { return y-1; }


Graphic*  Parallel :: CreateGraphic()
{
  Line* g1 = new Line(left, y + 1, right, y + 1);
  Line* g2 = new Line(left, y - 1, right, y - 1);
  g1->SetBrush(psingle);
  g2->SetBrush(psingle);
  Picture* pict = new Picture();
  pict->Append(g1);
  pict->Append(g2);
  g = pict;
  return pict;
}
```

```
void Parallel :: GetBounds(Coord& xx,
                           Coord& yy,
                           Coord& w,
                           Coord& h)
{
  xx = x + PAD_SIZE;
  yy = y + PAD_SIZE;
  h = 2;
  w = right - left;
}


void Parallel :: SetBounds(Coord xx,
                           Coord yy,
                           Coord w,
                           Coord )
{
  x = AdjustX(xx) - PAD_SIZE;
  y = AdjustY(yy) - PAD_SIZE;
  left = x - w/2;
  right = x + w/2;
}


Graphic* Parallel :: GetAdjusted(Coord newx)
{
  if ( newx == 999 ) {
    Coord xmin=0;
    Coord xmax=0;
    if ( nr_of_inputs != 0 || nr_of_outputs != 0 ) {
      xmin = PAD_SIZE;
      xmax = -PAD_SIZE;
      for (int i=0; i < nr_of_inputs; ++i) {
        xmin = xmin < input[i]->GetX() ? xmin : input[i]->GetX();
        xmax = xmax > input[i]->GetX() ? xmax : input[i]->GetX();
      };
      for (i=0; i < nr_of_outputs; ++i) {
        xmin = xmin < output[i]->GetX() ? xmin : output[i]->GetX();
        xmax = xmax > output[i]->GetX() ? xmax : output[i]->GetX();
      }
    }
    if ( xmin != xmax ) {
      left = xmin;
      right = xmax;
    }
    else if ( xmin != 0 ) {
      left  = xmin - 30;
      right = xmin + 30;
    }
  }
  else {
    Coord xmax = -PAD_SIZE;
    Coord xmin = PAD_SIZE;
    for (int i=0; i < nr_of_inputs; ++i) {
      xmin = xmin < input[i]->GetX() ? xmin : input[i]->GetX();
      xmax = xmax > input[i]->GetX() ? xmax : input[i]->GetX();
    };
    for (i=0; i < nr_of_outputs; ++i) {
      xmin = xmin < output[i]->GetX() ? xmin : output[i]->GetX();
      xmax = xmax > output[i]->GetX() ? xmax : output[i]->GetX();
    }

    right = newx > xmax ? newx : xmax;
    left = newx < xmin ? newx : xmin;
    if ( left == right ) {
```

```
      left = left - 30;
      right = right + 30;
    }
  }
  return CreateGraphic();
}


void Parallel :: WriteOnFile(ofstream& curFile)
{
  curFile.put('D');
  curFile << id << " ";
  curFile << x << " " << y << " ";
  curFile << left << " " << right << " ";
  curFile << nr_of_inputs << " ";
  for (int i=0; i < nr_of_inputs; ++i) {
    curFile << Type2Signalement(input[i]->GetType()) << " ";
    curFile << input[i]->GetId() << " ";
    curFile << inputconnection[i]->GetId() << " ";
  }
  curFile << nr_of_outputs << " ";
  for (i=0; i < nr_of_outputs; ++i) {
    curFile << Type2Signalement(output[i]->GetType()) << " ";
    curFile << output[i]->GetId() << " ";
    curFile << outputconnection[i]->GetId() << " ";
  }
  curFile << endl;
}


////////////////////////////////////////////////////////

Selection :: Selection(int nr) : Symbol()
{
  type = SelectionType;
  id = nr;
  left = 0;
  right = 0;
}


Selection :: Selection(ifstream& ifstr) : Symbol()
{
  nr_of_inputs = 0;
  nr_of_outputs = 0;
  type = SelectionType;
  int iid,icid,oid,ocid,nr;
  char isign,osign;
  ifstr >> id >> x >> y >> left >> right;
  ifstr >> nr;
  for (int i=0; i<nr; ++i)
    ifstr >> isign >> iid >> icid;
  ifstr >> nr;
  for (i=0; i<nr; ++i)
    ifstr >> osign >> oid >> ocid;
}


Graphic* Selection :: CreateGraphic()
{
  Line* l = new Line(left, y, right, y);
  l->SetBrush(psingle);
  g = l;
  return l;
}
```

```
void Selection :: GetBounds(Coord& xx,
                            Coord& yy,
                            Coord& w,
                            Coord& h)
{
  xx = x + PAD_SIZE;
  yy = y + PAD_SIZE;
  h = 2;
  w = right - left;
}


void Selection :: SetBounds(Coord xx,
                            Coord yy,
                            Coord w,
                            Coord )
{
  x = AdjustX(xx) - PAD_SIZE;
  y = AdjustY(yy) - PAD_SIZE;
  left = x - w/2;
  right = x + w/2;
}


Graphic* Selection :: GetAdjusted(Coord newx)
{
  if ( newx == 999 ) {
    Coord xmin=0;
    Coord xmax=0;
    if ( nr_of_inputs != 0 || nr_of_outputs != 0 ) {
      xmin = PAD_SIZE;
      xmax = -PAD_SIZE;
      for (int i=0; i < nr_of_inputs; ++i) {
        xmin = xmin < input[i]->GetX() ? xmin : input[i]->GetX();
        xmax = xmax > input[i]->GetX() ? xmax : input[i]->GetX();
      };
      for (i=0; i < nr_of_outputs; ++i) {
        xmin = xmin < output[i]->GetX() ? xmin : output[i]->GetX();
        xmax = xmax > output[i]->GetX() ? xmax : output[i]->GetX();
      }
    }
    if ( xmin != xmax ) {
      left = xmin;
      right = xmax;
    }
    else if ( xmin != 0 ) {
      left  = xmin - 30;
      right = xmin + 30;
    };
  }
  else {
    Coord xmax = -PAD_SIZE;
    Coord xmin = PAD_SIZE;

    for (int i=0; i < nr_of_inputs; ++i) {
      xmin = xmin < input[i]->GetX() ? xmin : input[i]->GetX();
      xmax = xmax > input[i]->GetX() ? xmax : input[i]->GetX();
    };
    for (i=0; i < nr_of_outputs; ++i) {
      xmin = xmin < output[i]->GetX() ? xmin : output[i]->GetX();
      xmax = xmax > output[i]->GetX() ? xmax : output[i]->GetX();
    }
```

```
    right = newx > xmax ? newx : xmax;
    left = newx < xmin ? newx : xmin;

    if ( left == right ) {
      left = left - 30;
      right = right + 30;
    }
  }
  return CreateGraphic();
}


void Selection :: WriteOnFile(ofstream& curFile)
{
  curFile.put('E');
  curFile << id << " ";
  curFile << x << " " << y << " ";
  curFile << left << " " << right << " ";
  curFile << nr_of_inputs << " ";
  for (int i=0; i < nr_of_inputs; ++i) {
    curFile << Type2Signalement(input[i]->GetType()) << " ";
    curFile << input[i]->GetId() << " ";
    curFile << inputconnection[i]->GetId() << " ";
  }
  curFile << nr_of_outputs << " ";
  for (i=0; i < nr_of_outputs; ++i) {
    curFile << Type2Signalement(output[i]->GetType()) << " ";
    curFile << output[i]->GetId() << " ";
    curFile << outputconnection[i]->GetId() << " ";
  }
  curFile << endl;
}


////////////////////////////////////////////////////////

Connection :: Connection(int nr) : Symbol()
{
  type = ConnectionType;
  id = nr;
}


Connection :: Connection(ifstream& ifstr) : Symbol()
{
  type = ConnectionType;
  ifstr >> id;
  ifstr >> n; // Number of vertices.
  for (int i=0; i < n; ++i)
    ifstr >> vx[i] >> vy[i];
}


Graphic* Connection :: CreateGraphic()
{
  MultiLine* ml = new MultiLine(vx, vy, n);
  ml->SetBrush(psingle);
  g = ml;
  return ml;
}


void Connection :: GetVertices(Coord *&xx, Coord *&yy, int &nn)
{
  nn = n;
  for ( int i = 0 ; i < n ; ++ i ) {
    xx[i]=vx[i];
    yy[i]=vy[i];
  }
}


void Connection :: SetVertices(Coord* xx,Coord* yy,int nn)
{
  n = nn;
  for ( int i = 0 ; i < n ; ++ i ) {
    vx[i]=xx[i];
    vy[i]=yy[i];
  }
}


void Connection :: WriteOnFile(ofstream& curFile)
{
  curFile.put('F');
  curFile << id << " ";
  curFile << n << " "; // Number of vertices.
  for (int i=0; i < n; ++i) {
    curFile << vx[i] << " " << vy[i] << " ";
  }
  curFile << endl;
}
```

```
#include <InterViews/canvas.h>
#include <InterViews/filechooser.h>
#include <InterViews/Graphic/grblock.h>
#include <InterViews/Graphic/label.h>
#include <InterViews/Graphic/lines.h>
#include <InterViews/Graphic/persistent.h>
#include <InterViews/Graphic/pfile.h>
#include <InterViews/Graphic/picture.h>
#include <InterViews/Graphic/polygons.h>
#include <InterViews/Graphic/ppaint.h>
#include <InterViews/Graphic/rasterrect.h>
#include <InterViews/Graphic/splines.h>
#include <InterViews/Graphic/stencil.h>
#include <InterViews/menu.h>
#include <InterViews/message.h>
#include <InterViews/painter.h>
#include <InterViews/rubband.h>
#include <InterViews/rubcurve.h>
#include <InterViews/rubgroup.h>
#include <InterViews/rubline.h>
#include <InterViews/rubrect.h>
#include <InterViews/rubverts.h>
#include <InterViews/sensor.h>
#include <libc.h>
#include <string.h>
#include <fstream.h>
#include <iostream.h>


#include <editor.H>
#include <grafcetnorm.H>
#include <growingconnection.H>
#include <mystring.H>
#include <stddialog.H>
#include <type2sign.H>


Norm* norm = new Norm();


Editor :: Editor()
     : GraphicBlock(nil,p = new Picture,PAD_SIZE)
{
  sl = new Symbollist();
  stepnr = 0;
  transitionnr = 0;
  parallelnr = 0;
  selectionnr = 0;
  connectionnr = 0;
  go = false;
  being_edited = "gcdir/nameless";
}



boolean Editor :: isEditmode()
{
  return !go;
}



void Editor::Handle(Event&)
{}
```

```
boolean Editor :: SlideRect(Coord& x, Coord& y,
                            int w, int h, Coord offx, Coord offy,
                            boolean move_command)
{
  SlidingRect r(output, GetCanvas(), 0, 0, w, h, -offx, -offy);

  Listen(allEvents);
  boolean done = false;
  boolean slider_created = false;
  boolean loop = true;
  Event e;
  do {
    Read(e);
    switch (e.eventType) {
    case DownEvent:
      if ( e.target != this ) {
        //e.target = nil;
        loop = false;
        UnRead(e);
      }
      else if ( !slider_created) {
        if ( move_command ) done = true;
        slider_created = true;
      }
      else {
        done = true;
      }
    case MotionEvent:
      if ( slider_created || move_command) {
        x = e.x; y = e.y;
        r.Track(x, y);
      }
      break;
    default:
      if (e.target != this)
        e.target->Handle(e);
      break;
    }
  } while ( loop && !done );

  r.Erase();

  Listen(input);       //Restore input sensor

  x += w / 2 - offx;
  y += h / 2 - offy;
  return done;
}



void Editor :: Disconnect(Symbol* s)

// Disconnects the symbol from it's neighbours
// and delets the connectionlines.

{
  Graphic* g;
  while ( s->NrOfInputs() > 0 ) {
    // The symbols output pointing at this input
    // is set to nil. Erase its graphic.
    s->GetInput()->DisconnectOutput(s);
    g = s->GetInput()->GetGraphic();
```

```
    p->Remove(g);
    delete g;
    g = s->GetInput()->GetAdjusted(999);
    p->Append(g);

    // Take the connection out of the editors picture.
    g = s->GetInputConnection()->GetGraphic();
    p->Remove(g);
    delete g;

    // Get the connection out of the symbollist.
    sl->Out( s->GetInputConnection() );

    s->DisconnectFirstInput();
  };

  while ( s->NrOfOutputs() > 0 ) {
    // The symbols input pointing at this output
    // is set to nil.
    s->GetOutput()->DisconnectInput(s);
    g = s->GetOutput()->GetGraphic();
    p->Remove(g);
    delete g;

    g = s->GetOutput()->GetAdjusted(999);
    p->Append(g);

    // Take the connection out of the editors picture.
    g = s->GetOutputConnection()->GetGraphic();
    p->Remove(g);
    delete g;

    // Get the connection out of the symbollist.
    sl->Out( s->GetOutputConnection() );

    s->DisconnectFirstOutput();
  };
  Update();
}



boolean Editor :: HitSymbol(Symbol *&s)
{
  Listen(allEvents);
  boolean halfway = false;
  boolean done = false;
  boolean loop = true;
  Event e;
  do {
    Read(e);
    if (e.target != this) {
      if (e.eventType==DownEvent) {
        UnRead(e);
        loop = false;
      }
      //e.target->Handle(e);
    }
    else if (e.eventType==DownEvent) {
      s = sl->GetSymbol(e);
      if ( s != nil && s->GetType() != ConnectionType )
        halfway = true;
    }
    else if (e.eventType==UpEvent && halfway ) {
      // It is important to read the upevent, else it can block
      // a following dialogbox.
      done = true;
    }
  } while ( !done && loop );
  Listen(input);
  return done;
}



void Editor :: WriteText()
{
  boolean doit;
  do {
    Symbol *s;
    doit = HitSymbol(s);
    if ( doit ) {
      s->EditText();

      Graphic *g = s->GetGraphic();
      p->Remove(g);
      delete g;

      p->Append(s->CreateGraphic());
      Update();
    }
  } while ( doit );
}



void Editor :: MoveSymbol()
{
  // Moves the symbol to a new position and redraws it.
  // Erase and delete the connectionlines drawn to the symbol.

  // Chose the symbol by hitting it with a down event.
  Symbol *s=nil;
  boolean doit;
  do {
    doit = HitSymbol(s);
    if ( doit ) {
      // Disconnect the symbol from it's neighbours.
      Disconnect(s);

      // Erase the old graphic.
      Graphic *g = s->GetGraphic();
      p->Remove(g);
      Update();
      delete g;

      // Place the new graphic with the mouse and draw it.
      Coord x,y,w,h;
      s->GetBounds(x,y,w,h);
      SlideRect(x,y,w,h,0,0,true);
      s->SetBounds(x,y,w,h);
      p->Append(s->CreateGraphic());
      Update();
    }
  } while (doit);
}



void Editor :: DeleteSymbol()
```

```
{
  Symbol* s;
  HitSymbol(s);
  StdMessage* sm;
  int buttonnr;
  sm = new StdMessage(new VBox(new Message("Do you really want"),
                               new Message("to delete this"),
                               new Message("symbol?")),
                      "Cancel","Yes");
  buttonnr = sm->Display();
  delete sm;
  if (buttonnr == 2) {
    Disconnect(s);
    Graphic* g = s->GetGraphic();
    p->Remove(g);
    delete g;
    Update();
    sl->Out(s);
  }
}


void Editor :: MakeStep()
{
  boolean doit;
  do {
    // Place the new graphic with the mouse and draw it.
    Coord x,y;
    doit = SlideRect(x,y,30,20,0,0);
    if ( doit ) {
      Step* s = new Step(stepnr);
      stepnr++;
      sl->Into(s);
      s->SetBounds(x,y,0,0);
      p->Append(s->CreateGraphic());
      Update();
    }
  } while (doit);
}


void Editor :: MakeInitStep()
{
  Coord x,y;
  boolean doit = SlideRect(x,y,30,20,0,0);
  if(doit) {
    InitStep* is = new InitStep(stepnr);
    stepnr++;
    sl->Into(is);
    is->SetBounds(x,y,0,0);
    p->Append(is->CreateGraphic());
    Update();
  }
}


void Editor :: MakeTransition()
{
  boolean doit;
  do {
    // Place the new graphic with the mouse and draw it.
    Coord x,y;
    doit = SlideRect(x,y,10,0,0,0);
    if ( doit ) {
      Transition* t = new Transition(transitionnr);
      transitionnr++;
      sl->Into(t);
      t->SetBounds(x,y,0,0);
      p->Append(t->CreateGraphic());
      Update();
    }
  } while (doit);
}


void Editor :: MakeParallel()
{
  boolean doit;
  do {
    // Place the new graphic with the mouse and draw it.
    Coord x,y;
    doit = SlideRect(x,y,30,2,0,0);
    if ( doit ) {
      Parallel* par = new Parallel(parallelnr);
      parallelnr++;
      sl->Into(par);
      par->SetBounds(x,y,30,2);
      p->Append(par->CreateGraphic());
      Update();
    }
  } while (doit);
}


void Editor :: MakeSelection()
{
  boolean doit;
  do {
    // Place the new graphic with the mouse and draw it.
    Coord x,y;
    doit = SlideRect(x,y,30,0,0,0);
    if ( doit ) {
      Selection* sel = new Selection(selectionnr);
      selectionnr++;
      sl->Into(sel);
      sel->SetBounds(x,y,30,0);
      p->Append(sel->CreateGraphic());
      Update();
    }
  } while (doit);
}


void Editor :: MakeConnection()
{
  Symbol* from;
  Symbol* to;
  Coord* x;
  Coord* y;
  int n;
  boolean doit;
  do {
    doit = GrowConnection(from, to, x, y, n);
    // Check if the graphcet norm is followed before
```

```
    // creating the connection.
    if ( doit && norm->Norm_Followed(from,to) ) {
      Connection* c = new Connection(connectionnr);
      connectionnr++;

      // Let the symbols know about the connection.
      from->ConnectOutput(to,c);
      to->ConnectInput(from,c);

      AdjustConnection(from, to, x, y, n);
      c->SetVertices(x, y, n);

      p->Append(c->CreateGraphic());
      sl->Into(c);
      Update();

    }
  } while (doit);
}



boolean Editor :: GrowConnection(Symbol *&from, Symbol *&to,
                                 Coord *&x, Coord *&y, int& n)
{
  GrowingConnection gc(output, canvas);
  Listen(allEvents);
  boolean conn_start = false;
  boolean loop = true;
  Symbol* s;
  Event e;
  do {
    Read(e);
    s = sl->GetSymbol(e);
    switch (e.eventType) {
    case DownEvent:
      if ( e.target != this ) {
        loop = false;
        UnRead(e);
        }
      else if ( s != nil && s->GetType() != ConnectionType ) {
        gc.Attach(s, e.x, e.y);
        conn_start = true;
      }
      else if ( s == nil && conn_start ) {
        gc.Append(e.x, e.y);
      }
      break;
    case MotionEvent:
      gc.Track(e.x, e.y);
      break;
    default:
      if (e.target != this)
        e.target->Handle(e);
    }
  } while (loop && !gc.Valid());
  Listen(input);

  gc.GetCurrent(from, to, x, y, n);
  // Erase gc and return the coordinates for the
  // multiline.
  return gc.Valid();
}
```

```
boolean Editor :: Overlap(Symbol* from, Symbol* to)
{
  boolean ol = false;
  if (to->GetLeft() <= from->GetRight() &&
      to->GetLeft() >= from->GetLeft() ||
      to->GetRight() <= from->GetRight() &&
      to->GetRight() >= from->GetLeft())
    ol = true;
  return ol;
}



void Editor :: AdjustConnection(Symbol *from, Symbol *to,
                                Coord *&x, Coord *&y, int n)
// Adjusts the end vertices on the connectionline.
{
  boolean simplestart =  from->GetType() == StepType ||
                         from->GetType() == InitStepType ||
                         from->GetType() == TransitionType;

  boolean simpleend =  to->GetType() == StepType ||
                       to->GetType() == InitStepType ||
                       to->GetType() == TransitionType;
  // Adjust all vertices.
  for ( int i = 1; i < n-1; ++i ) {
    x[i] = x[i] - PAD_SIZE;
    y[i] = y[i] - PAD_SIZE;
  }
  y[0] = from->GetBottom();
  y[n-1] = to->GetTop();
  if (!simplestart && !simpleend) {
    // Implementation can be extended to manipulate
    // the width on both these wide symbols.
    if (Overlap(from, to))
      if (from->GetRight() == to->GetLeft())
        x[0] = x[1] = from->GetRight();
      else
        if (from->GetLeft() == to->GetRight())
          x[0] = x[1] = from->GetLeft();
  }
  else {
    // Check how the connection starts.
    if ( simplestart )
      x[0] = x[1] = from->GetX();

    // Check how the connection ends.
    if ( simpleend )
      x[n-1] = x[n-2] = to->GetX();
    else {
      Graphic* g = to->GetGraphic();
      p->Remove(g);
      delete g;
      g = to->GetAdjusted(x[0]);
      p->Append(g);
    }
    // If the connection started with a parallell or select,
    // adjust that symbols width.
    if (!simplestart) {
      Graphic* g = from->GetGraphic();
      p->Remove(g);
      delete g;
      g = from->GetAdjusted(x[n-1]);
```

```
      p->Append(g);
    }
  }
}



void Editor :: New()
{
  if ( sl->First() ) {
    sl->ClearList(p);
  }
  Update();
  being_edited = "gcdir/nameless";
  stepnr = 0;
  transitionnr = 0;
  parallelnr = 0;
  selectionnr = 0;
  connectionnr = 0;
  go = false;
  compiled = false;
}



boolean Editor :: ChooseFile(char* header,
                             char* button)
{
  FileChooser* fc = new FileChooser(header,
                                    "or press cancel.",
                                    "gcdir",10,25,button);
  Frame* frame = new ShadowFrame(fc);
  InterViews_world->InsertPopup(frame);

  boolean status = fc->Accept();
  if ( status ) {
    being_edited = cpy(fc->Choice());
    delete frame;
  }
  else {
    delete frame;
  }
  return status;
}



void Editor :: SaveGraphAs()
{
  boolean save = ChooseFile("Save current graph","Save");
  if ( save )
    SaveGraph();
}



void Editor :: SaveGraph()
{
  // Create a file to save the Grafcet in.
  ofstream curFile(being_edited,ios::out);
  curFile.clear();
  Symbol* tmp = sl->First();
  while ( tmp != nil ) {
    tmp->WriteOnFile(curFile);
    tmp = tmp->GetNext();
  };
  curFile.close();
}



void Editor :: ReadGraph()
{
  boolean read_it = ChooseFile("Select a file to read","Load");
  if ( read_it )
    ReadIt();
}



void Editor :: ReadIt()
{
  // Read the graphcetfile twice. Create the symbols in the first loop
  // and connect them in the second.
  int texttest; // 1 if text 0 else.
  int id,iid,icid,oid,ocid;
  char sign,isign,osign;
  int n;
  int nr;
  Coord x,y,left,right;
  Coord vx[10],vy[10];
  stepnr = 0;
  transitionnr = 0;
  parallelnr = 0;
  selectionnr = 0;
  connectionnr = 0;

  ifstream ifstrl(being_edited,ios::in);
  if ( sl->First() ) {
    // If the symbollist isn't empty.
    sl->ClearList(p);
  }
  Transition* t;
  Step* s;
  InitStep* is;
  Parallel* par;
  Selection* sel;
  Connection* conn;
  int i;
  while ( ifstrl >> sign ) {
    switch ( sign ) {
    case 'A':
      transitionnr++;
      t = new Transition(ifstrl);
      p->Append(t->CreateGraphic());
      sl->Into(t);
      break;
    case 'B':
      stepnr++;
      s = new Step(ifstrl);
      p->Append(s->CreateGraphic());
      sl->Into(s);
      break;
    case 'C':
      stepnr++;
      is = new InitStep(ifstrl);
      p->Append(is->CreateGraphic());
```

```
      sl->Into(is);
      break;
    case 'D':
      parallelnr++;
      par = new Parallel(ifstr1);
      p->Append(par->CreateGraphic());
      sl->Into(par);
      break;
    case 'E':
      selectionnr++;
      sel = new Selection(ifstr1);
      p->Append(sel->CreateGraphic());
      sl->Into(sel);
      break;
    case 'F':
      connectionnr++;
      conn = new Connection(ifstr1);
      p->Append(conn->CreateGraphic());
      sl->Into(conn);
    };
  }
  ifstr1.close();
  ifstream ifstr2(being_edited,ios::in);

  Symbol* thissymbol;
  Symbol* from;
  Symbol* to;
  Symbol* fromconn;
  Symbol* toconn;

  SymbolType inType,outType;
  while ( ifstr2 >> sign ) {
    switch ( sign ) {
    case 'A': // Transition
      ifstr2 >> id >> x >> y;
      ifstr2 >> isign >> iid >> icid;
      ifstr2 >> osign >> oid >> ocid;
      for (i=0;i<2;++i) {
        ifstr2 >> texttest;
        if ( texttest > 0 ) {
          ifstr2.get();         // skip space after length
          char* text = new char[texttest+1];
          ifstr2.get(text, texttest+1, '\0');
          delete text;
        }
      }
      thissymbol = sl->GetSymbol(Signalement2Type(sign),id);
      inType = Signalement2Type(isign);
      if ( inType != NoType ) {
        from = sl->GetSymbol(inType,iid);
        fromconn = sl->GetSymbol(ConnectionType, icid);
        thissymbol->ConnectInput(from,fromconn);
      }
      outType = Signalement2Type(osign);
      if ( outType != NoType ) {
        to = sl->GetSymbol(outType,oid);
        toconn = sl->GetSymbol(ConnectionType, ocid);
        thissymbol->ConnectOutput(to,toconn);
      }
      break;



    case 'B':  // Step
      ifstr2 >> id >> x >> y;
      ifstr2 >> isign >> iid >> icid;
      ifstr2 >> osign >> oid >> ocid;
      for (i=0;i<5;++i) {
        ifstr2 >> texttest;
        if ( texttest > 0 ) {
          ifstr2.get();         // skip space after length
          char* text = new char[texttest+1];
          ifstr2.get(text, texttest+1, '\0');
          delete text;
        }
      }
      thissymbol = sl->GetSymbol(Signalement2Type(sign),id);
      inType = Signalement2Type(isign);
      if ( inType != NoType ) {
        from = sl->GetSymbol(inType,iid);
        fromconn = sl->GetSymbol(ConnectionType, icid);
        thissymbol->ConnectInput(from,fromconn);
      }
      outType = Signalement2Type(osign);
      if ( outType != NoType ) {
        to = sl->GetSymbol(outType,oid);
        toconn = sl->GetSymbol(ConnectionType, ocid);
        thissymbol->ConnectOutput(to,toconn);
      }
      break;



    case 'C':   // Initstep
      ifstr2 >> id >> x >> y;
      ifstr2 >> isign >> iid >> icid;
      ifstr2 >> osign >> oid >> ocid;
      for (i=0;i<2;++i) {
        ifstr2 >> texttest;
        if ( texttest > 0 ) {
          ifstr2.get();         // skip space after length
          char* text = new char[texttest+1];
          ifstr2.get(text, texttest+1, '\0');
          delete text;
        }
      }
      thissymbol = sl->GetSymbol(Signalement2Type(sign),id);
      inType = Signalement2Type(isign);
      if ( inType != NoType ) {
        from = sl->GetSymbol(inType,iid);
        fromconn = sl->GetSymbol(ConnectionType, icid);
        thissymbol->ConnectInput(from,fromconn);
      }
      outType = Signalement2Type(osign);
      if ( outType != NoType ) {
        to = sl->GetSymbol(outType,oid);
        toconn = sl->GetSymbol(ConnectionType, ocid);
        thissymbol->ConnectOutput(to,toconn);
      }
      break;



    case 'D':    // Parallel
      ifstr2 >> id >> x >> y >> left >> right;
      thissymbol = sl->GetSymbol(Signalement2Type(sign),id);
      ifstr2 >> nr;
      for (i=0; i<nr; ++i) {
```

```
      ifstr2 >> isign >> iid >> icid;
      from = sl->GetSymbol(Signalement2Type(isign),iid);
      fromconn = sl->GetSymbol(ConnectionType, icid);
      thissymbol->ConnectInput(from,fromconn);
    }
    ifstr2 >> nr;
    for (i=0; i<nr; ++i) {
      ifstr2 >> osign >> oid >> ocid;
      to = sl->GetSymbol(Signalement2Type(osign),oid);
      toconn = sl->GetSymbol(ConnectionType, ocid);
      thissymbol->ConnectOutput(to,toconn);
    }
    break;



  case 'E':   // Selection
    ifstr2 >> id >> x >> y >> left >> right;
    thissymbol = sl->GetSymbol(Signalement2Type(sign),id);
    ifstr2 >> nr;
    for (i=0; i<nr; ++i) {
      ifstr2 >> isign >> iid >> icid;
      from = sl->GetSymbol(Signalement2Type(isign),iid);
      fromconn = sl->GetSymbol(ConnectionType, icid);
      thissymbol->ConnectInput(from,fromconn);
    }
    ifstr2 >> nr;
    for (i=0; i<nr; ++i) {
      ifstr2 >> osign >> oid >> ocid;
      to = sl->GetSymbol(Signalement2Type(osign),oid);
      toconn = sl->GetSymbol(ConnectionType, ocid);
      thissymbol->ConnectOutput(to,toconn);
    }
    break;



  case 'F':  // Connection
    ifstr2 >> id;
    ifstr2 >> n; // Number of vertices.
    for ( i=0; i < n; ++i)
      ifstr2 >> vx[i] >> vy[i];
    ;
  };
}
ifstr2.close();
Update();
}
```

```
#include <grafcetcontrol.H>
#include <symbol.H>
#include <stream.h>
#include <math.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
#include <InterViews/box.h>
#include <InterViews/event.h>
#include <InterViews/message.h>
#include <InterViews/sensor.h>
#include <stddialog.H>



GrafcetControl :: GrafcetControl()
 : Editor(),
   time_table(new TimeTable()),
   var_table(new VarTable())
{
  input = new Sensor(allEvents);
  go = false;
  compiled = false;
  transition_vector = nil;
}



void GrafcetControl :: Go()
{
  if ( compiled && !go ) {
    go = true;
    input->CatchChannel(0);
    Coord l,b,r,t;
    for ( int j = 0; j<nr_of_steps; ++j) {
      step_changed[j] = false;
    }
    for ( int i=0; i<nr_of_initsteps; ++i) {
      step[initkey[i]] = 1;
      token[initkey[i]]->SetBrush(psingle);
      p->Append(token[initkey[i]]);
      token[initkey[i]]->GetBox(l,b,r,t);
      Redraw(l,b,r,t);
      var_table->Update(step_name[initkey[i]],1);
    }
    Calc();
    if ( !time_table->Empty() ) {
      cur_time_logic = time_table->GetTime_logic_to_update();
      double dsec = time_table->GetNextDelay();
      int isec = int(floor(dsec));
      int iusec = nint( ( dsec - double(isec) ) * 1e6 );
      input->CatchTimer(isec,0);
    }
  }
}



void GrafcetControl :: Stop()
{
  if ( go ) {
    input->Ignore(TimerEvent);
    input->IgnoreChannel(0);
    go = false;
    Coord l,b,r,t;
    // Erase all tokens before quit running.
    for (int i = 0; i < nr_of_steps; ++i)
      if ( step[i] == 1 ) {
        token[i]->GetBox(l,b,r,t);
        p->Remove(token[i]);
        Redraw(l,b,r,t);
        step[i] = 0;
      }
  }
}



void GrafcetControl :: Handle(Event& e)
{
  if ( go ) {
    if ( e.eventType == TimerEvent ) {
      cur_time_logic->Time_update();
      time_table->FirstOut();
      input->Ignore(TimerEvent);
      Calc();
    } else if ( e.eventType == ChannelEvent ) {
      // ReadChannel and update the inputvector.
      char str[10];
      int state;
      cin >> str;
      cin >> state;
      var_table->Update(str,state);
      Calc();
    }
    if ( !time_table->Empty() ) {
      cur_time_logic = time_table->GetTime_logic_to_update();
      double dsec = time_table->GetNextDelay();
      int isec = int(floor(dsec));
      int iusec = nint( ( dsec - double(isec) ) * 1e6 );
      input->CatchTimer(isec,iusec);
    }
  }
}



void GrafcetControl :: Calc()
{
  Coord l,b,t,r;
  boolean one_more = true;
  int loopcontrol = 0;
  while ( one_more && loopcontrol < nr_of_steps ) {
    loopcontrol++;
    if ( loopcontrol == nr_of_steps ) cerr << "loooop\n";
    one_more = false;

    // check all transitions
    for (int nr=0; nr < nr_of_transitions; ++nr ) {
      if ( isEnable(nr) && transition_vector->Is(nr) ) {
        fire[nr]=true;
      } else {
        fire[nr]=false;
      }
    }

    // let all flanklogics that wasn't enable pass the flank
    var_table->PassFlank();
```

```
    // fire all transitions that is true and enable and
    // calculate the new active set of steps and update
    // the tokens.
    // S(x+1)=S(x)+A*T(x)
    for (nr=0; nr < nr_of_transitions; ++nr ) {
      if ( fire[nr] ) {
        for (int j=0; j < nr_of_steps; j++) {
          if ( A(j,nr) == 1 ) {
            step[j] = step[j] + A(j,nr);
            step_changed[j] = true;

            // Update the graphics.
            token[j]->SetBrush(psingle);
            p->Append(token[j]);
            token[j]->GetBox(l,b,r,t);
            Redraw(l,b,r,t);

            // Update the variable table.
            var_table->Update(step_name[j],step[j]);
            one_more = true;
          }
          if ( A(j,nr) == -1 ) {
            step[j] = step[j] + A(j,nr);
            step_changed[j] = true;

            // Update the graphics.
            token[j]->GetBox(l,b,r,t);
            p->Remove(token[j]);
            Redraw(l,b,r,t);

            // Update the variable table.
            var_table->Update(step_name[j],step[j]);
            one_more = true;
          }
        }
      }
    }
  }

  // update all actions that changed in this round.
  for (int j=0; j<nr_of_steps; ++j) {
    if ( step_changed[j] ) {
      action_table->Update(step_action[j],step[j]);
      step_changed[j] = false;
    }
  }

  // send all actions that has changed on cerr.
  action_table->SendOutput();
}



boolean GrafcetControl :: isEnable(int transition_nr)
{
  boolean enable = true;
  int j = 0;
  while ( what[transition_nr][j] != 999 && enable ) {
    if ( step[what[transition_nr][j]] == 0 )
      enable = false;
    j++;
  }
  return enable;
}
```

```
int GrafcetControl :: A(int i, int j)
{
 return a[i][j];
}




//  * COUNT STEPS AND TRANSITIONS
//  * NOTE TOKEN POSITIONS
//  * CHECK THAT ALL SYMBOLS ARE CONNECTED


boolean GrafcetControl :: CheckSyntax()
{
  compiled = true;
  StdAlert* sa;
  nr_of_initsteps = 0;
  for (int i=0;i<10;++i)
    initkey[i]=999;
  nr_of_steps = 0;
  nr_of_transitions = 0;
  Symbol* s = sl->First();
  while ( s != nil ) {
    if (s->GetType() == StepType) {
      s->SetId(nr_of_steps);
      step[nr_of_steps] = 0;
      step_name[nr_of_steps] = s->asStep()->step_text[0];
      step_action[nr_of_steps] = s->asStep()->step_text[1];
      token[nr_of_steps] = new FillCircle( s->GetX()-11, s->GetY()-6, 2);
      nr_of_steps++;
      if ( s->NrOfInputs() == 0 || s->NrOfOutputs() == 0 ) {
        sa = new StdAlert(new VBox(new Message("Unconnected step.")));
        sa->Display();
        delete sa;
        compiled = false;
      }
    }
    if (s->GetType() == InitStepType) {
      s->SetId(nr_of_steps);
      step[nr_of_steps] = 1;
      step_name[nr_of_steps] = s->asInitStep()->initstep_text[0];
      step_action[nr_of_steps] = "\0";
      token[nr_of_steps] = new FillCircle( s->GetX()-11, s->GetY()-6, 2);
      initkey[nr_of_initsteps] = nr_of_steps;
      nr_of_initsteps++;
      nr_of_steps++;
      if ( s->NrOfInputs() == 0 || s->NrOfOutputs() == 0 ) {
        compiled = false;
        sa = new StdAlert(new VBox(new Message("The initstep is not"),
                                   new Message("connected.")));
        sa->Display();
        delete sa;
      }
    }
    if (s->GetType() == TransitionType) {
      s->SetId(nr_of_transitions);
      nr_of_transitions++;
      if ( s->NrOfInputs() == 0 || s->NrOfOutputs() == 0 ) {
        compiled = false;
        sa = new StdAlert(new VBox(new Message("Unconnected transition.")));
        sa->Display();
```

```
        delete sa;
      }
    }
    if (s->GetType() == ParallelType || s->GetType() == SelectionType )
      if ( s->NrOfInputs() == 0 && s->NrOfOutputs() == 0 ) {
        sa = new StdAlert(new VBox(new Message("Unconnected parallels or"),
                                   new Message("selections.")));
        sa->Display();
        delete sa;
        compiled = false;
      }
    s = s->GetNext();
  };

  if ( compiled ) {
    time_table->Reset();
    var_table->Reset();

    if ( action_table )
      delete action_table;
    action_table = new ActionTable(nr_of_steps, step_action);

    if ( transition_vector )
      delete transition_vector;
    transition_vector = new Transition_vector(nr_of_transitions,
                                              sl,var_table,time_table);
    return compiled;
  }
}



//
//   MAKE RELATION
//
//   The incidence matrix and the what matrix is defined here.
//
//   The incidence matrix tells how tokens are moved when
//   transition is fired and the what matrix tells if a transition
//   is enable.


void GrafcetControl :: MakeRelation()
{
  for (int i=0; i < nr_of_transitions; ++i )
    for (int j=0; j<10; ++j )
      what[i][j] = 999;

  for (i=0; i < nr_of_steps; i++) {
    for (int j=0; j < nr_of_transitions; j++)
      a[i][j] = 0;
  }

  // Make the incidence matrix and the what matrix complete.
  Symbol* s = sl->First();
  while ( s != nil ) {
    if (s->GetType() == TransitionType &&
        s->GetInput() != nil &&
        s->GetOutput() != nil) {
      switch ( s->GetInput()->GetType() ) {
      case InitStepType:
      case StepType:
        a[s->GetInput()->GetId()][s->GetId()] = -1;
        what[s->GetId()][0] = s->GetInput()->GetId();
        break;
      case ParallelType:
        for (i = 0; i < s->GetInput()->NrOfInputs(); ++i ) {
          a[s->GetInput()->GetInput(i)->GetId()][s->GetId()] = -1;
          what[s->GetId()][i] = s->GetInput()->GetInput(i)->GetId();
        }
        break;
      case SelectionType:
        switch ( s->GetInput()->GetInput()->GetType() ) {
        case InitStepType:
        case StepType:
          a[s->GetInput()->GetInput()->GetId()][s->GetId()] = -1;
          what[s->GetId()][0] = s->GetInput()->GetInput()->GetId();
          break;
        case ParallelType:
          for (i=0; i < s->GetInput()->NrOfInputs();++i) {
            a[s->GetInput()->GetInput()->GetInput(i)->GetId()][s->GetId()]=-1;
            what[s->GetId()][i]=
              s->GetInput()->GetInput()->GetInput(i)->GetId();
          }
        };
      };
      switch (s->GetOutput()->GetType()) {
      case InitStepType:
      case StepType:
        a[s->GetOutput()->GetId()][s->GetId()] = 1;
        break;
      case ParallelType:
        for (i=0; i < s->GetOutput()->NrOfOutputs(); i++)
          a[s->GetOutput()->GetOutput(i)->GetId()][s->GetId()] = 1;
        break;
      case SelectionType:
        switch (s->GetOutput()->GetOutput()->GetType()) {
        case InitStepType:
        case StepType:
          a[s->GetOutput()->GetOutput()->GetId()][s->GetId()]=1;
          break;
        case ParallelType:
          for (i=0;i<s->GetOutput()->GetOutput()->NrOfOutputs();i++)
            a[s->GetOutput()->GetOutput()->GetOutput(i)->GetId()][s->GetId()]=1;
        };
      };
    }
    s = s->GetNext();
  };
  compiled = true;
}
```

```
#include <iostream.h>
#include <stdlib.h>

#include <InterViews/button.h>
#include <InterViews/box.h>
#include <InterViews/event.h>
#include <InterViews/Graphic/base.h>
#include <InterViews/Graphic/ellipses.h>
#include <InterViews/menu.h>
#include <InterViews/message.h>
#include <InterViews/sensor.h>

#include <string.h>
#include <time.h>
#include <sys/timeb.h>

#include <comp.H>
#include <grafcetcontrol.H>
#include <symbol.H>
#include <mystring.H>
#include <gcdialog.H>
#include <stddialog.H>



//////////////////////////////////////////////////////////////
//
//    CreateFather is a rekursive funktion that builds
//    the logical structure of a transition condition.
//    When the transition vector builds it's condition
//    it calls CreateFather.
//
//    The structure of the logics will be a binary
//    tree.

Logics* CreateFather(char* str,VarTable* vt,TimeTable* tt)
{
  Logics* father;
  int ands = 0;
  int ors  = 0;
  int nots = 0;
  int ups = 0;
  int downs = 0;
  int timers = 0;
  int or[3];  // Remembers the position of a character.
  int and[3];
  int not[4];
  int up[3];
  int down[3];
  int timer[2];
  str = ParentheseGroup(str);
  int index = 0;
  boolean is_substr = false;
  boolean is_timer = false;
  while ( str[index] ) {
    switch ( str[index] ) {
    case '|':
      if ( !is_substr ) { or[ors] = index; ors++; };
      break;
    case '&':
      if ( !is_substr ) { and[ands] = index; ands++; };
      break;
    case '(':
      is_substr = true;
      break;
    case ')':
      is_substr = false;
      break;
    case '!':
      if ( !is_substr ) { not[nots] = index; nots++; };
      break;
    case '^':
      if ( !is_substr ) { up[ups] = index; ups++; };
      break;
    case '_':
      if ( !is_substr ) { down[downs] = index; downs++; };
      break;
    case '/':
      is_timer = true;
      timer[timers] = index;
      timers++;
    };
    index++;
  };
  if ( is_timer ) {
    char* t1 = cpy(str,0,timer[0]-1);
    char* t2;
    char* condition;
    if ( timers == 2 ) {
      t2 = cpy(str,timer[1]+1,strlen(str)-1);
      condition = cpy(str,timer[0]+1,timer[1]);
    }
    else {
      t2 = "0";
      condition = cpy(str,timer[0]+1,strlen(str));
    }
    Time_logic* t = new Time_logic(condition,t1,t2,vt,tt);
    father = t;
  }
  else if ( ors > 0 ) {
    char* lstr = cpy(str,0,or[0]);
    char* rstr = cpy(str,or[0]+1,strlen(str));
    Or* o = new Or(lstr,rstr,vt,tt);
    father = o;
  }
  else if ( ands > 0 ) {
    char* lstr = cpy(str,0,and[0]);
    char* rstr = cpy(str,and[0]+1,strlen(str));
    And* a = new And(lstr,rstr,vt,tt);
    father = a;
  }
  else if ( nots > 0 ) {
    char* tstr = cpy(str,not[0]+1,strlen(str));
    Not* n = new Not(tstr,vt,tt);
    father = n;
  }
  else if ( ups > 0 ) {
    char* tstr = cpy(str,up[0]+1,strlen(str));
    Up* u = new Up(tstr,vt,tt);
    father = u;
  }
  else if ( downs > 0 ) {
    char* tstr = cpy(str,down[0]+1,strlen(str));
    Down* d = new Down(tstr,vt,tt);
    father = d;
  }
  else if ( strlen(str) == 1 && *str == '1' ) {
    One* one = new One();
    father = one;
  }
```

```
  else { // Variable
    Var* exist = vt->Exist(str);
    Var* v;
    if ( exist == nil ) {
      v = new Var(str);
      vt->Insert(v);
      exist = v;
    };
    father = exist;
  };
  return father;
}



////////////////////////////////////////////////////////////////
//
//    Logic classes that is the components of the binary
//    tree expressing the transition condition.
//

Logics :: Logics() {}
boolean Logics :: Is() {return false;}
void Logics :: Reset() {}
void Logics :: Update(int) {}
void Logics :: CatchFlank(int) {}
void Logics :: PassFlank() {}
void Logics :: Look_at_flanklogic(Logics*) {}


Time_logic :: Time_logic(char* condition,char* time1,char* time2,
                         VarTable* vt, TimeTable* tt)
{
  time_table = tt;
  t1 = atoi(time1);
  t2 = atoi(time2);
  logic = CreateFather(condition,vt,tt);
  logic->Look_at_flanklogic(this);
  newlogicstate = false;
  oldlogicstate = false;
  mystate = false;
  timer_ticking = false;
}


void Time_logic :: Reset()
{
  mystate = false;
  timer_ticking = false;
  newlogicstate = false;
  oldlogicstate = false;
  logic->Reset();
}


boolean Time_logic :: Is()
{
  return mystate;
}


void Time_logic :: CatchFlank(int state)
{
  newlogicstate = state;
  if ( !newlogicstate ) mystate = false;
  if ( !timer_ticking ) {
    if ( newlogicstate && !oldlogicstate ) {
      timer_ticking = true;
      time_table->InsertTimer(t1,this);
    }
    else if ( !newlogicstate && oldlogicstate && t2>0) {
      timer_ticking = true;
      time_table->InsertTimer(t2,this);
    }
  }
  oldlogicstate = newlogicstate;

}


void Time_logic :: Time_update()
{
  mystate = newlogicstate;
  timer_ticking = false;
}


One :: One()
{}


boolean One :: Is()
{
  return true;
}


void One :: Reset() {}



And :: And(char* lstr, char* rstr,VarTable* vt,TimeTable* tt)
     : Logics()
{
  leftlogic = CreateFather(lstr,vt,tt);
  rightlogic = CreateFather(rstr,vt,tt);
}


boolean And :: Is() {
  return leftlogic->Is() && rightlogic->Is();
}


void And :: Reset()
{
  leftlogic->Reset();
  rightlogic->Reset();
}


Or :: Or(char* lstr, char* rstr,VarTable *vt,TimeTable* tt)
     : Logics() {
  leftlogic = CreateFather(lstr,vt,tt);
  rightlogic = CreateFather(rstr,vt,tt);
}


boolean Or :: Is() {
  return leftlogic->Is() || rightlogic->Is();
```

```
}


void Or :: Reset()
{
  leftlogic->Reset();
  rightlogic->Reset();
}


Not :: Not(char* str,VarTable *vt,TimeTable* tt) : Logics() {
  logic = CreateFather(str,vt,tt);
}
boolean Not :: Is() {
  return !logic->Is();
}


void Not :: Reset()
{
  logic->Reset();
}


Up :: Up(char* str,VarTable *vt,TimeTable* tt) : Logics() {
  logic = CreateFather(str,vt,tt);
  logic->Look_at_flanklogic(this);
  oldlogic = false;
  newlogic = false;
}


boolean Up :: Is()
{
  boolean condition = newlogic && !oldlogic;
  return condition;
}


void Up :: CatchFlank(int state)
{
  oldlogic = newlogic;
  newlogic = state;
}


void Up :: PassFlank()
{
  oldlogic = newlogic;
}

void Up :: Reset()
{
  logic->Reset();
  oldlogic = false;
  newlogic = false;
}


Down :: Down(char* str,VarTable *vt,TimeTable* tt) : Logics() {
  logic = CreateFather(str,vt,tt);
  logic->Look_at_flanklogic(this);
  oldlogic = false;
  newlogic = false;
}
```

```
boolean Down :: Is()
{
  boolean condition = oldlogic && !newlogic;
  return condition;
}


void Down :: CatchFlank(int state)
{
  oldlogic = newlogic;
  newlogic = state;
}

void Down :: PassFlank() {
  oldlogic = newlogic;
}

void Down :: Reset()
{
  oldlogic = false;
  newlogic = false;
  logic->Reset();
}


Var :: Var(char* str)
     : Logics()
{
  is = false;
  name = str;
  next = nil;
  nr_of_flanklogics = 0;
}

boolean Var :: Is()
{
  return is;
}

void Var :: Look_at_flanklogic(Logics* l)
{
  flanklogics[nr_of_flanklogics] = l;
  nr_of_flanklogics++;
}

void Var :: Update(int state)
{
  is = state;
  for ( int i=0; i<nr_of_flanklogics; ++i) {
    flanklogics[i]->CatchFlank(state);
  }
}

void Var :: PassFlank() {
  for ( int i=0; i<nr_of_flanklogics; ++i) {
    flanklogics[i]->PassFlank();
  }
}

char* Var :: GetName() { return name; }
void Var :: SetNext(Var* v) { next = v; }
Var* Var :: GetNext() { return next; }
void Var :: Reset() { is = false; }
```

```
////////////////////////////////////////////////////////
//
//    The TimeTable consist a list of timers.
//    Each timer knows what time_logic to update
//    and when to do it.
//
//    A The first timer in the list tells the time
//    at the next timerevent. When the timerevent
//    comes the first timer is taken out of the list
//    and the timer_logic is updated.

TimeTable :: TimeTable()
{
  first = nil;
}


void TimeTable :: Reset()
{
  while ( first ) {
    Timer* tmp = first;
    first = first->GetNext();
    delete tmp;
  }
}


TimeTable :: ~TimeTable()
{
  Timer* tmp;
  while ( first && first->GetNext() ) {
    tmp = first;
    first = first->GetNext();
    delete tmp;
  }
}


double TimeTable :: GetTime()
{
  timeb tp;
  ftime(&tp);
  return double(tp.time) + double(tp.millitm) / 1000.0;
}


void TimeTable :: InsertTimer(int t,Time_logic* tl)
{
  double tt = double(t)+GetTime();
  Timer* timer = new Timer(tt,tl);
  Timer* tmp1 = first;
  Timer* tmp2;
  if ( first == nil ) {
    first = timer;
  }
  else if ( tt < first->GetTime() ) {
    first = timer;
    first->SetNext(tmp1);
    tmp1->SetPrev(first);
  }
  else {
    boolean done = false;
    while ( !done ) {
      tmp2 = tmp1;
      tmp1 = tmp1->GetNext();
      if ( tmp1 ) {
        if ( tt < tmp1->GetTime() ) {
          timer->SetNext(tmp1);
          timer->SetPrev(tmp2);
          tmp2->SetNext(timer);
          tmp1->SetPrev(timer);
          done = true;
        }
      }
      else {
        tmp2->SetNext(timer);
        timer->SetPrev(tmp2);
        done = true;
      }
    }
  }
}


boolean TimeTable :: Empty()
{
  boolean empty = true;
  if ( first ) empty = false;
  return empty;
}


Time_logic* TimeTable :: GetTime_logic_to_update()
{
  Time_logic* tl = nil;
  if ( first )
    tl = first->GetTime_logic();
  return tl;
}


void TimeTable :: FirstOut()
{
  Timer* tmp = first;
  if ( first ) first = first->GetNext();
  if ( first ) first->SetPrev(nil);
  delete tmp;
}


double TimeTable :: GetNextDelay()
{
  double t = first->GetTime();
  double t_global = GetTime();
  if ( t < t_global ) {
    t = t_global;
  }
  return t - t_global;
}


Timer :: Timer(double t,Time_logic* time_logic)
{
  pTimer = nil;
  nTimer = nil;
  tl = time_logic;
  time = t;
```

```
}


Timer* Timer :: GetNext()
{
  return nTimer;
}


Timer* Timer :: GetPrev()
{
  return pTimer;
}


void Timer :: SetNext(Timer* next)
{
  nTimer = next;
}


void Timer :: SetPrev(Timer* prev)
{
  pTimer = prev;
}


Timer :: ~Timer() {}

double Timer :: GetTime()
{
  return time;
}


Time_logic* Timer :: GetTime_logic()
{
  return tl;
}



//////////////////////////////////////////////////////////////////////
//
//   The transition_vector knows the logical
//   relation between variables and transitions

Transition_vector :: Transition_vector(int transitions,
                                       Symbollist* sl,
                                       VarTable* var_table,
                                       TimeTable* time_table)
{
  nr_of_transitions = transitions;
  for (int i=0; i < nr_of_transitions; ++i) {
    Symbol* t = sl->GetSymbol(TransitionType,i);
    //char* str = cpy();
    transition_vector[i]
      = CreateFather(t->asTransition()->transitioncondition[0],
                     var_table,time_table);
  }
  for (i=nr_of_transitions; i < 100; ++i)
    transition_vector[i] = nil;
}
```

```
Transition_vector :: ~Transition_vector()
{
  for (int i=0;i<nr_of_transitions;++i)
    delete transition_vector[i];
}


boolean Transition_vector :: Is(int i)
{
  return transition_vector[i]->Is();
}


//////////////////////////////////////////////////////////////////////
//
//   The var table is a list that include all variables
//   and there states. Steps that is used as transition-
//   conditions is also included.
//
VarTable :: VarTable()
{
  first_var = nil;
}


Var* VarTable :: Exist(char* var_name)
{
  boolean exist = false;
  Var* vp = first_var;
  while ( vp && !exist ) {
    if ( StrEqual(vp->GetName(),var_name) ) {
      exist = true;
    }
    else
      vp = vp->GetNext();
  }
  return vp;
}


void VarTable :: Insert(Var* v)
{
  if ( first_var ) {
    Var* tmp = first_var;
    first_var = v;
    v->SetNext(tmp);
  }
  else
    first_var = v;
}


void VarTable :: Update(char* var_name,int state)
{
  boolean found = false;
  Var* vp = first_var;
  while ( vp && !found ) {
    if ( StrEqual(vp->GetName(),var_name) ) {
      found = true;
      vp->Update(state);
    }
    else
      vp = vp->GetNext();
  }
}
```

```
void VarTable :: PassFlank() {
  Var* vp = first_var;
  while ( vp ) {
    vp->PassFlank();
    vp = vp->GetNext();
  }
}


void VarTable :: Reset()
{
  first_var = nil;
}

////////////////////////////////////////////////////////////////////////


Action :: Action(char* str)
{
  is = false;
  old_is = false;
  name = str;
  next = nil;
}


void Action :: Update(int state)
{
  is = state;
}

void Action :: SendOutput() {
  if ( is != old_is && strlen(name) > 0 )
    cerr << name << " " << int(is) << endl;
  old_is = is;
}


char* Action :: GetName() { return name; }
void Action :: SetNext(Action* a) { next = a; }
Action* Action :: GetNext() { return next; }
void Action :: Reset() { is = false; }

////////////////////////////////////////////////////////////////////////
//
//   The action table is a list that include all step actions
//   and there states.

ActionTable :: ActionTable(int steps, char* step_action[100])
{
  first_action = nil;
  for (int i=0; i<steps; ++i) {
    if ( !Exist(step_action[i]) ) {
      Insert(new Action(step_action[i]));
    }
  }
}


Action* ActionTable :: Exist(char* action_str)
{
  boolean exist = false;
  Action* ap = first_action;
  while ( ap && !exist ) {
    if ( StrEqual(ap->GetName(),action_str) ) {
      exist = true;
    }
    else
      ap = ap->GetNext();
  }
  return ap;
}


void ActionTable :: Insert(Action* a)
{
  if ( first_action ) {
    Action* tmp = first_action;
    first_action = a;
    a->SetNext(tmp);
  }
  else
    first_action = a;
}


void ActionTable :: Update(char* action_str,int state)
{
  boolean found = false;
  Action* ap = first_action;
  while ( ap && !found ) {
    if ( StrEqual(ap->GetName(),action_str) ) {
      found = true;
      ap->Update(state);
    }
    else
      ap = ap->GetNext();
  }
}


void ActionTable :: Reset()
{
  Action* tmp = first_action;
  while ( tmp ) {
    tmp->Reset();
    tmp = tmp->GetNext();
  }
}


void ActionTable :: SendOutput() {
  Action* tmp = first_action;
  while ( tmp ) {
    tmp->SendOutput();
    tmp = tmp->GetNext();
  }
}
```

```
#include <InterViews/box.h>
#include <InterViews/message.h>
#include <grafcetnorm.H>
#include <symbol.H>
#include <stddialog.H>

Norm :: Norm() {}

boolean Norm :: Norm_Followed(Symbol* from, Symbol* to)
{
  StdAlert* sa;
  boolean ok = true;
  switch ( from->GetType() ) {
  case TransitionType:
    switch ( to->GetType() ) {
    case TransitionType:
      ok = false;
      sa = new StdAlert(new VBox(new Message("Connection ignored!"),
                                 new Message("Connecting two transitions is"),
                                 new Message("not by the norm, SS IEC 848.")));
      sa->Display();
      delete sa;
      break;
    case ParallelType:
      if (to->NrOfInputs() > 0 ) {
        ok = false;
        sa = new StdAlert(new VBox(new Message("Connection ignored!"),
                                   new Message("It's not by the norm,"),
                                   new Message("SS IEC 848, to connect a"),
                                   new Message("transition to a parallel"),
                                   new Message("that have another input.")));
        sa->Display();
        delete sa;
      }
      if (to->NrOfOutputs() > 0 &&
          to->GetOutput()->GetType() == TransitionType ) {
            // The transition can't be connected as input
            // to the parallel if the parallel has a transition
            // as output.
            ok = false;
            sa = new StdAlert(new VBox(new Message("Connectoin ignored!"),
                                       new Message("The transition can't be"),
                                       new Message("input to a parallel with"),
                                       new Message("a transition on output")));
            sa->Display();
            delete sa;
          }
      if (to->NrOfOutputs() > 0 &&
          to->GetOutput()->GetType() == SelectionType ) {
            // The transition can't be connected as input
            // to the parallel if the parallel has a selection
            // as output.
            ok = false;
            sa = new StdAlert(new VBox(new Message("Connectoin ignored!"),
                                       new Message("The transition can't be"),
                                       new Message("input to a parallel with"),
                                       new Message("a selection on output")));
            sa->Display();
            delete sa;
          }
      break;

    case SelectionType:
      if (to->NrOfInputs() > 0 &&
          to->GetInput()->GetType() != TransitionType ) {
            // The transition can't be connected as input
            // to the select, if another symbol than a transition
            // was connected before this connection.
            ok = false;
            sa = new StdAlert(new VBox(new Message("Connection ignored!"),
                                       new Message("The transition can't be"),
                                       new Message("input to a select with"),
                                       new Message("other symbols than trans"),
                                       new Message("itions on input.")));
            sa->Display();
            delete sa;
          }
      if (to->NrOfOutputs() > 0 &&
          to->GetOutput()->GetType() == TransitionType ) {
            // The transition can't be connected as input
            // to the select, if a transition is connected
            // as output.
            ok = false;
            sa = new StdAlert(new VBox(new Message("Connection ignored!"),
                                       new Message("The transition can't be"),
                                       new Message("connected on the selects"),
                                       new Message("input if there is a tran"),
                                       new Message("sition on the selects"),
                                       new Message("output.")));
            sa->Display();
            delete sa;
          }
      break;
    };
    break;
  case StepType:
    switch ( to->GetType() ) {
    case StepType:
      // A step can't be connected with a step.
      ok = false;
      sa = new StdAlert(new VBox(new Message("Connection ignored!"),
                                 new Message("Two steps can't be connected."),
                                 new Message("It's not by the norm,"),
                                 new Message("SS IEC 848")));
      sa->Display();
      delete sa;
      break;
    case InitStepType:
      // A step can't be connected with a initstep.
      ok = false;
      sa = new StdAlert(new VBox(new Message("Connection ignored!"),
                                 new Message("Two steps can't be connected."),
                                 new Message("It's not by the norm,"),
                                 new Message("SS IEC 848.")));
      sa->Display();
      delete sa;
      break;
    case SelectionType:
      if (to->NrOfInputs() > 0 ) {
        // The step can't be connected as input
        // to the select if a symbol was connected
        // as input before.
        ok = false;
        sa = new StdAlert(new VBox(new Message("Connection ignored!"),
                                   new Message("The step can't be connected"),
                                   new Message("on a select input if there's"),
                                   new Message("one input connected before")));
        sa->Display();
        delete sa;
      }
```

```
    if (to->NrOfOutputs() > 0 &&
        to->GetOutput()->GetType() == StepType ) {
          // The step cant be connected as input
          // to the select if a step is connected
          // as output to the select.
          ok = false;
          sa = new StdAlert(new VBox(new Message("Connection ignored!"),
                                     new Message("A step can't be input to"),
                                     new Message("a select with a step on"),
                                     new Message("the outpot")));
          sa->Display();
          delete sa;
    }
    break;
  case ParallelType:
    if (to->NrOfInputs() > 0 &&
        to->GetInput()->GetType() != StepType ) {
          // The step can't be connected as input
          // to the parallel if there is another input
          // connected that is not a step.
          ok = false;
          sa = new StdAlert(new VBox(new Message("Connection ignored!"),
                                     new Message("Don't mix input types"),
                                     new Message("to a select symbol."),
                                     new Message("It's not by the norm,"),
                                     new Message("SS IEC 848.")));
          sa->Display();
          delete sa;
    }
    if (to->NrOfOutputs() > 0 &&
        to->GetOutput()->GetType() == StepType ) {
          // The step can't be connected as input
          // to the parallel if a step is connected
          // as output on the parallel.
          ok = false;
          sa = new StdAlert(new VBox(new Message("Connection ignored!"),
                                     new Message("Steps can't be both"),
                                     new Message("input and output on"),
                                     new Message("a parallel.")));
          sa->Display();
          delete sa;
    }
    break;
  };
  break;
case InitStepType:
  switch ( to->GetType() ) {
  case StepType:
    ok = false;
    sa = new StdAlert(new VBox(new Message("Connection ignored!"),
                               new Message("Two steps can't be connected."),
                               new Message("It's not by the norm,"),
                               new Message("SS IEC 848.")));
    sa->Display();
    delete sa;
    break;
  case InitStepType:
    ok = false;
    sa = new StdAlert(new VBox(new Message("Connection ignored!"),
                               new Message("Two steps can't be connected."),
                               new Message("It's not by the norm,"),
                               new Message("SS IEC 848.")));
    sa->Display();
    delete sa;
    break;
```

```
  case SelectionType:
    if (to->NrOfInputs() > 0 ) {
      // The step can't be connected as input
      // to the select if a symbol was connected
      // as input before.
      ok = false;
      sa = new StdAlert(new VBox(new Message("Connection ignored!"),
                                 new Message("A step can't be connected"),
                                 new Message("to a select with previous"),
                                 new Message("connections on the input.")));
      sa->Display();
      delete sa;
    }
    if (to->NrOfOutputs() > 0 &&
        to->GetOutput()->GetType() == StepType ) {
          // The step cant be connected as input
          // to the select if a step is connected
          // as output to the select.
          ok = false;
          sa = new StdAlert(new VBox(new Message("Connection ignored!"),
                                     new Message("The select can't have"),
                                     new Message("steps on both input and"),
                                     new Message("output.")));
          sa->Display();
          delete sa;
    }
    break;
  case ParallelType:
    if (to->NrOfInputs() > 0 &&
        to->GetInput()->GetType() != StepType ) {
          // The step can't be connected as input
          // to the parallel if there is another input
          // connected that is not a step.
          ok = false;
          sa = new StdAlert(new VBox(new Message("Connection ignored!"),
                                     new Message("Don't mix input types"),
                                     new Message("to the parallel.")));
          sa->Display();
          delete sa;
    }
    if (to->NrOfOutputs() > 0 &&
        to->GetOutput()->GetType() == StepType ) {
          // The step can't be connected as input
          // to the parallel if a step is connected
          // as output on the parallel.
          ok = false;
          sa = new StdAlert(new VBox(new Message("Connection ignored!"),
                                     new Message("The parallel can't have"),
                                     new Message("steps on both input and"),
                                     new Message("output.")));
          sa->Display();
          delete sa;
    }
    break;
  };
  break;
case ParallelType:
  switch ( to->GetType() ) {
  case ParallelType:
    ok = false;
    sa = new StdAlert(new VBox(new Message("Connection ignored!"),
                               new Message("Why connect two parallels?")));
    sa->Display();
    delete sa;
    break;
```

```
case TransitionType:
  if (from->NrOfOutputs() > 0 ) {
    // A transition can't be connected as output to a
    // parallel if one output already is connencted on
    // the parallel.
    ok = false;
    sa = new StdAlert(new VBox(new Message("Connection ignored!"),
                               new Message("When a parallel and a"),
                               new Message("transition is connected,"),
                               new Message("the parallels output must"),
                               new Message("be unconnected.")));
    sa->Display();
    delete sa;
  }
  if (from->NrOfInputs() > 0 &&
      from->GetInput()->GetType() == TransitionType ) {
        // The transition can't be connected as output
        // to the parallel if a transition is connected
        // as input.
        ok = false;
        sa = new StdAlert(new VBox(new Message("Connection ignored!"),
                                   new Message("A parallel can't have"),
                                   new Message("transitions on both"),
                                   new Message("input and output.")));
        sa->Display();
        delete sa;
  }
  if (from->NrOfInputs() > 0 &&
      from->GetInput()->GetType() == SelectionType ) {
        // The transition can't be connected as output
        // to the parallel if a selection is connected
        // as input.
        ok = false;
        sa = new StdAlert(new VBox(new Message("Connection ignored!"),
                                   new Message("A parallel can't have a"),
                                   new Message("transitions on output"),
                                   new Message("when a selection is"),
                                   new Message("connected on input.")));
        sa->Display();
        delete sa;
  }
  break;
case StepType:
  if (from->NrOfOutputs() > 0 &&
      from->GetOutput()->GetType() != StepType ) {
        // The step can't be connected as output
        // to the parallel if there is another symbol
        // connected as output that is not a step.
        ok = false;
        sa = new StdAlert(new VBox(new Message("Connection ignored!"),
                                   new Message("Don't mix output types"),
                                   new Message("on the parallel")));
        sa->Display();
        delete sa;
  }
  if (from->NrOfInputs() > 0 &&
      from->GetInput()->GetType() == StepType ) {
        // The step can't be connected as output
        // to the parallel if a step is connected as input.
        ok = false;
        sa = new StdAlert(new VBox(new Message("Connection ignored!"),
                                   new Message("The parallel cant have"),
                                   new Message("steps on both input and"),
                                   new Message("output.")));
        sa->Display();
        delete sa;
  }
  if (from->NrOfInputs() > 0 &&
      from->GetInput()->GetType() == InitStepType ) {
        // The step can't be connected as output
        // to the parallel if a initstep is connected as input.
        ok = false;
        sa = new StdAlert(new VBox(new Message("Connection ignored!"),
                                   new Message("The parallel cant have"),
                                   new Message("steps on both input and"),
                                   new Message("output.")));
        sa->Display();
        delete sa;
  }
  break;
case InitStepType:
  // A init step is not ment to one output
  // to a parallel.
  ok = false;
  sa = new StdAlert(new VBox(new Message("Connection ignored!"),
                             new Message("It's not by the norm to"),
                             new Message("connect a parallel to a"),
                             new Message("initstep")));
  sa->Display();
  delete sa;
  break;
case SelectionType:
  if (from->NrOfOutputs() > 0 ) {
    // If the parallel output is connected , a select
    // can't be connected as output on that parallel.
    ok = false;
    sa = new StdAlert(new VBox(new Message("Connection ignored!"),
                               new Message("If a parallel is to be"),
                               new Message("connected to a select,"),
                               new Message("the parallel can't have"),
                               new Message("another output.")));
    sa->Display();
    delete sa;
  }
  if (from->NrOfInputs() > 0 &&
      from->GetInput()->GetType() != StepType ) {
        // If the parallels input is not a step, the
        // connection isn't valid.
        ok = false;
        sa = new StdAlert(new VBox(new Message("Connection ignored!"),
                                   new Message("If the parallels input"),
                                   new Message("is connected it must"),
                                   new Message("be with a step to make"),
                                   new Message("this connection valid.")));
        sa->Display();
        delete sa;
  }
  break;
};
break;
case SelectionType:
  switch (to->GetType()) {
  case InitStepType:
    if (from->NrOfOutputs() > 0 ) {
        // A select can only have transitions as multiple output.
        ok = false;
        sa = new StdAlert(new VBox(new Message("Connection ignored!"),
                                   new Message("Only transitions can be"),
                                   new Message("multiple on the selects"),
                                   new Message("output.")));
```

```
#include <growingconnection.H>


GrowingConnection :: GrowingConnection(Painter* p, Canvas* c)
     : ml(p, c, vx, vy, 0, 0, 0), n_conn(0)
{
  conn[0] = conn[1] = nil;
}


void GrowingConnection :: Track(Coord x, Coord y)
{
  ml.Track(x, y);
}


void GrowingConnection :: Append(Coord x, Coord y)
{
  ml.AppendVertex(x, y);
}


void GrowingConnection :: Attach(Symbol* s, Coord x, Coord y)
{
  if (n_conn < 2 && conn[0] != s) {
    Append(x, y);
    conn[n_conn] = s;
    n_conn++;
  };
}


void GrowingConnection :: GetCurrent(Symbol *&from,Symbol *&to,
                                     Coord *&x, Coord *&y, int& n)
{
  ml.Erase();
  ml.GetCurrent(x, y, n);
  from = conn[0];
  to = conn[1];
}


boolean GrowingConnection :: Valid()
{
  return n_conn == 2;
}
```

```
#include <InterViews/control.h>
#include <InterViews/message.h>
#include <grafcetcontrol.H>
#include <makesymbol.H>


//////////////////////////////////////////////////////////////
Make_Transition :: Make_Transition()
     : Control(new Message("  transition  "))
{}

void Make_Transition :: Do()
{
  if ( gccontrol->isEditmode() ) {
    Highlight(true);
    gccontrol->MakeTransition();
    Highlight(false);
  }
}

//////////////z///////////////////////////////////////////////

Make_Step :: Make_Step()
     : Control(new Message("      step       "))
{}

void Make_Step :: Do() {
  if ( gccontrol->isEditmode() ) {
    Highlight(true);
    gccontrol->MakeStep();
    Highlight(false);
  }
}

//////////////////////////////////////////////////////////////////

Make_InitStep :: Make_InitStep()
     : Control(new Message("    initstep   "))
{}

void Make_InitStep :: Do()
{
  if ( gccontrol->isEditmode() ) {
    Highlight(true);
    gccontrol->MakeInitStep();
    Highlight(false);
  }
}

//////////////////////////////////////////////////////////////

Make_Selection :: Make_Selection()
     : Control(new Message("   selection  "))
{}

void Make_Selection :: Do()
{
  if ( gccontrol->isEditmode() ) {
    Highlight(true);
    gccontrol->MakeSelection();
    Highlight(false);
  }
}

//////////////////////////////////////////////////////////////


Make_Parallel :: Make_Parallel()
     : Control(new Message("    parallel   "))
{}

void Make_Parallel :: Do()
{
  if ( gccontrol->isEditmode() ) {
    Highlight(true);
    gccontrol->MakeParallel();
    Highlight(false);
  }
}
//////////////////////////////////////////////////////////////

Make_Connection :: Make_Connection()
     : Control(new Message("   connect   "))
{}

void Make_Connection :: Do()
{
  if ( gccontrol->isEditmode() ) {
    Highlight(true);
    gccontrol->MakeConnection();
    Highlight(false);
  }
}

MakeText :: MakeText()
     : Control(new Message("       text       ")) {}

void MakeText :: Do()
{
  if ( gccontrol->isEditmode() ) {
    Highlight(true);
    gccontrol->WriteText();
    Highlight(false);
  }
}
MakeMove :: MakeMove()
     :Control(new Message("     move      "))
{}

void MakeMove :: Do()
{
  if ( gccontrol->isEditmode() ) {
    Highlight(true);
    gccontrol->MoveSymbol();
    Highlight(false);
  }
}
```

```
#include <command.H>
#include <grafcetcontrol.H>
#include <InterViews/menu.h>
#include <InterViews/message.h>
#include <libc.h>


QuitCommand :: QuitCommand()
     : MenuItem("Quit") {}


void QuitCommand :: Do() {
  exit(0);
}


NewCommand :: NewCommand()
     : MenuItem("New") {}


void NewCommand :: Do()
{
  if ( gccontrol->isEditmode() )
    gccontrol->New();
}


ReadCommand :: ReadCommand()
     : MenuItem("Open") {}


void ReadCommand :: Do()
{
  if ( gccontrol->isEditmode() )
    gccontrol->ReadGraph();
}


SaveCommand :: SaveCommand()
     : MenuItem("Save") {}


void SaveCommand :: Do()
{
  if ( gccontrol->isEditmode() )
    gccontrol->SaveGraph();
}


SaveAsCommand :: SaveAsCommand()
     : MenuItem("Save as") {}


void SaveAsCommand :: Do()
{
  if ( gccontrol->isEditmode() )
    gccontrol->SaveGraphAs();
}


DeleteCommand :: DeleteCommand()
     : MenuItem("delete")
{}


void DeleteCommand :: Do()
{
  if ( gccontrol->isEditmode() )
    gccontrol->DeleteSymbol();
}


RelationCommand :: RelationCommand()
     : MenuItem("Relation")
{}


void RelationCommand :: Do()
{
  if ( gccontrol->isEditmode() ) {
  }
}


StopCommand :: StopCommand()
     : MenuItem("Stop")
{}


void StopCommand :: Do()
{
  if ( !gccontrol->isEditmode() )
  gccontrol->Stop();
}


GoCommand :: GoCommand()
     : MenuItem("Go")
{}


void GoCommand :: Do()
{
  if ( gccontrol->isEditmode() ) {
    boolean syntax_ok = gccontrol->CheckSyntax();
    if ( syntax_ok ) {
      gccontrol->MakeRelation();
      gccontrol->Go();
    }
  }
}
```