

J.E.A. van Hintum

# Quality Constraints & Constrained Quality



## Quality Constraints & Constrained Quality

J.E.A. van Hintum

1. De grootste 'constraint' in de meeste constraint programma's is het (statische karakter van het) constraint systeem zelf. Hoe meer een constraint systeem 'unconstrained constraint' programmering mogelijk maakt, hoe krachtiger en breder inzetbaar het is <sup>[1]</sup>.

[1] dit proefschrift.

2. De 'Object Factory' is een eerste echte stap in de richting van modulaire software ontwikkeling (zodat willekeurige software componenten van verschillende software leveranciers aan elkaar gekoppeld kunnen worden) omdat de 'Quality Factory' niet probeert de communicatie tussen de verschillende componenten te uniformeren d.m.v. het inbouwen van vertaalfuncties (vgl. CORBA, COM of OLE), maar een omgeving biedt die de verschillende componenten met elkaar kan verbinden zonder dat de source-code van de oorspronkelijke component hiervoor hoeft te worden aangepast of voorbereid <sup>[2]</sup>.

[2] dit proefschrift.

3. Er zijn 4 criteria waaraan een programma moet voldoen voordat een deadlock op kan treden (mutual exclusive use of, partial allocation of, circular wait for and non-interruptable use of a resource) <sup>[3]</sup>. De afwezigheid van een deadlock aantonen door te bewijzen dat één of meer criteria niet gelden is relatief eenvoudig. Een belangrijke stap voor software ontwikkeling in parallelle programmeeromgevingen zou het formuleren van een theorie zijn die het bewijzen van de afwezigheid van 'dead-locks' mogelijk maakt, juist als het programma wel aan alle vier de criteria voldoet.

[3] E. Meijer, *Niet-sequentiële systemen*, KU Nijmegen, 1985.

4. De cut-operator (!) van Prolog <sup>[4]</sup> kan worden gezien als een nullary constraint relatie avant-la-lettre.

[4] LPA PROLOG Professional Compiler V2.0, *Programmer's Reference Manual, Edition 1*, pp. 30, Logic Programming Association Ltd. 1988.

5. In programmeertalen is de kwaliteit van de 'semantiek' en 'syntax' equivalent voor kwantiteit. De kwaliteit van de semantiek van een programmeertaal heeft een direct effect op de hoeveelheid tijd die nodig is om in een programma (geschreven in deze programmeertaal) de aanwezige fouten te vinden en de kwaliteit van de syntax van een programmeertaal heeft een direct effect op het aantal regels code die men moet schrijven om het programma te coderen.
6. Naar analogie van samenstellingen zoals 'steelpan', 'autostuur' en 'sportschoen' zou 'paspoort' eigenlijk 'poortpas' (of nog beter 'grenspas') moeten zijn. De verkeerde constructie van het woord 'paspoort' is er (onbewust) misschien ook de oorzaak van dat Nederland altijd problemen heeft gekend met het verwezenlijken van een betrouwbaar en fraudebestendig paspoort.



7. Tegenwoordig beschouwen gelovigen de Mens als een creatie van God terwijl zij die geen geloof aanhangen, god beschouwen als een creatie van de mens; iets wat radicaal indruist tegen het 'Godsbewijs' van Anselmus <sup>[5]</sup>. Dit toont aan dat bewijzen tijd- en tijdgeestafhankelijk zijn.

[5] St. Anselmus, Proslogion.

8. Wat met een wiskundige formule aan ruimte wordt gewonnen door compactheid en abstractieniveau staat in schril contrast tot die ruimte die weer nodig is om in een begeleidende tekst de formule uit te leggen. Dit geeft tevens de oorzaak aan waarom wiskundeboeken ofwel dun en onleesbaar zijn ofwel dik en onuitgelezen.
9. Het gebruik van de cut-and-paste methode leidt meestal tot meer extra werk dan dat het bespaart omdat toch vaak allerlei kleine dingetjes in de 'geknippte' informatie moeten worden aangepast aan de context waarin de informatie wordt 'geplakt'. In dat licht bezien is het vreemd dat steeds meer programma's de cut-and-paste mogelijkheid toevoegen aan de user-interface.
10. Een proefschrift dient voorzien te zijn van voldoende afbeeldingen opdat ook mensen die niet ingewijd zijn in het vakgebied (lees: familie) toch met enig genoegen zo'n proefschrift door kunnen bladeren.

# Quality Constraints & Constrained Quality



Technische Universiteit Eindhoven

# Quality Constraints & Constrained Quality

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor  
aan de Technische Universiteit Eindhoven,  
op gezag van de rector Magnificus, prof.dr. M. Rem,  
voor een commissie aangewezen door het college van Dekanen  
in het openbaar te verdedigen op  
maandag 3 november 1997 om 16.00 uur

door

**Johannes Everhardus Antonius van Hintum**

geboren te Grave

Dit proefschrift is goedgekeurd door:

1e Promotor: Prof.Dr.Dipl.Ing. D.K. Hammer

2e Promotor: Prof.Dr. S.J. Mullender

Co-promotor: Dr. I. Herman

De wet van Hanggi:

Hoe trivialeer je onderzoek is, des te meer mensen  
zullen het leren en het ermee eens zijn.

Gevolgtrekking:

Hoe belangrijker je onderzoek is, des te minder  
mensen zullen het begrijpen.



# Table of Contents

<b>Table of Contents</b>	<b>v</b>
<b>Acknowledgements</b>	<b>ix</b>
<b>Nederlandse samenvatting (Summary in Dutch)</b>	<b>xi</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Constraint Programming	3
1.2 Multimedia	4
1.3 Quality of Service	5
1.4 Human Perception	6
1.5 Research Topics and Goals	6
1.6 Overview	9
<b>2 Factors Influencing (Perceived) Quality</b>	<b>11</b>
2.1 QoS in Network Communication	13
2.2 Application Areas for Multimedia QoS	18
2.2.1 Color Systems and Images	19
2.2.2 Movies	21
2.2.3 Sound	22
2.3 Compression Techniques	23
2.3.1 Examples of Compression Techniques for Multimedia Data	23
2.3.2 Relevance of Compression Techniques for Resource Management	25
2.3.3 Relevance of Compression Techniques for Perceived Quality	26
2.4 A Perceptual Quality Constraint System: The Quality Factory	29
<b>3 Constraint Satisfaction and Constraint Systems</b>	<b>35</b>
3.1 Terminology	37
3.2 Constraint Networks	38
3.2.1 Direction of the Dependencies	38
3.2.2 Cyclic Dependencies	40
3.2.3 Priorities for the Dependencies	41
3.2.4 Cardinality of Dependencies	42
3.2.5 Dynamic Abilities of the Constraint Network	43
3.2.6 Number of Constraint Networks	44
3.3 Propagation Techniques	44
3.3.1 Incremental versus All-at-Once	44
3.3.2 Sequential versus Parallel	45
3.3.3 Refinement versus Perturbation	45
3.4 Satisfaction Techniques	46
3.4.1 Propagation of Known States	46
3.4.2 Propagation of Degrees of Freedom	47
3.4.3 Relaxation	47
3.4.4 Prototyping	48
3.4.5 Graph and Term Rewriting	48
3.4.6 Redundant Views	50
3.5 Constraint Systems	52
3.5.1 Alien	52



3.5.2 Equate	53
3.5.3 GICS	55
3.5.4 Kaleidoscope	55
3.5.5 Procol	56
3.5.6 ThingLab I and ThingLab II	57
3.5.7 Summary	58

## **4 The MADE Constraint System** **59**

4.1 The Rationales for Creating the MADE Environment	61
4.1.1 Active Objects in the MADE Model	62
4.1.2 Delegation in the MADE Model	62
4.2 The mC++ Programming Language	63
4.2.1 MADE Objects and Communication	64
4.2.1.1 Communication	64
4.2.1.2 Prototypes and Delegation	65
4.2.2 MADE Object Classes	65
4.2.2.1 Active Objects	66
4.2.2.2 Mutex Objects	67
4.2.2.3 Unprotected Objects	67
4.2.2.4 Thread Control and Message Servicing	67
4.2.3 mC++ Runtime Methods	68
4.2.3.1 Dynamic Call Interface	68
4.2.3.2 Delegation Interface	69
4.2.3.3 General Messages	70
4.3 The Concepts of the MADE Constraint System	70
4.3.1 Constraints in Multimedia Environments	70
4.3.2 Design Decisions in the MADE Constraint System	73
4.3.2.1 The Constraint User Interface (CUI)	74
4.3.2.2 The Meta Constraint Graph Object (MCGO)	74
4.3.2.3 The Constraint Graph Object (CGO)	75
4.3.2.4 The Constraint Object WRAPper (CO_WRAP)	76
4.3.2.5 The Constraint Object Class (CO)	76
4.3.2.6 The Router	77
4.3.3 Naming Conventions, Symbols and Notations	77
4.3.4 Network Configurations	79
4.3.4.1 Independent Connections	80
4.3.4.2 Dependent Connections	82
4.4 The Implementation of the MADE Constraint System	83
4.4.1 Creating a Constraint Class	85
4.4.1.1 Constraint Class Declarations	85
4.4.1.2 Constraint Class Definitions	86
4.4.1.3 Examples of Constraint Class Declarations and Definitions	88
4.4.2 Initializing the Constraint System	90
4.4.3 Building Up the Constraint Network	90
4.4.3.1 Registering a Dependent Object	92
4.4.3.2 Registering an Independent Object	93
4.4.3.3 Registering a Constraint Object	93
4.4.3.4 Examples of Adding Constraint Objects to the Network	93
4.4.3.4.1 Adding the SliderToDegree Constraint Object	93
4.4.3.4.2 Adding the CFK Constraint Object	94
4.4.3.4.3 Adding the DegreeToClock Constraint Object	95
4.4.4 Breaking Down the Constraint Network	97
4.4.4.1 Temporary Breakdown	98
4.4.4.2 Permanent Breakdown	98
4.4.5 Triggering a Constraint Object	98

4.4.6 Performing a Renewed-Triggering Action on a Constraint Object	102
4.4.7 Constraint Satisfaction and Information Propagation	104
4.4.7.1 Executing the Constraint Function	104
4.4.7.2 Making Execution Plans	106
4.4.7.3 Satisfaction Using Execution Plans	108
4.4.8 Satisfaction and Propagation in other Configurations	112
4.4.9 The Constraint Network and the Database	113
4.4.10 Notification of Constraint Satisfaction	115
4.5 Performance of the MADE Constraint System	116
4.5.1 The Translation from mC++ to C++	116
4.5.2 The Performance in a MADE Constraint Network	120
4.6 Conclusions	123
<b>5 The Quality Factory</b>	<b>127</b>
5.1 Services	130
5.2 Filters	133
5.3 Formats	135
5.4 Resources and Negotiation	138
5.5 Software Reuse	140
5.5.1 Abstraction	141
5.5.1.1 Function Descriptions	141
5.5.1.2 Argument Relations	145
5.5.1.3 (Dynamic) Contexts	148
5.5.1.4 Summary	151
5.5.2 Selection	151
5.5.3 Specialization	153
5.5.4 Integration	153
5.5.5 Summary	154
5.6 Propagation of Information	154
5.7 Towards Constraint Relations	156
5.7.1 Creating Objects for the Arguments	156
5.7.1.1 The Argument Object	156
5.7.1.2 The Result Argument Object	158
5.7.1.3 The Fields of a Reference	160
5.7.1.4 Execution of Argument Relations in Normal and Dynamic Contexts	162
5.7.2 The Bookkeeping of the Argument Objects in the Quality Factory	164
5.7.3 Definitions of the Constraint Classes	165
5.7.3.1 The ConstrCO Constraint Object	165
5.7.3.2 The DeconstrCO Constraint Object	166
5.7.3.3 The EvalCO Constraint Object	167
5.7.3.4 The SyncCO Constraint Object	168
5.7.3.5 The ExecCO Constraint Object	169
5.7.3.6 The DefCO Constraint Object	170
5.7.3.7 The RstCO Constraint Object	170
5.8 Performance of the Quality Factory	171
5.9 Conclusions	172
<b>6 Discussions and Conclusions</b>	<b>177</b>
6.1 Discussions on the MADE Constraint System	181
6.2 Discussions on the Quality Factory	184
6.3 Conclusions	186
<b>Annex</b>	<b>187</b>
A Description of Provided and Requested Services	189
B Specification of the Argument Relations	193

C The Negotiation Objects	195
D Overview of the Constraint Networks	197
<b>References</b>	<b>211</b>
<b>About the Author</b>	<b>223</b>

# Acknowledgements

The work described in this dissertation is the result of research I did while employed at the CWI in Amsterdam. Many people have aided this effort amongst whom I would like to mention few especially.

First of all, thanks go out to the many co-workers of the MADE project for their valuable comments on the implementation of the MADE system in general and the MADE constraint system in particular. Of those co-workers, special thanks go out to my colleagues at the CWI which I would like to mention by name: Farhad Arbab, Michal Haindl, Frans Heeman, Han Noot, Graham Reynolds, Bèhr de Ruiter, Frank van Dijk and Arno Siebes.

Many thanks to Remco Veltkamp, who supported and guided me during my research when it threatened to come to a standstill, and to Paul ten Hagen, who was always prepared to discuss my research with me and who produced a never ending stream of new thoughts and ideas which prevented me to restrict my attention to only the small area of constraint programming and forced me to widen my horizon and see the possible applications of constraint programming in every day life.

Furthermore, I want to mention all the members of the former department Interactive Systems at the CWI. In this respect I want to mention all my roommates (of which were many): Edwin Blake, Frans Heeman, Richard Kelleners, Benjamin Rhille, Graham Reynolds, Bèhr de Ruiter and Lloyd Rutledge. They all provided me with the necessary distractions when I got stuck over some annoying programming error or when the computer got stuck over a broken name file server.

I would also like to thank the people at the Digital Media Engineering department of SUN Microsystems Laboratories which helped me during my two month stay in Mountain View, CA, USA: Brent Browning, Kate Heinlen, Gail Lowell, George Salem, Nannette Simpson and Robert Watson. In this respect I would also like to thank Emil Sarpa, head of the extern relations at SML, for making my stay there possible.

Special thanks go out to Karin van Dun who offered to proofread this dissertation and correct all the errors in the use of prepositions, the structure of the sentences and the use of conjugations.

Finally I wish to thank my girl-friend Tineke for putting up with me during periods when she could only find me working behind my computer. I will never forget and always appreciate all the domestic tasks she did instead of me so I could continue working.



# Nederlandse samenvatting

## (Summary in Dutch)

In deze dissertatie wordt een systeem beschreven dat gebruikers van multimediaspresentaties in staat stelt om opdrachten te geven aan dit systeem met de doelstelling de kwaliteit van de gepresenteerde informatie aan te passen aan de eigen wensen. Wat dit systeem zo bijzonder maakt is dat deze opdrachten mogen worden geformuleerd in termen van de eigen waarnemingen. Dit eist natuurlijk aanpassingen aan het onderliggende systeem zodat dit de waarnemingen van de gebruiker kan 'vertalen' naar opdrachten op het niveau van de onderliggende programmatuur die met opdrachten van veel technischer aard moeten worden aangestuurd. Deze vertaalslag tussen de opdrachten op basis van waargenomen kwaliteit en de opdrachten op basis van technische specificaties vereist een zekere 'speelruimte'; er bestaat in het algemeen geen 1 op 1 vertaling van de verschillende niveaus en de resulterende 'vertaling' zal dan ook altijd een benadering zijn van de oorspronkelijke opdracht. Deze benodigde speelruimte is aanwezig in de waargenomen kwaliteit; wat in technische termen een lagere kwaliteit is, hoeft in termen van waargenomen kwaliteit niet slechter te zijn of is soms (objectief gezien) niet slechter. Een grafische afbeelding van 300x200 punten die wordt weergegeven op een monitor die maar 65.535 van de technische mogelijke 1,6 miljoen kleuren tegelijkertijd kan weergeven, is niet van de best mogelijke *technische* kwaliteit; echter de waargenomen kwaliteit is optimaal aangezien deze afbeelding maar 60.000 punten heeft (minder dan het mogelijke aantal kleuren dat tegelijkertijd kan worden weergegeven) zodat geen concessies hoeven worden gedaan ten aanzien van het aantal kleuren dat in de afbeelding gebruikt kan worden. Het voordeel van deze vertaalslag is dat ook gebruikers zonder technische achtergrond met het systeem kunnen werken.

Een andere manier waarop de aanwezige speelruimte kan worden gebruikt is door de exacte 'vertaling' af te laten hangen van andere processen die reeds op de computer draaien. Elk proces op een computer gebruikt zogenaamde 'resources': middelen waarmee een bepaalde taak kan worden volbracht. Voorbeelden hiervan zijn geheugenruimte, de rekenkracht van de processor, netwerkverbindingen met andere computers etc. Vaak heeft een proces meerdere van deze resources nodig om zijn taak uit te kunnen voeren en heel vaak heeft het proces niet *alle* resources die er zijn nodig. Bovendien kan een proces op meerdere manieren van de verschillende resources gebruik maken; als meer geheugenruimte en rekenkracht van de processor mag worden gebruikt, kan het zijn dat het proces minder beslag hoeft te leggen op de netwerkverbinding (of omgekeerd). Er zijn dus mogelijkheden om te schuiven met de mate waarin processen beslag op verschillende resources. Zeker wanneer er meerdere processen draaien op een computer zal er een strijd ontstaan tussen deze processen om beschikking te kunnen krijgen over de aanwezige resources. Het is dus mogelijk dat door een ongunstige toekenning van resources aan processen slechts een aantal processen tegelijkertijd hun taak kunnen vervullen, terwijl een 'slimme' toekenning juist het aantal processen die tegelijkertijd hun taak vervullen kan optimaliseren. Door nu tijdens de vertaling van opdrachten op basis van waarnemingen naar opdrachten op basis van technische specificaties ook te kijken naar de toekenning van resources kan een beter presterend systeem (zowel in termen van waargenomen als technische kwaliteit) worden gerealiseerd.

Het blijkt nu dat het hierboven beschreven systeem zo krachtig is dat het ook gebruikt kan worden om andere problemen op te lossen zoals het koppelen van losse softwaremodules van verschillende leveranciers (iets wat nog steeds een probleem is). In feite kan dit koppelen gezien worden als een vertaling van de visie van de ene module naar de visie van de andere module (vergelijk de vertaling van waargenomen kwaliteit naar technische kwaliteit). De waarde van het systeem, dat is beschreven in deze dissertatie, blijft dan ook niet alleen beperkt tot multimediaspresentaties maar kan veel breder worden gezien in het licht van modulair software-ontwerp.

De uitgangspunten in deze dissertatie zijn altijd de elegantie en flexibiliteit van het systeem geweest en niet de efficiëntie en snelheid van het systeem. Het resulterende systeem is dan ook niet zozeer een complete en afgeronde implementatie van de ideeën zoals hierboven beschreven, als wel een eerste aanzet tot de realisatie van echt modulair software-ontwerp.

# CHAPTER 1

## Introduction





In this thesis four notions will play an important role. The first notion is *constraint programming*, the second is *multimedia*, the third is *quality of service* and the last notion is *human perception*. These four notions can be merged and used in one application in such a way that this application can accept quality of service requests in terms of the user's perception of the presented multimedia data and translate these requests automatically (using constraint technology) into requests in terms of a specific application.

In this context, it may be helpful to explain the relation between the image on the cover page and the contents of this thesis. On the cover, three marionettes are shown, together with a number of strings which connect the limbs of these puppets to wooden cross-like control devices. The puppeteer can use the control device to make the puppets move. However, the way the puppeteer has to alter the position of the control device may, at first glance, have little relation with the way in which the puppets move; i.e. the puppeteer, who only sees the control device, has a different view on the movements of the puppets than the audience which only sees (the limbs of) the puppets move and not the puppeteer or the control device. The viewer sees that 'an arm is raised' or 'a foot is lifted' whereas the puppeteer knows that he has to tilt (a particular end of) the control device at a particular angle. Thus it is important that the movements of the control device are translated to movements of the limbs of the puppets. This is done by the strings; altering the position of (one of the ends of) the control device results in the movement of one or more limbs of a puppet due to the fact that the strings keep the distance between its two ends the same at all times.

The show, in which the puppets are used, can be compared with the presentation of multimedia data by an application. The human perception relates to the view of the audience which is limited to the movements of the limbs. The viewers of the puppet show sees that 'an arm is raised' or 'a foot is lifted' but they have no knowledge of either the control device or the way in which this control device has to be handled the same way as the viewer of the multimedia presentation does not see which actions are precisely taken by the application. The view of the puppeteer represents the view of the application. In order to make the puppets move, the puppeteer has to change the position of the control device without the need or ability to see how they actually move. The constraints are implemented by the strings; the position of the ends of the control device are 'automatically' translated into movements of the limbs.

What this comparison should make clear is that the viewer of the puppet show/multimedia presentation can have a different view upon what is presented by the puppeteer/application than the view of the puppeteer himself/application itself. In order to bridge this gap between the two views some kind of mechanism (strings/constraints) is needed to translate one view into the other.

In the remainder of this chapter, the four notions mentioned at the beginning of this introduction will be elaborated on, the research topics and goals of this thesis will be identified and a structure and overview of the remaining chapters will be given.

### 1.1 Constraint Programming

Constraint programming is an area with relations to a lot of other disciplines; the principle of constraint programming can be briefly summarized as:

**specifying the behavior of an application by defining dependency relations between a number of objects using a constraint system specific notation which enables this constraint system to maintain these relation without explicit intervention of the program in which these relations are specified**

This principle can be applied to a very diverse group of situations. This is, among other things, due to the fact that the description of constraint programming is very general. The disadvantage of this general character of constraint programming is that totally different views exist on what is considered to be a constraint and a constraint system. For instance, a clear distinction is made between Constraint Logic Programming ([Hentenryck 91], [Lassez 87]) and more imperative based constraint programming ([Borning et al. 87], [Freeman-Benson 90], [Hintum et al. 95]). Where the emphasis in CLP languages is on reducing the search space by labeling subtrees of the searchtree as irrelevant for the solution of the (constraint) problem, the imperative constraint languages focus more on the propagation of information (of changes made to the constrained objects) along the constraint relations.

Another consequence of the general character of constraint programming is that it is difficult to design a constraint system that is capable of handling these general constraints and be efficient at the same time. In practice, several constraint systems are designed which focus on only a small area in which they can maintain the constraint relations in a quick and reliable way. Due to these kinds of design decisions, the practical usability of these constraint systems is reduced; they can only be used for limited purposes. In the area of multimedia and multimedia presentations, the number of areas in which constraints can be applied is very broad. Multimedia is a notion which covers a lot of disciplines within computer science and in most of these disciplines constraints can be applied. Therefore, a constraint system for multimedia purposes has to be very general so it can be used in all those different disciplines but, at the same time, has to be powerful enough to support all the specific details of each of the individual disciplines.

In the remainder of this thesis it will be argued that constraint programming provides an elegant way to model system behavior which also allows for a clear maintenance of programs due to the relatively high level of abstraction used to describe constraint relations.

### 1.2 Multimedia

Applications that produce multimedia presentations are programs that, strictly speaking, use more than one kind of media to show information to the user of that program. Multimedia can be described as:

**the definition, manipulation and presentation of digital information  
and the relations between the different pieces of information**

In general, the following digital media can be distinguished: text, (3D) audio, (3D) still computer graphics, (3D) moving computer graphics (animation), still images, moving images (including video). Beside these media, possible future extensions to this group could be formed by smell, tactile, taste, etc. Important in the field of multimedia is the fact that it can deal with a broad range of media which have very different characteristics.

Different multimedia standards and systems have already been developed or are being developed to assist a programmer in using the different media in a multimedia presentation. Standards include JPEG ([JPEG 93]), MPEG ([MPEG 94]), MHEG ([Price 93]) and HyTime ([Biezunski 92], [DeRose et al. 94]). At the moment much research is done with respect to compression images using wavelets ([Hilton et al. 94]). Packages which are already on the market are ScriptX and GainMomentum which provide a programming/development environment to create multimedia presentations in which the different kinds of media can be integrated. Also in the MADE project ([Arbab et al. 93a],[Arbab et al. 93b],[Heeman et al. 93], [Herman et al. 94]) a number of tools

and utilities were designed which can assist a programmer in making multimedia presentations. The whole set of tools and utilities together (media-editors, media-players, browsers, data-structures, preprocessor for C++ language-extensions) provide an environment in which different media can be used and combined in existing and new programs.

PREMO (Programming Environment for Multimedia Objects) ([Atkinson et al. 95], [Blake et al. 95], [Correia et al. 95], [ISO-N1190 95]) also is a multimedia system which is being designed to provide means to define the way in which information is presented. It does so by creating a front-end to another application. This front-end is not a part of the application itself, but has a close relation to it and communicates and exchanges information with it. What sets PREMO aside is that it addresses the aspects of distribution. Influences of CORBA ([OMG 95]), COSS, COM ([Microsoft 94]), DAVIC ([DAVIC-1 95], [DAVIC-2 95], [DAVIC-3 95], [DAVIC-4 95], [DAVIC-6 95], [DAVIC-7 95], [DAVIC-8 95], [DAVIC-9 95], [DAVIC-11 95]) and MSS ([IMA 92]) can be found in the PREMO proposal; data, as part of the multimedia presentation, may be stored at different physical locations and its lifetime-cycle (the status of the data between the time it is created and the time it is destroyed) may be controlled by others than the PREMO front-end. This aspect of distribution in multimedia systems ([Koegel 94], [Little 94]) is becoming more and more important as the size of the data increases as well as the demands put on the quality of that data. Storage space is too expensive or is not available at all to simply make a local copy (i.e. a copy of the data at the physical location of the computer at which the multimedia presentation is viewed) of the data used in a multimedia presentation.

All the above mentioned multimedia systems (as well as other systems not mentioned above) try to provide environments where relations between different media can be managed; this thesis presents arguments that constraint programming is a very elegant way to support this management.

### 1.3 Quality of Service

Quality of Service deals with the possibility to specify with what kind of quality a certain functionality should be provided:

**the ability to define a required level of precision and/or detail of a certain service and the certainty that this requirement can and will influence the realized level of precision and/or detail**

The desired quality can be specified via a so-called request. In general, the desired quality will probably always be the highest possible quality. However, there are a number of situations, especially in multimedia presentations, where a lower quality for some of the functionality is desired. These situations do not occur when only one single video is displayed or when resources (like bandwidth of the network, cpu-time, memory, etc.) are always sufficiently available. However, in the case where one has to pay for the use of resources, a lower quality may lead to less costs due to the fact that a lower quality may need less resources for a shorter time. Also when resources have to be shared, the quality of some of the functionality may be reduced so other functions can be provided with a higher quality. Such a situation occurs when, for instance, audio and video have to be sent over the same network connection and the bandwidth of the connection does not allow for both signals to be sent over at the same time. In that case, video may be delivered at a lower quality to make sure that the audio can be played at an acceptable level of quality.

Quality of Service systems provide means to control the quality of the provided functionality. Some of these systems allow for a pre-set quality (the quality is specified in advance and cannot be altered once the function is activated) whereas other systems allow for dynamic alteration of the delivered quality. These last systems are also capable of negotiating between the different quality

requests for different functionality which share the same resources. In those cases, a previous requested quality level may have to be degraded in order to service the new request.

Again, the key notion in Quality of Service are relations; relations between multimedia data and between multimedia data and resources. Also here, constraint relations can be used to specify these relations.

### 1.4 Human Perception

An important notion in multimedia presentations is human perception. In general, the human perception can be described as:

**the ability of a human beings to sense signals and collect information from their environment**

In (multimedia) presentations it is not necessary to present information that will not be perceived by the viewer. Because there are situations where the human senses are not able to perceive all information that is, in theory, present in their environment, (slightly) deformed data can be displayed without the viewer actually noticing that the quality is degraded. This notion is already used in several multimedia standards for audio and video compression. During the compression some of the information of the sound/image may be lost and, as a result of that, the quality may be degraded. However, this loss in quality is limited in such a way that the average human being will not be able to make a distinction between the original uncompressed data and the data which is first compressed and decompressed before being presented. It is important to note that what is considered to be 'no perceived quality loss' can differ from person to person and from situation to situation.

The advantage of compressing data (and thus presenting a lower quality) is that the usage of scarce resources can be reduced. Compressed data may need less memory to be stored, less bandwidth to be sent over a network connection and less cpu-time to be processed. However, these advantages should at least balance the resources needed to compress and/or decompress the data.

Another aspect that deals with the human perception is that, because things can appear to be different from what they really are, a viewer can issue a quality request using a different view than the application is using to present the data. The structures used to store data of a multimedia presentation (like frames of a movie, the samples of a sound, etc.) may not be visible to the viewer. Therefore, the perception of the viewer to a great extent influences how a quality request can and will be formulated.

Also in this case, the key notion is the specification of the relations between the capabilities of the human sense and the quality of the presentation of the desired information. Also here, constraints can be used to describe these relations.

### 1.5 Research Topics and Goals

Literature in computer science recognizes that constraint programming is a powerful tool to describe and solve complex problems. This power lies in the fact that the formalism to describe the problem in terms of constraints is small and simple and that it abstracts from how to solve the problem and focusses on what problem to solve. Nevertheless, the use of constraint programming is still limited to a few isolated areas. Constraint systems exist for scheduling, geometric computations,

interval computation, algebraic systems, boolean systems etc. ([Bordegoni 92], [Borning et al. 86], [Davis 87], [Gosling 83], [Hardman et al. 92], [Hentenryck 91], [Hentenryck et al. 91], [Leler 88], [Maloney et al. 89], [Nelson 85], [Rankin 91], [Velkamp et al. 92], [Wilk 91]) However, constraint systems which can support constraint programming in all these areas at the same time are hard to find. The reason for this is that either the constraint relations are predefined (i.e. the constraint system recognizes and supports only a fixed set of operations) or the objects/structures/data formats are predefined. By restricting the relations and/or the data that a constraint system supports, it is possible to include application-area specific information in the constraint system to help it interpret the abstract descriptions of the constraint problem and translate these descriptions into concrete algorithms.

In this thesis, it will be shown that it is possible to create a constraint system that does not have the application-area specific knowledge, can support a broad range of constraint problems and is still powerful enough to support all the different constraints in the various application areas. To be able to do so, the constraint system that is proposed in this thesis does not have any predefined constraint relations. Every constraint relation that is needed first has to be programmed. A constraint function is the programmed version of the constraint relation, bound to a certain programming language. In the remainder of this text, all program fragments will be written in mC++<sup>1</sup> (unless specified otherwise). In the constraint function, code is written which focuses on how this specific constraint relation is to be maintained; i.e. a constraint function can inspect data which is important for this constraint relation and adjust other pieces of information accordingly. The programmer can make use of special constructs which are provided by the constraint system. These allow the programmer to check the status of the various constraint relations and provide important information to write code which takes into account the status of the various other constraint relations. These constraint functions can now be used in the abstract descriptions of the constraint problems. Note that each constraint function can be written as a self-contained piece of code which exactly specifies how a single particular constraint relation should be maintained and which can thus be reused as part of a library.

One of the important subjects of this thesis is the design of a general constraint system which is able to support a broad range of constraint problems. As already mentioned this design is complex and it is the complexity of the design this thesis is focussing on rather than on defining (large) class-libraries. The design was primarily guided by the requirement to make a flexible and elegant system rather than an efficient one. Therefore this thesis will not present an in-depth analysis of the performance of the system, but focuses on its functionality; relatively much attention is paid to constructs which give the system its flexibility and elegance. Concepts such as delegation and parallelism turned out to be important to achieve this and are, therefore, discussed in detail.

The choice to implement a constraint system for a multimedia environment was inspired by the fact that multimedia is a very broad application area in the sense that it merges a number of fields in computer science and even fields outside computer science. Due to the fact that all these different areas come together in the field of multimedia, a multimedia constraint system has to be able to support constraint relations from all those areas. Therefore, the multimedia environment is a good place to start with a general constraint system. Furthermore, a constraint system can be very valuable in a multimedia environment as different media have to be coordinated to form a multimedia presentation. The relations between the various media can be modelled very well using constraint relations. However, the results of the research described in this thesis can be generalized and applied to other application areas.

---

<sup>1</sup> The language mC++ is the programming language developed in the framework of the MADE project. It is an extension of the C++ programming language. More details on mC++ are given in Chapter 4, §4.2.

Another problem, which is more specific to constraint systems in object-oriented languages, has to do with the fact that objects in object-oriented languages tend to encapsulate (part of) their data whereas constraint relations often need to have access to this data. This makes it hard for a constraint system to monitor whether the value of a certain variable has been changed (and if so to take the appropriate actions to maintain the various constraint relations). Some object-oriented constraint systems try to solve this by not allowing an encapsulated variable to be constrained by a constraint relation while others require that constrained objects provide certain functions in their interfaces so the constraint system can assume the presence of these functions and can use them to maintain the constraint relation. In this thesis, the problem of encapsulation will be dealt with in another way, which puts less restrictions on (the interface of) the object. Because mC++ is an object-oriented language, changes to (the data of) an object always occur by invoking one of the member functions of that object. This fact is used by the constraint system to monitor when constraint functions have to be activated. In this context, delegation proved to be very useful; member functions are hooked to the constraint system by means of delegation. This allows a programmer to constrain existing objects without the need to modify the code of the object itself.

Finally, this thesis describes an application in which the constraint system is used to its full potential. This application is the so-called Quality Factory. The Quality Factory provides a number of features which can be considered essential in multimedia environments. As mentioned before, several standards exist in the field of multimedia. Nowadays, applications are built which support a small set of, one or only part of one such standard. This can cause problems when, for instance, a drawing program can only produce images in the format *GIF* and the word processor can only import images in the format *Encapsulated PostScript*. In order to be able to use the images of this particular drawing program in this word processor, a converter is needed to convert the *GIF* format to *Encapsulated PostScript*. In principle, it is possible to connect these applications in such a way that the drawing program is connected to the converter and the converter connected to the word processor. The output of the drawing program is then used as input for the converter and its output is, in turn, used as input for the word processor. In practice, several problems may arise when this is attempted. The Quality Factory uses constraints to realize the connections between the several applications and makes sure that (part of) the output of one application can be used as (partial) input by another application. In this way, the Quality Factory tries to take another step towards software reuse.

This same approach can also be used for another purpose. As pointed out earlier, the human perception of a multimedia presentation can be different from the view that an application has on the data of the multimedia presentation. The Quality Factory can connect applications, which communicate in terms of the viewer of a multimedia presentation, via converters to the applications which handle the definition, manipulation, and presentation of the data in the multimedia presentation. In this way, the viewer of a multimedia presentation does not have to have any understanding of how the data is stored, structured or organized. In his own words, the viewer can issue quality of service requests which will then be translated, by the converters, into requests in terms of the applications which really handle the multimedia data. Because the Quality Factory can make the connections between converters and applications transparent for both sides, no special adaptations have to be made to any side.

In this thesis two propositions are introduced and defended:

- viewers of a multimedia presentation have no knowledge of the resource claims done by that presentation and, therefore, a translation scheme should be provided which translates the notions of these viewers (base upon their perception) into the more technical notions (used by most applications) to provide a better user interface.

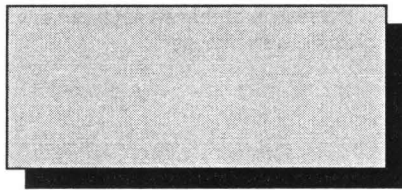


- constraint programming may provide an elegant and interesting solution to the problem as described in the previous proposition.

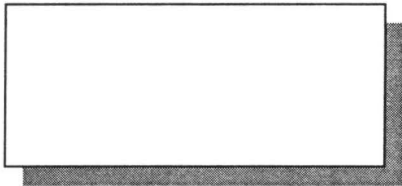
Several approaches have already been suggested and used to solve the problem described in the first proposition. However, none of these approaches used constraint programming as a possible way to tackle the problem. This text describes how constraint programming can be used as the basis for Quality of Service management and it shows that constraint programming allows for a gradual adaptation of the quality of the presentation. It is important to note here that the possible solutions (as provided by the constraint-based Quality-of-Service manager) are not always useable in practice; hardware may impose constraints on what can actually be realized. Therefore, some of the examples given in the text put more emphasis on the flexibility of the constraint-based approach than on the practical use of the solution. The goal of this research is not to present the best possible solution to for Quality-of-Service management but the best possible solution for constraint-based Quality-of-Service management; whereas other approaches may provide more efficient, practical and better solutions, the constraint-based approach may provide a more flexible solution which is also elegant in terms of the interface of the viewers of a multimedia presentation. The Quality Factory is a specialized system to provide perceptual based Quality of Service, but the approach used for this application can be generalized for other areas too; modular software design, software reuse and proving the correctness of program are three examples which can benefit from the approach taken in the constraint-based framework of the Quality Factory.

## 1.6 Overview

The remainder of this thesis is divided into five chapters. In this section, for each of the chapters a short description of their contents is presented. Before this is done, something will be explained about the layout used in this text. Throughout the text several sections are inserted which have the following appearance:



These sections contain some source-code which is used to explain what is said in the text. The source-code in these sections will, in most cases, not be complete (part of the code is left out (indicated by '...')). These code-fragments are not real examples in the sense that they are not part of a working program but they are used to show roughly how the ideas presented in the text can be realized in actual programming code. Another kind of layout is used for the real examples (i.e. parts of the source-code of real working programs). These sections can be identified by the following appearance:





Chapter 2 describes the problems that occur when the user's perceivable quality of service is addressed. Current Quality of Service systems for network communication do not address this issue. These systems do not satisfy the user's needs in the sense that human beings have a different view upon, for instance, a multimedia presentation than a computer application. In this chapter, it is argued that an new approach is needed. In this new approach, constraints play an important role.

Chapter 3 gives an introduction to constraint systems. It introduces and explains some terminology in the field of constraint programming and describes several of the most common information propagation and constraint satisfaction techniques. Finally, this chapter gives the reader an impression of a number of object-oriented constraint systems which already exist.

In the next chapter, a specific constraint system is described in more detail; the MADE constraint system. This constraint system has been designed and implemented in an object-oriented environment and is intended for managing constraint relations in multimedia presentations. Because it was designed for multimedia presentations, it has a versatile character which not only proves to be very useful in these kind of environments but also necessary. Throughout the text the working of the MADE constraint system is explained by an example program. This example program illustrates the working of the different parts of the constraint system in detail as well as the information which is maintained by the constraint system and which is used to satisfy the different constraints.

Chapter 5 describes the details of the Quality Factory; the notions which are important to be able to specify the connections between user requirements and system requirements (services, filters, formats, resources and negotiation), the ideas behind the reuse of existing software in the Quality Factory and the relations of the Quality Factory with the MADE constraint system. This chapter also describes the working of the Quality Factory by going through an example where the user specifies his requirements with respect to a certain application and where these user's requirements have to be translated to system requirements.

Finally, Chapter 6 will present the results of the research done in this thesis and discuss whether the goals, as formulated in this chapter, were reached.

# CHAPTER 2

Factors  
Influencing  
(Perceived)  
Quality



Quality of Service is a general concept; it deals with the realization of certain services at a certain, requested level. The quality of a service may be altered either to a higher or to a lower quality. The request for a higher quality may result from the desire to get a quicker, more accurate, more detailed or more steadily supplied result from a service. A request for a lower quality may be used to allow other services to provide higher quality results; the demand for a lower quality will result in lower claims on the capacities of processor, memory, network etc. so other services can use these resources (too). To manage the qualities requested from the different services and the qualities that can be realized by the different services, so-called Quality of Service (QoS) Systems are designed. An area in which a lot of research is done with respect to QoS is network communication. Several systems are designed to control the quality of delay and jitter and the (reservation of) bandwidth. These systems will be discussed in §2.1. The problem with these kinds of QoS systems is that they use metrics which are convenient to manage the different resources (bandwidth, jitter, delay, etc), but which are not always intuitive to the user of a program. Especially in multimedia presentations, the user will have a different view on the data than the system that presents this data. In §2.2 the reader is given some examples of multimedia application areas where QoS can be applied. The examples described in this section are not intended to be complete or innovative. They merely show some of the problems which the Quality Factory has to face with respect to human perception in multimedia environments. §2.3 shows a number of different compression schemes which can be used to compress multimedia data. Again, this section does not give a complete list but wants to make the user aware of some of the possible methods which can be used to reduce the size of the multimedia data. In this section it is also explained why compression is so important for resource management and perceived quality. In §2.4, a new approach is proposed which takes into account the considerations concerning the human perception as expressed in §1.5.

### 2.1 QoS in Network Communication

QoS in network communication is very important for distributed multimedia systems; it guarantees the soundness of the presented data and forms the basis for real-time performance ([Staepli et al. 95]). However, with these systems alone, the demands of a multimedia presentation on the bandwidth of network connections is very high; network communication programs tend to claim more resources than they, strictly speaking, need to be able to compensate for possible (future) flaws in their communication. In order to provide feasible and cost effective solutions, multimedia systems must be able to handle more than just QoS in network communication; additional mechanisms have to be used to make the flow of information more manageable ([Steinmetz 94a]). It may be concluded that network communication QoS systems are only one of the means that may provide an overall quality of service. In this section the general ideas of QoS in network communication are presented. In the remainder of this thesis, these kinds of QoS systems are considered to be available; they are one of the possible types of programs for which the viewer's requirements are translated by the Quality Factory. It is important to realize that the overview that is given in this section discusses systems which do not support perceived quality. The purpose of this overview is to give the reader an impression of the current QoS systems and their limitations.

For the area of network communication, a large collection of QoS systems has already been designed ([Demeure et al. 96], [Hutschenreuther et al. 96], [Knightly et al. 96], [Leydekkers et al. 96], [Lima et al. 96], [Nahrstedt et al. 96], [Rolia et al. 96]). Some of these systems are especially designed to be used in multimedia environments whereas other systems have a more general design. With respect to QoS in network communication several aspects can be distinguished. In [ISO-N9309 95] and [ISO-N9681 95] a number of aspects are described. The list below summarizing these aspects shows that the concerns of the ISO where not based on the perceptual aspects of network communication:

- Time

Characteristics which fall under *time* are the date/time when a certain event occurs, the time delay between the occurrence of a particular event after another event has occurred and the time interval in which some data is considered to be valid.
- Coherence

*Coherence* characteristics indicate whether the elements of a data collection are coherent in time, in space or in both time and space. Temporal coherence characteristics may indicate whether the elements of a data collection are produced, transmitted or received within a certain time interval. Spatial coherence characteristics may indicate whether duplicates of a data collection (presented at several different nodes in the network) are identical or not.
- Capacity

*Capacity* characteristics include throughput (the amount of data transmitted over a connection during a certain fixed time interval), processing capacity (which can be split in system throughput, which expresses the number of instructions during a fixed time interval, and operation load, which is defined as the ratio of used capacity and available capacity) and burst capacity.
- Integrity

Characteristics with respect to *integrity* deal with the probability that a certain error does (not) occur. These errors include addressing errors, transmission errors, connection errors and errors in recovering from an error.
- Cost

*Cost* characteristics are means of assigning a value to an object. These values do not necessarily have to be expressed in currency/unit and also do not have to represent an absolute value. Furthermore, cost characteristics can be influenced not only by the actual costs but also by the need for investments to provide the necessary facilities. Examples of these characteristics are user costs, resource costs, communication costs, event costs and data costs.
- Security

Characteristics for this group include protection against unauthorized access to a resource, unauthorized access to data and unauthorized viewing of data and means to establish the authenticity of the user.
- Reliability

Availability is an important characteristic from the *reliability* group. It specifies the proportion of time that satisfactory service is available. It includes the availability of connections and processes, but it also includes measurements for Mean-Time-Between-Failures, Mean-Time-To-Repair, fault tolerance (the impact of component error/failure on the provided service) and fault containment (the ability to provide services when one or more errors/faults have occurred).
- Other

Precedence is the only characteristic defined under this aspect. It can be used to assign the relative importance or the urgency to a certain event.

Although the groups mentioned above are identified and known, most systems which have been implemented are only concerned with the end-to-end delay (i.e. the time between the moment a request is made and the moment the results of this request are perceived). Therefore, most of these systems deal with throughput, delay (of various kinds), jitter and buffering. For the other aspects more research is still necessary. In this respect, it is worth mentioning that for the aspects of cost, safety and security much research is going on ([DAVIC-1 95], [DAVIC-2 95]). This primarily has to do with the fact that these are essential parts of upcoming digital services like Video-on-Demand, electronic shopping over the internet, money transactions over the internet, etc.

It is possible to divide the aspects, as described above, into two groups; the group of functional aspects (Coherence, Capacity, Integrity and Cost) and the group of non-functional aspects (Time, Security and Reliability). For each of these two groups QoS requirements can be made and both types of QoS requirements are important with respect to the perceived quality. In this text however, the QoS requirements for the group of non-functional aspects will not be discussed. This means that requirements with respect to, among other things, latency, the duration of a presentation, the synchronization of a number of media or the way in which data should be accessed to guarantee its security and reliability are not treated. Note however, that the non-functional aspects can be treated in a similar way as the function aspects are treated in this text.

Besides the different characteristics which can be specified for a quality request, the quality request can also be given a priority. The most commonly used priority levels are (from highest priority to lowest priority) *guaranteed*, *statistical* (also called *predictive* or *compulsory* [sic]) and *best effort*. In the first case, the QoS system has to reserve enough resources so, in all cases, the requested quality can be delivered. This means that, in most cases, more resources than are strictly necessary to deliver the requested quality will be allocated. However, to be able to provide the requested quality, the QoS has to reserve resources for the worst possible case; i.e. for the maximum workload as defined by the traffic model ([Vogt 95]). This situation leads to underutilization of the system and needless rejection of new reservation requests. In the case of best effort priority level, no reservations are made. Whenever some resources are available, these can be used. However, when these resources are needed for another higher priority request, best effort communication is suspended until resources become available again.

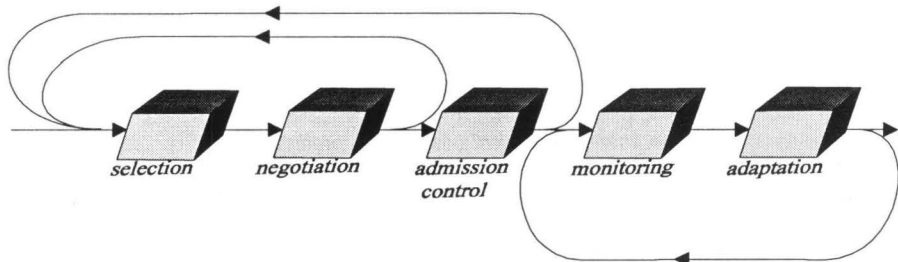
An important notion in QoS for network communication is reservation of resources. Depending on the resources which can be used, different qualities for different aspects of network communication can be realized. Three well known systems in this respect are IIS (which will be discussed later), ST-II ([Delgrossi et al. 94], [Delgrossi et al. 95], [Rajagopal et al. 96]) and RSVP ([Braden et al. 96]). Depending on the quality required, a request for allocation of bandwidth and buffers is made.

From the description below it may become clear that current QoS systems fail to provide services in terms of human perception (compare the information needed to specify requests in IIS) or to provide resource allocation adapted for perceived quality (compare the static resource allocation of RSVP).

RSVP, for instance, defines two important aspects; the protocol to send information over a network and the reservation services which make use of the protocol and are used for QoS management. Reservation messages are sent from node to node through the network. At each node, it is decided whether that node can grant the requested reservations. If not, that node does not propagate the request to the next node in the network, but sends a failure message back to the previous node. If the reservation can be realized, a (possibly modified) request is sent to the next node. This procedure is repeated until the final destination of the data is reached. This destination node will then return a success message to its previous node. All nodes along the path of the reservation message will wait for either a failure or success message from the node, to which they propagated the reservation message, and pass this message to their previous node. In this way, the original requester will receive a failure or success message and can act accordingly. However, in case of a failure message, it is hard to find out what went wrong at which node. ST-II uses a similar approach with respect to reservations as RSVP.

In general, each node in a QoS system goes through a number of steps for each reservation message that it receives. A general situation is described below; almost all QoS systems know some or all of the phases described here. In *selection*, a message is selected that will be serviced. Based on other reservations, made to this node earlier, the current reservation is considered in the *negotiation* phase. In the *admission control* the system determines whether enough resources are

available to make the reservation. During the *monitoring* phase, the QoS system makes sure that the reservations are maintained or, if not necessary anymore, canceled. During monitoring the delivered quality can also be watched; if the demanded quality is not met, the QoS system may change some settings in the *adaptation* phase. Note that the QoS system may come back on one of its previous actions and redo a previous step (as indicated in the figure).



Below, the relations between RSVP and ST-II on the one hand and these phases on the other hand are shown to give the reader an impression how these phases are used in a real QoS system:

	RSVP	ST-II
Selection	Select the next request to service.	Select the next message to service.
Negotiation	Check the current request with other requests made to this node earlier. Sometimes reservations for several different requests may be combined.	The route, along which the reservations have to be made, can be altered if the proposed route is not available to the current node. Also the flow specs can be altered if necessary.
Admission control	Determine whether enough resources are available to service the new request.	Determine whether enough resources are available to service the new request.
Monitoring	Check whether the current connections still have to be maintained (using reconfirmation messages).	Check whether the node along the path operates correctly with respect to the flow spec.
Adaptation	Cancel connections when no reconfirmation message is received anymore.	Reroute path of communication connection or reboot node along the path.

Internet Integrated Services (IIS) ([Shenker 95], [Shenker et al. 95a], [Shenker et al. 95b], [Shenker et al. 95c], [Shenker et al. 95d]) also describes methods to realize a certain quality with respect to delay. However, IIS puts the emphasis on buffering (using a bucket) instead of reservation of bandwidth. The three levels of quality supported are Guaranteed, Predictive and Best Effort. The different levels of priority each put different restrictions on the bounds of the end-to-end

delay. Below, some inequalities are given which characterize the different levels of priority in terms of the different parameters of the traffic model. These inequalities are presented here to give the reader a basic idea of how IIS deals with the different priorities and what kind of detailed knowledge is needed to make a request:

b	=	bucket size	M	=	Maximum packet size
r	=	bucket rate			
R	=	service Rate	C	=	maximum Control bytes
p	=	peak rate	D	=	maximum processing Delay

Guaranteed Priority Level:

$$p > R \rightarrow \text{delay} \leq \left( \frac{b-M}{R} \right) + \left( \frac{p-R}{p-r} \right) + \left( \frac{M+\sum C}{C} \right) + \sum D$$

$$p \leq R \rightarrow \text{delay} \leq \left( \frac{M+\sum C}{C} \right) + \sum D$$

Predictive Priority Level:

during the last minute the following must hold true:

$$\text{delay} \leq \sum_{i=0}^{59} \text{delay}_i + \frac{1}{60}$$

during the last hour the following must hold true:

$$\text{delay} \leq \sum_{i=0}^{359} \text{delay}_i + \frac{1}{360}$$

Best Effort Priority Level:

$$\text{delay} \leq \infty$$

As seen in IIS, problems with regard to the ability to deliver the required quality can be partially solved by allocating extra memory to (temporarily) buffer the data that has to be sent/is sent over the network connection. These kinds of solutions can help to smooth the network communication; data can be stored in a buffer when the actual communication rate is higher than required and the buffer can be emptied when the required rate cannot be met by the actual communication rate. However, network congestion in multimedia applications cannot always be solved by buffering data; once the application detects such a congestion, there is often no time to patch up the situation ([Amenyo et al. 93]). Therefore, it is important to look at systems which take actions in advance so network congestion occurs as little as possible (or not at all).

In general, it may be concluded that the current QoS systems for network communication deal with other aspects and with aspects at a different level of abstraction than the system that will be described in this thesis. This latter system aims at providing a number of abstraction levels at which requests can be issued; the translation from the viewer's high(er) level of abstraction to the level at which things need to be specified for the program is made by this system (and not by the viewer).



### 2.2 Application Areas for Multimedia QoS

The QoS systems described in the previous section are specialized in network communication. They require very detailed knowledge of the system and the network in order to specify QoS requests; this knowledge is not always present. In this section the focus is on QoS in multimedia environments. In a multimedia presentation network communication may be important to be able to present data which is distributed over several network nodes. However, when multimedia data has to be transported across a network, it may turn out to be very hard to make this feasible and cost effective (even with the current state of technology). Furthermore, network communication is, in general, not the part of the presentation that is directly visible to the viewer. If, in case such a QoS system for network communication is used, the viewer may want to adjust certain aspects of the quality of a presentation, the new quality requirements may have to be expressed in terms which are not familiar to this viewer and may be regarded as irrelevant to these requirements. When, for instance, the speed of a movie should be doubled (this may be desired when a movie is copied at twice the normal speed), the following adjustments may have to be made:

- the frame-rate should become 48 frames per second (instead of the usual 24 frames per second).
- the throughput of the network connection between the location where the movie is stored and the location where the copy is stored should be boosted up from 7.9 to 15.8 Megabyte per second (for a 256-color 720x480 pixel sized movie).
- additional buffers may have to be allocated to accomplish this increase in speed.

In this example, the discrepancy between the metrics used by an arbitrary application and the metrics used by the viewer becomes clear. What the viewer sees and may require from a presentation is largely influenced by what the viewer perceives. The viewer's perception has four important characteristics:

- the viewer has a continuous perception.

The average viewer has no notion that a movie is subdivided in discrete frames. More generally, a viewer may have no notion at all that digitized media are (necessarily) stored in discrete sections. Especially sound and movies are perceived as continuous media whereas the computer actually presents a series of discrete samples in such a high tempo that the illusion of continuity is created.
- the viewer is unaware of the underlying structures and formats.

The fact that a movie is stored in another place than where the presentation is shown is irrelevant to the viewer. That each frame consists of 720 by 480 pixels as well as that each pixel can have 256 different colors at the same time and thus that exactly one byte is needed to encode each pixel is not of any interest to the average viewer. Also the fact that each frame can be compressed using some kind of compression technique so the throughput of the connection network can be reduced is not germane to the viewer. The viewer is more concerned with the content of the media than with the way in which this media is stored.
- the viewer may not be able to perceive all of the information that is presented.

Neither the human eye nor the human ear are perfect. The eyes have limited capacities when it comes to perceiving, among other things, colors, motion and contrasts between colors. The human ear is unable to hear very high or very low frequencies. This means that a presentation may present information which is not perceived by the viewer. Something which is not perceived will probably not be requested from a presentation system and it will not be missed when left out. At a speed of 48 frames per second, the human eye will not notice it when, for

instance, every 12<sup>th</sup>, 24<sup>th</sup>, 36<sup>th</sup> and 48<sup>th</sup> frame is omitted to reduce the required throughput.

- the viewer may be unaware of the used resources.

The viewer may not know which actions have to be performed and which resources are necessary to accomplish the task so some information can be used in a multimedia presentation. In the example above, the need to allocate (additional) buffers so the required throughput can be realized, may not be directly known to the viewer.

In the next sections a number of areas, in which QoS can be applied to data which is typically used in multimedia presentations, is presented. In each section some perceptual metrics are listed which can be used to characterize the different media. The list of areas is not intended to be complete and is to some extent arbitrary; it merely gives an overview of the most commonly used types of data in multimedia presentations today. Several other areas can be identified; examples of these are tactile, taste and smell. At present, these senses are not commonly recognized as being interaction mechanisms in multimedia presentations. Nevertheless some work is being done in these areas. The best known research (in the area of tactile) is with the electronic glove; the user of the application can be made aware of the pressure needed to perform a certain task. Also in the area of smell, some experiments have been carried out. Well known are the experiments of Morton Heilig with his Sensorama in 1962.

### 2.2.1 Color Systems and Images

Color schemes and images are two subjects which are closely related. This also means that the perceptual metrics of images and colors are closely related. In practice only a few color schemes are widely used, but in theory a large collection of color schemes exists.

A color scheme can be based on a number of primary colors and by mixing these primary colors other colors can be created. In general, these types of color schemes can be divided into additive and subtractive color schemes. Additive color schemes have the property that the more primary colors are added, the whiter the resulting color becomes: RGB is an example of such an additive color scheme. The primary colors in this color scheme are Red, Green and Blue and combining these three primary colors results in the color white. A disadvantage of the RGB color scheme is the fact that no standards exist for the primary colors red, green and blue and thus the colors of an image specified using the RGB color scheme may look different on different monitors. Therefore, within the RGB color scheme family several substandards exist. These substandards all define different values for the following factors:

Y	:	the brightness of the color
Y <sub>n</sub>	:	the luminance of white
Y <sub>r</sub>	:	the ratio of red in the luminance of a color
Y <sub>g</sub>	:	the ratio of green in the luminance of a color
Y <sub>b</sub>	:	the ratio of blue in the luminance of a color

A RGB color scheme can be standardized via these factors according to the following equations:

$$\begin{aligned} Y &= Y_r \cdot R + Y_g \cdot G + Y_b \cdot B; \\ Y_n &= Y_r + Y_g + Y_b \end{aligned}$$

A commonly used scheme is  $(Y_r, Y_g, Y_b) = (0.2125, 0.7145, 0.0721)$  ([ITU-R BT.709 90]). This scheme however, does not define a veridical<sup>1</sup> color scheme. This has to do with the fact that each primary color has a different brightness. Green will appear to be the brightest color, red less bright and blue the darkest. Adding a certain amount of green to a color will cause the resulting color to increase more in brightness than when the same amount of blue was added. Because the human eye is more sensible to brightness than to color, the perceptual change in the resulting color is not veridical. A veridical RGB color scheme is defined by  $(0.299, 0.587, 0.114)$ .

Subtractive color schemes have the property that the combination of the primary colors results in the color black. Leaving out some of the primary colors will result in the other colors of the spectrum; CMYK is an subtractive color scheme where the primary colors are Cyan, Magenta and Yellow. The K stands for black and this color is added to the color scheme to create a pure black color (as the combination of cyan, magenta and yellow results in a very dark brown almost black color which is not really black). For less demanding applications, RGB can easily be converted to CMYK. More demanding applications may encounter a problem as the CMYK color scheme cannot define all the colors which can be defined by the RGB color scheme.

Besides the color schemes based on primary colors, other color schemes exist which are based on hue, saturation and intensity or brightness (HSI and HBS). The hue is a perceptual metric which is used to determine to perceived color of an area; it can be used to specify that an area seems to appear as red, yellow, green or blue (or a combination of two of them) ([CIE 17.4]). Saturation is the perceptual metric which represents the whiteness of a color. Saturation runs from neutral gray through pastel to white. Although both, the HSI and HBS color schemes, use perceptual metrics, these systems are not veridical. Because the brightness of a color is computed as  $(R+G+B)/3$ , the ratio of the different colors to the brightness is not considered. This makes these systems useless for image computations and are, therefore, hardly used anymore.

The last color schemes that are presented in this section are CIEL\*a\*b\* and CIEL\*u\*v\*. These systems are much more veridical than the other systems discussed here. However, these systems are too computationally intensive to be successfully used in image coding for printing. These systems exploit the fact that the human eye is more sensitive to brightness than to color. Colors are therefore stored using one brightness component ( $L^*$ ) and two color components ( $a^*$  and  $b^*$  respectively  $u^*$  and  $v^*$ ).

Each of the above mentioned color schemes can, more or less, be translated into each other. Important to note is that each color system has its advantages and disadvantages when it comes to defining the quality of a color. In [Teunissen et al. 96] a number of perceptual metrics with respect to images are summarized. It should be noted that most of these metrics have a strong relation with the colors used in the image:

- sharpness (of image, color transitions, contour rendering and rendering of details)
- brightness (of colors and image)
- contrast
- darkness
- naturalness (of colors and image rendering)
- smear (in the image and the geometric distortion of the image)
- flicker (of lines and large areas)

---

<sup>1</sup> A veridical mapping is one in which changes in a dimension are represented as equal perceptual changes ([Rogowitz et al. 92]).

## 2.2.2 Movies

Movies are typically a sequence of images displayed after each other with a certain speed. This means that, besides the perceptual metrics identified for still images, additional metrics can be identified which describe this sequencing. The most obvious ones are *speed* of the sequencing, the *direction* of the sequencing and the *smoothness* with which the different images are displayed after each other.

MPEG is a format for storing digital movies. In MPEG a frame can be encoded in four different ways. Which scheme is used depends on factors like required compression ratio and the need to randomly access the different frames of the movie. The former factor will look for redundancies in subsequent frames (these redundancies can be left out of the encoded movie) whereas the latter factor must avoid the creation of too much links to other frames (so a frame can be accessed without the need to retrieve information stored in other frames). The four encoding schemes are:

- **I-frames** (intra-coded frames):

intra-coded frames can be used for random access to the movie. The information stored for an I-frame is self-contained; i.e. coded without any reference to other images. I-frames in MPEG movies are encoded using the JPEG format (see also §2.3.1).

- **D-frames** (DC intra-coded frames):

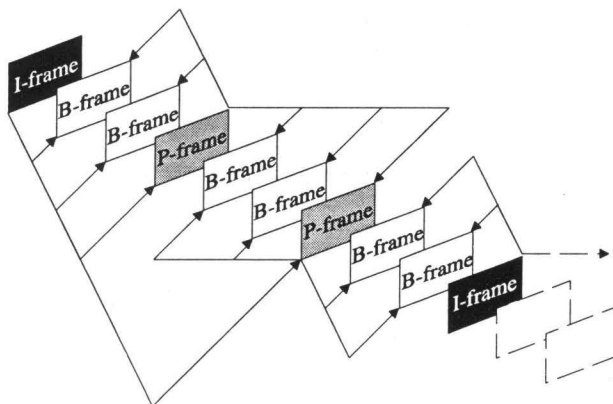
intra-coded frames where only the color-information of one pixel of an 8x8 bit image block is stored. These frames are also used for random access to a movie but have a higher compression rate than I-frames. Because a lot of information of the original frame is lost, they should only be used for fastforward or rewind mode.

- **P-frames** (predictively coded frames):

inter-coded frames which are used to exploit the possible redundancy between subsequential frames. A P-frame is described in relation to the previous I-frame as well as all P-frames in between. A P-frame has a higher compression ratio than an I-frame but can only be accessed after the previous I-frame and all the previous P-frames between the previous I-frame and the current P-frame are accessed. Thus P-frames are less randomly accessible but provide a better compression ratio.

- **B-frames** (bi-directionally predictively coded frames):

inter-coded frames which are also used to exploit the possible redundancy between subsequential frames. B-frames can be described in relation to previous and following I and/or P-frames. This allows for an even better compression ratio than with P-frames, but it also means that they can never be accessed in a random fashion.



For practical applications the following sequence has proved useful: IBBPBBPBB IBBPBBPBB ... In case this image is displayed with 25 frames/second, random access would still produce a movie of about 9 frames/second (IPP IPP ...) while a good compression ratio is also achieved. Note that it is possible that the sequence in which the frames are stored and have to be displayed can differ from the sequence in which the frames have to be decoded; in the case of an IBBPBBPBB sequence, first the I-frame has to be decoded, then the first P-frame and only then the first and second B-frame can be decoded. In the diagram above, this typical sequence of IBBPBBPBB frames is shown as well as the relations with respect to which frames are necessary to decode which other frames.

### 2.2.3 Sound

One aspect of sound is speech. The way in which a human being and an application will interpret a spoken sentence is, in general, different. Whereas the human being will probably make a distinction between the different words and possibly between the different sounds within a single word, the application will, more likely, look at the different sets of phonemes which are necessary to produce a certain sound or pronounce a certain word.

There are three different ways in which a multimedia presentation can deal with speech. These are:

- speech recognition

- an application will take either the recording of a human voice or a live sample and try to recognize certain commands in this sample. A very important perceptual quality is the *size of the vocabulary* that can be understood by the application. Small vocabularies are processed faster and (likely) with less errors, but force the speaker to shape the sentences in a particular way, convenient to the application ([Yankelovich 94], [Yankelovich et al. 94], [Yankelovich et al. 95]). Large vocabularies may lead in turn to more recognition errors:

- rejection error:

- the recognizer did not understand what was said. This typically happens when the speaker uses a word which is not present in the (small) vocabulary of the recognizer.

- substitution error:

- the recognizer interpreted a spoken sentence wrong and recognized a word from its vocabulary which was not spoken by the speaker but which sounded similar. This may happen in situation where different words, sounding similar, are contained in a (large) vocabulary.

- insertion error:

- the recognizer mistook noise from the background as a word spoken by the speaker.

Other perceptual characteristics of speech recognition are *pitch*, *loudness*, *nasality*, *timing and rhythm* and *breath control* ([Shaw et al. 96]).

- speech synthesis

- an application tries to produce sound that resembles the human voice. Typical examples in this area are text-to-speech applications which try to convert an ordinary text string into sound. Important perceptual qualities are *pacing*, *prosody*, *response length* and *response time*. They determine to a great extent whether a listener values the speech as natural and pleasant to listen to ([Yankelovich 94], [Yankelovich et al. 94], [Yankelovich et al. 95]).

- playback of pre-recorded speech

an ordinary sound file is used to store human speech. This sound file is played back using an ordinary sound file player.

Much research is still going on in the fields of speech recognition and speech synthesis. Nevertheless, much progress has already been made.

Another aspect of sound is music. Just as speech, music too is a special case of sound. Formats like Mod and MIDI use metrics like *key*, *pitch*, *duration*, *volume* and *instrument* to describe the contents of a music file. Instead of describing the actual sounds waves, these kind of formats describe how the sound can be generated. The advantage of such an approach is that the description of the sound is device independent and expressed almost totally in terms of perceivable metrics. The disadvantage of this approach is that the generated sound can differ dramatically when generated at two different places (using different hardware).

Finally, there is sound which has to specific structure. This type of sound can only be stored by describing the different sound waves that should be generated to reproduce the original sound as good as possible.

### 2.3 Compression Techniques

In this section three examples are given of how multimedia data, as described in the previous sections, can be compressed. Again, the choice of these examples is rather arbitrary. The purpose of these examples is merely to give the reader an impression of how different kinds of media can be compressed. The examples that are chosen in this section will discuss the compression of images, of speech and of sound in general (speech compression can achieve a better compression ratio than general audio compression as the former compression technique can use the fact that human speech is compressed and take advantage of this knowledge). In §2.3.2 the relevance of these kinds of compression techniques with respect to resource management is discussed and in §2.3.3 its relevance to perceived quality.

#### 2.3.1 Examples of Compression Techniques for Multimedia Data

In general, images can be compressed by three different levels of compression. The compression techniques which are the least sophisticated are the ones which combine a block of pixels and store only the value of one of these pixels (while the values of the other pixels are thrown away) or the average value of these pixels. Images can be compressed by an arbitrary factor (when the block contains  $b$  pixels, the compression ratio is  $b$ ) but the quality of the resulting image after decompression (where all pixels in one block are given the same value) can be very poor.

A more sophisticated technique is the one which makes use of the brightness of colors. An example of this technique is the  $Y C_r C_b$  technique. The idea behind this technique (where  $Y$  stands for luminance,  $C_r$  for the amount of red (Chroma Red) and  $C_b$  for the amount of blue in the color of the pixels<sup>2</sup>) is that the brightness is a more important parameter to describe an image than the chroma. This technique can vary the compression rate by varying the precision of the  $Y$ , the  $C_r$  and the  $C_b$  component. Commonly used schemes for  $Y:C_r:C_b$  are 4:4:4, 4:2:2 and 4:1:1. In these cases, the image is divided into blocks again; each block consists of 2 by 2 pixels. In the 4:4:4 compression

---

<sup>2</sup> The component green in a color is not used in this color system (as opposed to the components red and blue) because the information which would be stored for the green component of a color has a large overlap with the information for the luminance of a color and would, therefore, not add much extra information.

scheme, no compression is applied and for each pixel the luminance, the chroma red and the chroma blue are stored. In the 4:1:1 compression scheme, the luminance of each pixel in the block is stored as well as the average of the chroma red of all the pixels in the block and the average of the chroma blue of those pixels (hence the ratio 4:1:1 which indicates that a block of pixels is described by 4 luminance values, one chroma red value and one chroma blue value). In the case of the 4:2:2 compression ratio, a block is described by 4 luminance values, 2 chroma red values and 2 chroma blue values.

An example of one of the most sophisticated compression techniques is JPEG ([Leger et al. 91], [Mitchell et al. 91], [Steinmetz 94b], [Wallace 91], [JPEG 93]). In JPEG four different modes can be distinguished of which in this section only the lossy sequential DCT based mode (also called baseline mode) will be discussed. An image is, in the views of the JPEG standard, divided into a number of planes (for RGB values, the image would contain three planes; one for red, one for green and one for blue). These different planes are considered to be separate entities. Each plane is then divided into blocks (in the lossy compression mode this block is 8 by 8 pixels) and the blocks of the different planes are ordered in such a way that, during decompression, the different planes do not have to be decoded one by one, but can be decompressed in an interleaved manner so the complete image can be build up gradually. The compression of each individual block is done using a Discrete Cosine Transformation (DCT). The DCT maps the colors of the individual pixels to a two-dimensional frequency domain. The DCT of the upper-left pixel of each block corresponds to the lowest frequency in both dimensions and is called the DC coefficient. This DC coefficient determines the fundamental color of the 64 pixels in the block. The DCT of the other pixels (called AC coefficients) will, in average complex images, be close to zero. The next step in the compression algorithm is a lossy transform; the DC and AC coefficients are quantized. Using predefined tables, the precision of the 64 coefficients is reduced; the lower the precision the better the final compression ration and the worse the image quality after decompression. The last step in the process is an entropy encoding of the quantized DC coefficients (using Huffman or arithmetic coding). Using JPEG compression, images encoded using the DCT can be compressed such that an average pixel (8 bits) can decoded using from 0.25 up to 2 bits (the quality in these cases ranges from moderate to not distinguishable from the original).

Finally a small note on the compression of movies. For moving images, the same remarks hold true as those made for still images; the human eye is limited in what it can perceive. As moving images are simply a sequence of still images, the same techniques can be used to compress the data that describes a moving image as the ones that were used to compress the data of a still image. In addition, some other techniques may be used to compress the sequence of moving images even more; depending on the speed in which the different frames in a sequence are shown, an occasional frame may be left out during the presentation. Furthermore, as it is likely that sequential frames have almost the same contents, the contents of a frame may be described in terms of the contents of the previous or next frame. For this purpose also motion vector analysis may be performed on a sequence of frames to detect whether and how a certain area in the different frames moves from one location to another.

For the efficient storage of pre-recorded speech much research is done with respect to the compression of the speech. In [Degener 94], a compression algorithm (GSM 06.10 RPE-LTP) is described which uses the fact that the sound files it works with contain pre-recorded human speech. This algorithm works with frames of 160 samples of human speech (sampled at 8 kHz). Each frame covers about 20 ms of speech. This time frame is about as long as one glottal period for persons with a very low voice at the bottom their range or about ten glottal periods for persons with a very



high voice at the top of their range<sup>3</sup>. Within each frame, it is unlikely that the speech wave will change very much. The Long-Term Predictive (LTP) filter tries to predict the speech waves using the past waves. The results of this LTP analysis are subtracted from the original speech waves. The resulting residual signal is, hopefully, either weak (if the frame contained a voiced phoneme, the LTP will have predicted most of the glottal wave) or random (if the frame contains a voiceless phoneme, the residual signal is noisy and does not need to be transmitted precisely) and consequently cheaper to encode. The Residual Pulse Excitation (RPE) is used to compress the residual signal. During this last compression, information may be lost (RPE is a lossy compression technique). However, RPE will preserve the perceptual characteristics like pitch and loudness. This technique can achieve a total compression rate of 10:1.

General sound files can be used to store all kinds of audio. The difference with speech or music files is that this category is more general; no assumptions can be made with respect to the contents of the audio. Because of their general character, these files can be compressed less easily using the special features of the contents of the file. However, general sound can be compressed. Important in this respect is the perceptual quality characteristics used in sound ([Kayargadde et al. 96a]). In general, the following aspects can be distinguished:

- noise
- volume
- frequency
- range of frequencies

MPEG-I audio compression ([MPEG 94]) is based on the fact that the human ear can hear frequencies from about 20Hz to 20kHz and that some frequencies are masked by other frequencies. Sound which has a frequency which is near the frequency of another sound but has a smaller volume will be masked by that other sound. The greater the difference between the two frequencies, the smaller the volume of the first sound has to be so it can be masked by the second sound. Thus, the masking effect implies that noise is masked around a strong sound and thus less accuracy is needed to code the signal. An audio sound can thus be compressed by leaving out all frequencies which are not in the range from 20Hz to 20kHz and by leaving out all frequencies which are masked by other frequencies. In this way, MPEG-I audio compression can reach a compression ration of 6:1 or 7:1.

### 2.3.2 Relevance of Compression Techniques for Resource Management

The relevance of compression techniques for resource management may be obvious; compressed data requires less space and thus less resources when this data, for instance, is stored or transported over a network connection. In the first case less disk space or memory is occupied by the compressed data and in the latter case either less time is needed to send the compressed data over the network connection using the same bandwidth or less bandwidth is needed using the same amount of time.

In this way it seems that compression is always beneficial for resource management; compression always leads to the usage of less resources. However, this is not totally true. The resources needed to actually compress and decompress data should also be considered when the overall benefits of compression for resource management are discussed. And this means that

---

<sup>3</sup> A glottal period is the time that is needed to push air from the opening of the lungs to the glottis (the gap between the two vocal cords) and close it again. This process can be repeated between 50 and 500 times per second, depending on the physical construction of the larynx and the force with which the vocal cords are pulled.



especially claims on the resource time increase; compression and decompression may be very time-consuming. Some compression algorithms try to reduce the time needed to decompress the data, but in general this means that the compression takes even more time. Furthermore, (de)compression algorithms may also need other resources like memory. Due to these kinds of restrictions, the use of compression techniques may not always be worthwhile; in the end it may take an equal amount of resources to compress some data, send it over a network connection and decompress it again as what it takes to send the uncompressed data over.

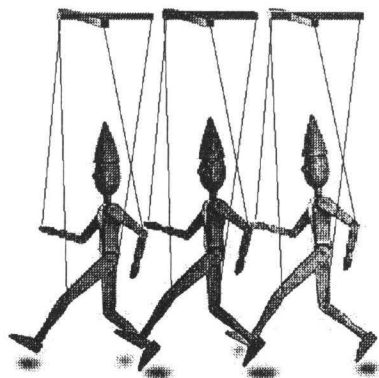
The advantage of compression techniques therefore lies in the way in which huge claims for one resource can now be divided into a number of (smaller) claims over different resources. Sending uncompressed data over a network connection may take much bandwidth, but sending over compressed data may reduce the claims on the bandwidth while new claims for memory may be introduced. Compression also makes it possible to exploit trade-offs between claims on different resources; much bandwidth will probably lead to a quicker transport of data over a network connection whereas less bandwidth leads to longer times to complete the communication. If not much bandwidth is available and there is no restriction on the time, a high compression ratio might be a solution, even when it takes the compression algorithm a long time to compute this highly compressed data. The lower the compression ratio, the quicker a compression algorithm can compress the data such that the desired ratio is met. In this case a trade-off between bandwidth and time exists.

These kinds of trade-offs become even more important when a number of different applications compete for the same resources. In that case, the huge claims of one application on a specific resource may lead to a situation where other applications cannot perform their tasks because the part of the resource which is not taken by the huge claim may not be sufficient for the other applications. However, if the claims of the former application can be reduced, the other applications may be able to perform their tasks as well. In such a situation, it may turn out that compressing the data of the first application (using some kind of trade-off) does not disturb its overall performance (too much), while the other applications can also perform their tasks within reasonable time. The main problem here is, of course, to find the right trade-offs such that all applications can perform their tasks within the time limits set for them.

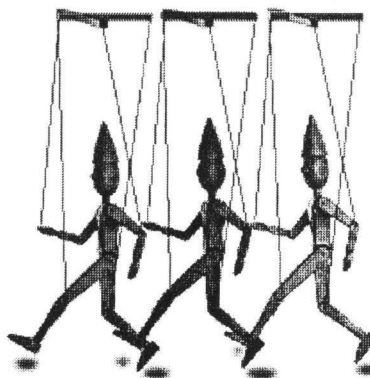
### 2.3.3 Relevance of Perceived Quality for Compression Techniques

The relevance of perceived quality to compression techniques has already been mentioned before in this section. The reference that was made was that the human eye was more sensitive to brightness than to color. Using this information, images could be compressed without much loss of perceived quality. The sensitivity of the human eye to brightness is not the only aspect that can be exploited when it comes to perceived quality; due to the fact that human beings are not capable of perceiving all the information that is presented to them via, for instance, an image or a sound sample, a lot of redundant and superfluous information can be removed from the various multimedia data. In [Kayargadde et al. 94], [Kayargadde et al. 96b] and [Kayargadde et al. 96c] it is shown how still images can be easily compressed without too much loss of perceived quality. In [Par et al. 94] it is shown how a three-channel (left, right and center) sound can be transported over only 2½ channels or how a three-channel sound can be played using a two-channel sound system (and where the center signal is not just thrown away but mixed with the signal for the left and the right channel). The resulting two-channel sounds have a perceived quality which is much higher than when the center signal would just be omitted. In general, compression techniques try to minimize the space needed to store the data itself while preserving as much control data as possible which can be used to restore the original data from the compressed data (minimalizing the data itself and maximalizing the control data).

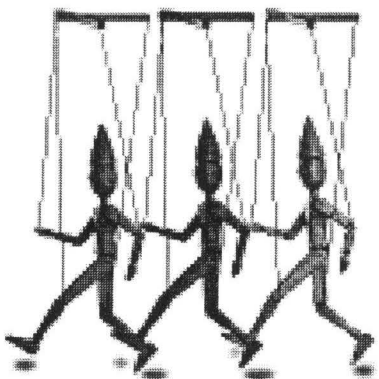
To show that the perceived quality of images does not degraded immediately when it is compressed, an simple example is shown here. In this example the information for all the pixels in certain block is replaced by the value of the upper-left pixel in that block. This is not an advanced technique to compress images, but it gives an impression what to expect when simple compression algorithms are used. Image (a) is the original image. Image (b) has a block size of 2 by 2 pixels, image (c) 4 by 4 and image (d) 8 by 8. This means that the compression ratios of the images are respectively 1:1, 4:1, 16:1 and 64:1. Image (b) still provides a good perceptual quality. Although the original image can clearly be recognized in image (c), the deformations due to the compression of the image are very well visible. The quality in image (d) is clearly very poor. The contours of the original image can still be recognized but all the details have clearly gone.



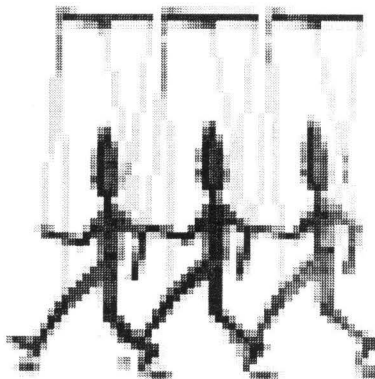
(a)



(b)



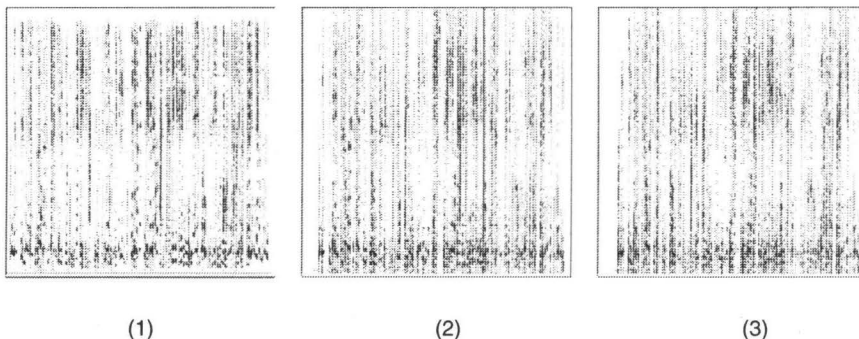
(c)



(d)

As an other example, the results of the GSM 06.10 RPE-LTP compression technique are shown here. The three diagrams show the frequency graphs of the three sound files which contain some spoken text. The first diagram shows the original recording of a dialog; the second diagram shows

the same dialog after it has been compressed and decompressed using the GSM algorithm. There are clearly some differences between the two diagrams, but the overall pattern is very similar. The third diagram shows the same dialog after it is compressed and decompressed five times in a row by the same GSM compression algorithm. In this last diagram, the patterns have still a strong resemblance with the first diagram, but also some differences can be noticed; the dark areas in the third diagram are concentrated around the places where the dark areas were present in the first diagram and the spots in the first diagram which were already light have become even lighter in the third diagram. The effect of the GSM compression algorithm clearly preserves the frequencies which occur often in the sound and remove the frequencies which are not used that often. When the sound of the file, belonging to the third diagram, is played back, the original sound is clearly recognized. However, also a small deformation can be heard which manifests itself as a ticking sound in the background. This sound was not in the original sound file and could also not be heard in the sound file belonging to the second diagram. This shows that some care has to be taken when data is manipulated; manipulation of manipulated data may, in the end, lead to information which does not meet the required quality level and create deformations in the data which can be easily perceived by the viewer of a multimedia system.



Important to note is that compression may leave the perceived quality intact while the claims on the different resources may be reduced. In order to do so, the quality of non-perceptual metrics may be sacrificed to keep the quality of perceptual metrics as high as possible. To be able to do so, it must be clear which perceptual metrics influence other perceptual and non-perceptual metrics. In [Rogowitz et al. 92], [Roufs 92], [Roufs et al. 94] and [Nijenhuis et al. 93], the relation between a number of perceptual metrics and non-perceptual metrics is presented and discussed. This discussion deals primarily with metrics for the quality of still images. Similar techniques can be applied to other media to achieve the same effect.

From the descriptions above it may be clear that the viewer of a multimedia presentation may use different terms to describe the quality of multimedia data. Furthermore, it may also be clear that compression can be useful to do good resource management and that higher compression ratios can be achieved when the compression algorithm only has to care about perceived quality (and not the absolute quality). Different compression algorithms may introduce claims on the various resources which are intrinsic to that algorithm. The MPEG compression technique cannot decode arbitrary frames; for instance, the corresponding of P and I frames have to be decompressed before a B frame can be decompressed. These kinds of relations should be considered carefully when a compression technique is chosen.

### 2.4 A Perceptual Quality Constraint System: The Quality Factory

The Quality Factory, as presented in this section, is a general framework which can be used for many different situations. In chapter 5, a specific example (a movie in FLI-format is converted and (de)compressed to meet the specific QoS requirements) is discussed in detail. This section is intended to provide a more general view upon the Quality Factory.

As an introduction to the Quality Factory, the reader is given a small impression of the multitude of formats, standards and techniques which exist in the world of multimedia. It may be clear that is very hard to make a system which is able to support all the different variations of the different media formats. The QoS systems, as discussed in §2.1, are by themselves not powerful enough to really support a multimedia system in presenting multimedia presentations whose contents are distributed over several computer systems. This is due to the fact that they can only give (limited) guarantees with respect to the amount of data they can send over a network connection. However, if the amount of data, that needs to be transferred between two different nodes on the network, exceeds the capacity of the connection, these QoS systems fail to provide a working solution. They also do not provide facilities which can convert formats, in which some multimedia data is stored, into another format which is more appropriate for the required QoS (for instance with respect to size). So the usability of the network communication QoS systems for multimedia systems is very important (provide the multimedia system with a steady stream of data), but limited; they can only work with a given data stream which contents has no meaning to them. Because the data stream is opaque to these systems, they cannot perform transformations on these data streams so, for instance, video data is compressed in order to be able to provide the required frame-rate. Also their terminology is not adaptable to the terminology of the user. This may lead to situations where the communication between user and application can become difficult.

In our proposed system, the QoS systems for network communication can form part of the foundation on which our system is based; i.e. existing systems can be integrated in the framework of the Quality Factory. At the lower level, these network communication QoS systems may provide services which transport multimedia data from one node in the network to another node. However, the necessary manipulation of multimedia data is performed by the *Quality Factory*. An important aspect for this manipulation is compression of multimedia data, as compression of information may lead to lower demands with respect to the (limited) available resources. The availability of resources is essential; without resources no presentation can be given, not even one with bad quality. Furthermore, compression may also lead to a higher rate of data exchange and thus to a faster playback of the multimedia data.

In the previous sections, it was already shown that the human perception is not perfect; information in the multimedia data may be left out (because it is redundant) or can be compressed (because the contents of (part of) the data can be predicted) such that it has hardly any effect on the perceived quality. Most of the described techniques which take advantage of this phenomenon are well known and several implementations exist. The Quality Factory does not want to describe new techniques or present new implementations for these kinds of techniques. The problem in the current situation is that each implementation makes use of only a few of the specific aspects of human perception and does so in its own way whereas the (real) quality lies in the combination of these different aspects.

In the Quality Factory, which is described in this thesis, Quality-of-Service management for multimedia environments is supported. What sets the Quality Factory apart is the fact that viewers can issue quality requests using their own notions and that the Quality Factory will translate these to requests in terms of more technical metrics which can be used by the different (presentation) programs. During this translation process, information has to be manipulated. For this purpose special arrangements have to be made in order to propagate information from the viewer to the

programs which do the actual presentation of the data. Because these arrangements are provided by the Quality Factory anyway, they may also be used for other purposes as well; the Quality Factory supports, besides the translation facilities, means to convert the storage format of data into another format when necessary such that programs can be combined, means to manage resource allocation of the resources which are needed during the presentation and means to define different scenarios such that the behavior of the Quality Factory can be adapted if needed. All these facilities together make it possible to let viewers, unfamiliar with any technical detail, define quality requirements. These requirements can be made at different levels. Requests made at a high abstraction level require not much knowledge from the viewer, but much intervention from the Quality Factory; the viewer has little control over what happens inside the Quality Factory itself (and probably does not want to) while the actions taken by the Quality Factory will be some standard scenario for the given request. Requests at a low but detailed abstraction level require more knowledge of the underlying system, but allow for more control. The Quality Factory allows arbitrary levels of abstraction and allow novice as well as experienced viewers to use the Quality Factory.

In the next paragraphs the idea behind and the potential of the Quality Factory is demonstrated. In the example, several of the facilities, mentioned above, will be used to create a general image viewing program.

Although JPEG is a standard, current JPEG implementations can differ dramatically from each other. JPEG defines several subformats and different implementations do or do not support each and every one of these subformats. Thus images compressed by arbitrary JPEG-encoding implementations do not necessarily have to be readable by other arbitrary JPEG-decoding implementations. In the worst-case situation, twenty-two (= the number of subformats) different JPEG-readers have to be available to be sure that an arbitrary JPEG image can be read:

	mode	sub-mode
JPEG	lossy sequential DCT based mode	-
	expanded lossy DCT based mode	sequential-8-Huffman
		sequential-8-arithmetic
		sequential-12-Huffman
		sequential-12-arithmetic
		progressive-successive-8-Huffman
		progressive-successive-8-arithmetic
		progressive-successive-12-Huffman
		progressive-successive-12-arithmetic
		progressive-spectral-8-Huffman
		progressive-spectral-8-arithmetic
		progressive-spectral-12-Huffman
		progressive-spectral-12-arithmetic
	lossless	no prediction
		X=A
		X=B
		X=C
		X=A+B-C
		X=A+(B-C)/2
		X=B+(A-C)/2
		X=(A+B)/2
	hierarchical	-

Atechnical viewers, given a JPEG-compressed image, do not want to know about these formats, subformats and the different programs which each support a different set of subformats when they only want to view their image. In current systems, this situation is not dealt with in a proper way; facilities to support several programs to read different subformats have to be either hard-coded in the application or no support is given for this at all. This means that the atechanical viewer is forced to find out the format in which the image stored and to locate a program which can be used to view the image.

The Quality Factory creates an environment where it is possible to have different JPEG readers that may each have different qualifications. These different JPEG readers can be connected to a general JPEG reader whose only task is to establish which available JPEG reader(s) can decode the current JPEG image (using the information of the (sub)format of the JPEG image). At this point the atechanical viewer does not have to find out which format is used to store the image; he can present the image to the general JPEG reader and this reader, together with the Quality Factory, will ensure that the correct image is displayed. The viewer issues a very abstract request: 'show (arbitrary) image'. The Quality Factory and the general JPEG reader will fill up the gap in the information that is not provided by the viewer: find out which (sub)format is used to store the image and select the proper JPEG reader. If more than one JPEG reader can be used, the Quality Factory has to choose one of them. This choice can be based on several things (the resource load can be one of these). However, a more experienced viewer may issue a less abstract request; based on the experience a specific JPEG reader may be chosen: 'show expanded lossy DCT sequential-12-Huffman image'.

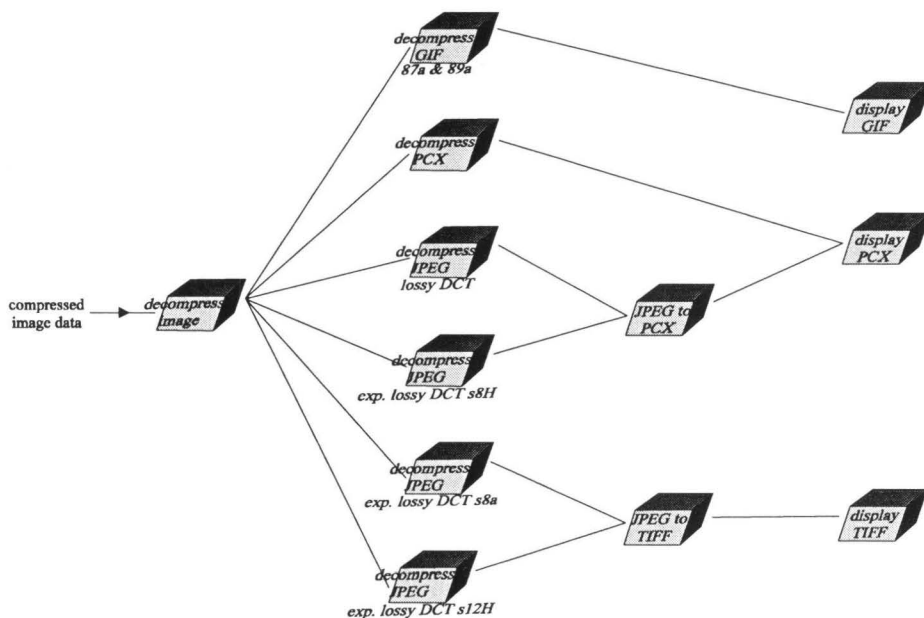
Because the Quality Factory puts no restrictions on the actual JPEG readers, their arguments (order and/or type) or the way in which input and/or output has to be handled, off-the-shelf modules can be used. By connecting the different modules in a proper way, information can flow from its source location to its destination and from one module to another. Each module that is visited on the way can modify the information as needed. This means that the format in which the information is stored can be changed, the information can be compressed (while, if possible, maintaining the perceived quality) or the information itself can be altered. The Quality Factory assists in making the links between the different modules and in assuring that the information is propagated through all the different links and to the various modules properly (taking into account possible dependencies between the different modules). On top of that, modules can be added dynamically to the system; not the Quality Factory itself has to be updated but only new links have to be defined. Thus, in the case of the Quality Factory, the atechanical viewer only has to know that there is a program which can be used to view images; this program will display the images regardless of their type or subformat. The diagram below shows the situation where an image in either GIF, PCX or (some subformats of the) JPEG format can be given to a general image-viewing program. This program will determine the format of the image and decompress and display the image accordingly. From the viewpoint of the viewers, this general system provides a much better perceived quality than the current systems; the viewers do not have to worry about formats and other things they cannot perceive, but they can concentrate on what is or should be presented. In the figure below a possible configuration is shown which can be described using the framework of the Quality Factory. The current connections between the different boxes show one of the many possible configurations and are in no way prescribed (or restricted) by the Quality Factory<sup>4</sup>.

During its task to propagate the information through the links between the modules, the Quality Factory can be instructed to discard certain links based upon the contents of the information which is propagated. This can be illustrated using the diagram above; a GIF-encoded image will only have

---

<sup>4</sup> In this figure 'exp. lossy' stands for expanded lossy, 's8H' for sequential-8-Huffman, 's8a' for sequential-8-arithmetic and 's12H' for sequential-12-Huffman.

to be propagated to the decompressor for GIF images and not to the decompressor for PCX images. Therefore only the links between 'decompress image'-'decompress GIF 87a & 89a' and between 'decompress GIF 87a & 89a'-'display GIF' have to be selected while the other links can be disregarded when a GIF-encoded image is received by the general 'decompress image' program. Furthermore, information (i.e. different media streams) can be divided into parts which are all propagated along different links. These are powerful features, which enable the Quality Factory to build an environment which is able to create a network of general image viewers as described above.



To be a general QoS system which supports perceived quality metrics, the general framework of the Quality Factory has to meet a number of requirements:

- The Quality Factory must be able to interpret commands issued by a viewer in terms of perceptual quality (see §2.2).
- The Quality Factory must be able to translate perceptual quality commands to commands which are more technical and can be used to control the behavior of software modules which do the actual manipulation of the data.
- The Quality Factory must be able to incorporate and use third-party modules when a description is given in which is specified what information under which circumstances is needed. This description may be provided by the third-party module itself.
- The Quality Factory must be able to connect different third-party modules to accomplish a task which cannot be accomplished by one of the other third-party modules alone and to direct the information flow between those modules based on the description that was mentioned in the previous item.



- The Quality Factory must support a number of built-in operations to allow for expressions which make it possible to perform some simple manipulations on the dataflow when two modules are connected. In this way, the manipulation of the dataflow is not only restricted to the modules, but simple manipulations can then also be done by the Quality Factory itself which reduces the number of modules needed. Examples of built-in operations can range from simple arithmetic operations like addition or multiplication to more complex operations which can extract a particular media stream from the presented data.
- The Quality Factory must support different contexts such that it is possible to define various situations in which (the manipulation of) the dataflow between modules can be specified differently for each of these cases. This ability allows a programmer to define what actions to take when, for instance, only a slow speed data-connection is available (do some serious compression) and when a high speed data-connection is available (do no compression).
- The Quality Factory must support the possibility to add or remove third-party modules from the set of supported modules dynamically; over time new modules may appear and can be used by the Quality Factory to perform certain tasks, while other modules may disappear.
- The Quality Factory has to make sure that the time it needs to respond to commands from either viewer or module is in agreement with the task that is performed. However, different tasks may need different response times; synchronizing audio and video is much more time critical than just the display of a single image.
- The Quality Factory must support facilities to allocate (scarce) resources over the different modules which are in the act of performing some kind of operation (modules which are idle should not take up resources). Conflicts between resource allocations should be handled gracefully.
- The Quality Factory must operate transparently to the different modules such that they do not have to be prepared before they can be used in the Quality Factory.

In the following chapters, the Quality Factory as well as the foundations it is based on will be discussed. One important foundation of the Quality Factory is constraint management. One of the propositions of this thesis is that constraints are very well suited for the specification of the relations between the different (third-party) modules. The next two chapters will discuss constraint management in general and the MADE constraint system in particular. In chapter 5, the implementation of the Quality Factory based on constraint technology is discussed as well as how well the above mentioned requirements are met.





# CHAPTER 3

## Constraint Satisfaction and Constraint Systems



This chapter will provide a short overview of the different aspects which characterize the various constraint systems. It tries to provide the reader with a firm basis with which the design of the MADE constraint system (described in chapter 4) and the working of the Quality Factory (described in chapter 5) can be understood more easily. First, the reader will be presented with the terminology used in the field of constraint satisfaction (§3.1). After that, the different variations in the supported constraint networks are discussed (§3.2) as well as the different propagation techniques (§3.3) and satisfaction techniques (§3.4). These aspects of constraint systems are important as they very much determine what kind of constraints can be specified in the constraint system and thus what kind of problems can be solved by that same constraint system. As a consequence of that, these aspects have influenced the design of the MADE constraint system to a large extent. In §3.5, some of the most important constraint systems will be discussed. It should be mentioned here that Constraint Logic Programming will not be considered in this thesis. The reader is referred to [Dincbas et al. 88], [Graf et al. 89], [Hentenryck 89], [Hentenryck 91] and [Hentenryck et al. 91] for more details on CLP.

### 3.1 Terminology

Within the area of constraint systems, a lot of different terminology is used. This may be due to the fact that constraint systems are used in a large number of areas. A selection of the most important areas in which constraints are used is presented in the following list:

- user interface control ([Borning et al. 86], [Maloney et al. 89])
- geometric layout ([Nelson 85], [Rankin 91], [Velkamp et al. 92])
- algebraic equations and inequalities ([Gosling 83], [Leler 88], [Wilk 91])
- animation ([Borning et al. 86])
- synchronization ([Bordegoni 92], [Frølund 92], [Hardman et al. 92])
- synthesis and analysis of electronic networks ([Sussman et al. 80])
- planning and scheduling ([Hentenryck 91], [Hentenryck et al. 91])

For each of these areas special dedicated constraint systems exist which focus on satisfying constraint problems in the small area for which they were designed. However, there are also more general constraint systems, which can be used in several areas. Throughout this thesis, the terminology, presented in this section, will be adopted. With respect to a constraint relation the following terms are used:

- properties  
The properties are the entities on which a constraint relation is defined. Properties are also called *nodes*, *variables* or *constraint-variables*. However, these last names may cause confusion in relation with variables of objects in object-oriented languages or variables of predicates in logical programming languages.
- a (dependency) relation  
The dependency relation describes how the different properties are related to each other. A relation can be used to check whether the values of a number of properties satisfy a constraint or it can be used to compute the value of a number of properties, given the values of a set of other properties. Relations are also called *arcs*.
- the constraint itself  
A constraint has a name which is used to identify the constraint. A number of properties and a dependency relation are associated with the constraint. This means that a reference to the name of a constraint also refers to the dependency relation and the properties it constrains.

The various constraints are maintained by a so-called *constraint system*. This constraint system maintains (at least) one *constraint network* which contains the different properties and dependency

relations. The task of the constraint system is to make sure that the values of the different properties are consistent with the dependencies specified in the dependency relations (see also [Freuder 78], [Mackworth 77] and [Zhang et al. 91]). If there are inconsistencies in the network, the constraint system is responsible for assigning different values to the properties so that the inconsistencies are removed. This process is called *constraint satisfaction* ([Tsang 93]). Constraint satisfaction may use the same constraint network as the one in which the different properties and dependency relations are contained or it can use a second (and a third, ...) constraint network where the intermediate results of the satisfaction process are stored. When a set of dependency relations constrain a property in the constraint network in such a way that no value can be found for that specific property that makes it consistent with all the dependency relations, the constraint network is said to be *over-constrained*. The opposite is also possible; several values can be found for a property so it is consistent with all the dependency relations. In that case, the constraint network is said to be *under-constrained*. The problem of having several candidate values for one property is often solved by using the *principle of least astonishment*; that candidate value is selected which lies closest to the original value of the property.

During the process of constraint satisfaction, three important tasks can be distinguished. These three tasks have a close relationship with the structure of the constraint network:

- triggering

The process of notifying the constraint system that a new satisfaction process is needed due to the fact that values of some of the properties in the constraint network are inconsistent with some of the dependency relations. The structure of the constraint network may limit the connections which are allowed between properties and dependency relations.

- information propagation

The process of advancing information through the constraint network, from one property to another. The structure of the constraint network thus determines how information can be propagated, to which properties and along which constraints.

- constraint satisfaction

The process of making the values of the different properties in the constraint network consistent with the dependency relations. The structure of the constraint network determines to a great extent in which order the different constraints have to be satisfied.

### 3.2 Constraint Networks

Different constraint systems may support different kinds of structures for their constraint networks. The expressive power of the constraint system is strongly determined by the structure of the constraint network which it supports. In this section, the different aspects of a constraint network are presented. For each of these aspects their effect on three processes, triggering, propagation and satisfaction, are discussed.

#### 3.2.1 Direction of the Dependencies

The relation between the various properties described in a dependency relation can be either *directional* or *adirectional*. This distinction between the direction of a dependency relation is also known as the difference between *uni-directional* and *multi-directional* dependency relations or as the difference between *one-way* and *multi-way* dependency relations.

Directional dependency relations explicitly specify the values of which properties are computed from the values of other properties. This means that directional dependencies clearly reduce the way

in which information may be propagated through the constraint network; information flows from the properties, which values are used to compute the values of other properties, to the properties for which its value is computed by using the values of the former properties. Directional dependencies also (partially) determine the way in which the different constraints in the constraint network are satisfied; the order in which the different constraints are satisfied should follow the order in which the information flows through the constraint network.

Adirectional dependency relations do not specify a direction. When an adirectional dependency is specified, two approaches are possible: the description of the dependency is analyzed and a set of directional dependency relations is created which is equivalent to the original adirectional description or the direction of the propagation of information is determined at the time the propagation process has reached the properties used in the dependency relation.

The difference between the different approaches (directional, adirectional converted to directional, true adirectional) can be shown by the different constraint networks that are created for each situation. Given the dependency relation:

$$*a = b + c*$$

where  $a$ ,  $b$  and  $c$  are the different properties. In the case of a directional dependency relation, a constraint network is created as shown in figure 3.1a.

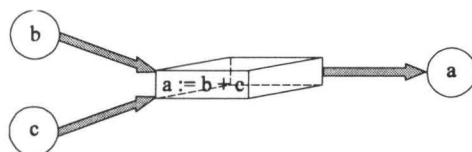


Figure 3.1a: A directional dependency.

When the values of properties  $b$  and  $c$  are known, the value for property  $a$  can be computed. In the case where the adirectional dependency relation is converted into a set of directional dependency relations, three dependency relations are created:

$$\begin{aligned} *a &:= b + c* \\ *b &:= a - c* \\ *c &:= a - b* \end{aligned}$$

The constraint network created in that situation is presented in figure 3.1b.

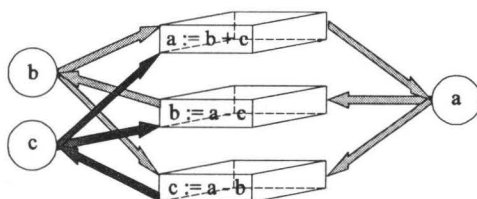


Figure 3.1b: An adirectional dependency translated to a set of directional dependencies.

In case the direction of the information propagation is adirectional, then actual direction of information flow (i.e.  $a := b + c$ ,  $b := a - c$  or  $c := a - b$ ) is determined at run-time and the constraint network as shown in figure 3.1c is created<sup>1</sup>.

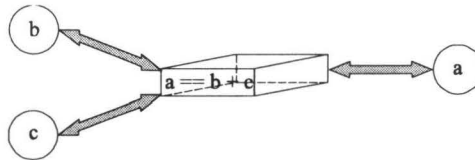


Figure 3.1c: An adirectional dependency.

### 3.2.2 Cyclic Dependencies

Cyclic dependencies in a constraint network may exist when the value of a property is (indirectly) dependent on its own value. The question whether a constraint network contains cyclic dependencies can be answered from different viewpoints. In the case that the constraint network only contains directional dependencies, a cycle is present if and only if it is possible to start at a property and follow the various dependency relations in their specified direction until the original property is reached again. In case the constraint network only contains adirectional dependency relations, a cycle is present if and only if it is possible to start at a property and follow the various dependency relations (a relation may not be followed twice) in an arbitrary direction until the original property is reached again. If the constraint network contains both, directional and adirectional dependency relations, a cycle exists if and only if it is possible to start at a property and follow the various dependency relations (an adirectional dependency relation may not be followed twice and a directional dependency relation may only be followed once in its specified direction) until the original property is reached again.

In case of a cyclic dependency in the constraint network, special actions have to be taken by the constraint system in order to ensure constraint satisfaction. This is necessary for two reasons:

- The dependency is cyclic, which means that the constraint system has to start with initial guesses for the values of some of the properties to start the process of satisfying the cycle and propagating information through it.
- The satisfaction process can take forever when the values of the properties in the cycle do not converge after the constraints in the cycle are satisfied a number of times. In that

---

<sup>1</sup> In the remainder of this chapter the following conventions are chosen:

‘:=’ stands for the assignment of the value of the expression on the right-hand side to the variable on the left-hand side.

‘==’ stands for the equality of the values of the expressions on both the left- and right-hand side. This equality may be enforced by assigning values to the variables on the left-hand side using the values of the variables on the right-hand side or vice versa.

‘=’ stands for an equality between the expressions on the left- and right-hand side for which the way in which this equality is enforced is not explicitly stated; ‘=’ may be realized via ‘:=’ (assignment) or via ‘==’ (equality).

case, the constraint system has to take drastic actions (like limiting the number of traversals of the cycle) to break the cycle.

### 3.2.3 Priorities for the Dependencies

Priorities for the different dependencies can be used by the constraint system to deal with situations of over- and under-constrained constraint networks. Constraint systems typically provide from two up to an infinite number of priority levels. Constraints with a higher priority need to be solved in favor of other constraints with a lower priority.

In theory, different schemes are used by the different constraint system implementations. In practice, however, there are three schemes which are used more frequently than others. The first one is based upon a numerical representation of the priority. Priorities are represented as a range of numbers. The constraint system can easily test whether a constraint has a higher priority than another constraint by verifying the result of the inequality  $\text{priority}(\text{constraint1}) < \text{priority}(\text{constraint2})$ . The advantage of this scheme is that new priority levels can easily be introduced; a unused number in the range of possible priority levels is chosen. A disadvantage of this scheme is that, if no gaps are left between the different existing priorities, it might be problematic to create a new priority which has a priority which lies somewhere in between the highest and lowest priority.

The second scheme makes use of a number of predefined priority levels. The most commonly used classification of priority levels is (from low to high priority) ([Borning et al. 87]):

- weak default:

This priority indicates that the value of a property is derived (using the values of other properties) and that this value should only be changed when absolutely necessary. This level is used to reduce the chances of a situation of under-constrained constraint networks; the weak default value of a property is assigned to that property when no other constraint applies to the property (and thus no specific value for the property can be derived by the constraint system).

- default:

This priority indicates that the value of a property should only be changed when absolutely necessary. This level is used to reduce the chances of situation with under-constrained constraint networks. This level is introduced in addition to the weak default level to make it possible to have a weak default property whose value is derived from the values of default properties. Changing the value of a default property will only cause the update of a derived weak default property and not (indirectly via the weak default property) that of another default property which is used in the derivation of the value of the weak default property.

- prefer:

This priority is the normal priority level at which constraints would be defined. This priority level overrides the default priorities but can still be overridden by stronger constraints.

- strongly prefer:

This priority level exist to provide a level in between the prefer level and the require level.

- require:

The require priority level is used for constraints which must hold, regardless of any other constraint in the constraint network.

The third scheme is a mixture of the two schemes mentioned above; a number of priorities are predefined by the constraint system and are mapped to a numerical representation. In addition to that, new priorities may be introduced.



The introduction of priorities has its effects on the satisfaction process of the constraint system. One of the implications of priorities is that the constraint system has to take into account the priorities of other constraints when a particular constraint has to be satisfied; the satisfaction of a constraint can not be done locally (i.e. more efficient in terms of time and other resources) at the constraint itself anymore, but it has to be done globally (i.e. more expensive in terms of time and other resources) for the whole network. Priorities also imply that the constraint system has to search for that set of constraints so the higher priority constraints are satisfied first and that, at the same time, as many lower priority constraints are satisfied as well. For that task special *comparators* are necessary which can compute the total error. This error is based upon the fact whether a constraint is satisfied (no error), not satisfied (largest error) or only partially satisfied.

The advantage of introducing priorities is that the problem of under-constrained constraint networks may be reduced; (weak) default priorities (also referred to as *stay*-constraints) reduce the number of possible values for the different properties (the values of the properties with a (weak) default constraint should not change). In that way the constraint network becomes more constrained. With the aid of the comparators, it is possible to find a value for a (weak) default constraint which lies the closest to the original value of the property (the *result of least astonishment*).

An over-constrained constraint networks can also be dealt with by using priorities. In that case, the priorities determine which constraints should and which constraints should not be satisfied. Thus, when a constraint network is over-constrained, priorities will not bring about the satisfaction of all the constraints in the constraint network, but they make sure that the most important constraints are satisfied.

### 3.2.4 Cardinality of Dependencies

The cardinality of the different dependencies determines how complicated a constraint network can become. In the literature constraints are called unary constraints, binary constraints and  $n$ -ary constraints. In these cases, the dependency relation affects only one, two or  $n$  properties respectively. In this text the more general notion of *cardinality of a dependency relation* will be used. Using the cardinalities, a finer subdivision can be made:

- 0-1 dependency relation (unary constraint).

- A 0-1 dependency relation is a relation which imposes a certain rule on one property. The value of the property is not dependent on the value of any other property in the constraint network. A typical example of such a dependency relation is a relation which assigns a default value to a property.

- 1-1 dependency relation (binary constraint).

- A 1-1 dependency relation is a relation where the value of one property is dependent on the value of one other property (and vice versa if the dependency is adirectional). An example of a 1-1 directional dependency relation is an assignment relation between two properties. An example of a 1-1 adirectional dependency relation is an equivalence relation between two properties.

- 1-m dependency relation ( $n$ -ary constraint)

- A 1-m dependency relation is a relation where the value of one property is used to determine the values of a number (at least one) of other properties. An example of such a relation is the situation where different graphical objects are aligned to a grid. The position of the different objects (= a set of dependent properties) is then dependent on the value of the grid-size (= a property).

- m-1 dependency relation ( $n$ -ary constraint)

- A m-1 dependency relation is a relation where the value of one property is dependent on the values of a number (at least one) of other properties. An example of such a relation is

the situation where the value of one property is defined as the sum of the values of two other properties. Note that if the dependency relation is adirectional, there is no distinction between 1-m and m-1 dependency relations.

◦ m-m dependency relation (n-ary constraint)

A m-m dependency relation is a relation where the values of a set (at least one) of properties is dependent on the values of a number (at least one) of other properties.

In practice, constraint systems may support a combination of the above mentioned types of dependency relations. Note that 1-1 dependency relations are a subset of the m-1 dependency relations and the 1-m dependency relations and that these are again a subset of the m-m dependency relations. This aspect of the constraint network does not have much influence on the propagation of information and the satisfaction of the constraints. It only affects the number of dependency relations needed to describe a certain situation in terms of constraints and, more importantly, the complexity of the situation that can be described. On each line in the table below, relations are shown which can be specified using the dependency relations indicated on that same line but not by using the dependency relations mentioned in the previous line:

type of dependency relation	dependency relation	description
0-1 dependency relation	$v := 5$	v is assigned a default value.
1-1 dependency relation	$v \leq 2\pi \cdot r^2$	v's value should not exceed the area of a circle with radius r.
m-1 dependency relation	$a := b + c$	a is defined to be the sum of b and c.
m-m dependency relation	$  (x,y) - (p,q)   < 100$	the distance between the points (x,y) and (p,q) is less than 100.

### 3.2.5 Dynamic Abilities of the Constraint Network

Besides the different kinds of dependencies which can be used to build a constraint network, a constraint system can also differ in the way in which dependencies can be added to and/or removed from it. Some of the constraint systems only allow the adding of constraints to the constraint network before the constraint system is initialized and has started to satisfy the constraints in the constraint network. In such a *static* constraint system, all the constraints have to be known to the constraint system in advance. Once the constraint system is initialized, no constraints may (or can) be added to and/or removed from the constraint network.

In *dynamic* constraint systems, constraints can be added to and/or removed from the constraint network while the constraint system is already running. This allows for the addition of constraints to the constraint network at the moment they are necessary (which means that the constraint system is not burdened with these constraints in an earlier phase when they were not used anyway) and the removal of constraints after they have outrun their usefulness (and thus not burden the constraint system with constraints which are not going to be used anymore).

In the case of a dynamic constraint system, the way information is propagated through the constraint network may differ each time a constraint is satisfied due to the fact that new constraints are added to or removed from the constraint network. Also the order in which the different constraints have to be satisfied may change after the structure of the constraint network has changed. The constraint system therefore has to take the changes in the structure of the constraint network into account when propagating information or satisfying constraints. This is important especially when the structure of the constraint network is changed while the constraint system is propagating information or satisfying constraints; in those situations conflicts may occur when the

constraint network is unstable due to these updates. Several constraint systems therefore do not allow updates of the constraint network while propagating information or satisfying constraints.

### 3.2.6 Number of Constraint Networks

The last aspect in which constraint systems can differ that is discussed in this section is the number of constraint networks which are used by the constraint system. Different constraint systems use different methods to satisfy the constraints in their constraint network. For administration purposes different constraint networks can be used simultaneously. Examples of these networks are:

- a network to store the properties and the relations between them (this happens in constraint systems which can only apply constraints on special constraint variables and not an arbitrary variables).
- a network to store the different relations in the constraint network, possibly ordered by priority (this happens in systems which can apply constraints on ordinary variable and where only a reference to that variable is necessary).
- a network to store the (temporal) value of the properties as computed by the constraint system during constraint satisfaction (this may happen in constraint systems where the history of the computations of a cycle is stored).

The number and nature of the networks used by the different constraint systems is very specific to these systems. The networks are data structures which are used by the various constraint systems to store their information regarding the process of constraint satisfaction and information propagation. They have no effect on these processes themselves.

### 3.3 Propagation Techniques

Different problems may be specified in different ways. This also means that different constraints, connected in the constraint network via different paths, can be used to describe a problem in terms of constraints. The way in which a problem is described in terms of constraints may be guided by reasons of readability (one way of describing a problem may be easier to understand than another), by the possible dependency relations which are supported by the constraint system (described in the previous section) and/or by the way in which the constraint systems propagates information through the constraint network. The way in which a problem is described can have an effect on the efficiency of the satisfaction process; knowledge about the way in which information is propagated can be used to come to a more efficient satisfaction process for the problem. It is therefore important to understand what kind of propagation mechanism is used in a certain constraint system. In general, three different principles in propagation techniques can be distinguished: incremental versus all-at-once (§3.3.1), sequential versus parallel (§3.3.2) and refinement versus perturbation (§3.3.3).

#### 3.3.1 Incremental versus All-at-Once

There exists a gliding scale for the granularity in which a constraint system can solve the different constraints in the constraint network. One extreme is the situation where the constraint network solves all the constraints in the constraint network at once. This means that the whole set of constraints is considered at the same time. At the end of the process of constraint satisfaction, the constraint system will provide one solution which satisfies all constraints. No temporal results are assigned to the different properties; once the constraint system has computed the end-result, the properties are assigned their final value. This approach is very useful in cases where the values of the different properties have many dependencies with each other (there exists a dense constraint network). An example where this approach may be very effective is when a set of equations and/or

inequalities has to be solved. In that case the values of the different coefficients in the equations/inequalities are strongly dependent on each other.

The other extreme is marked by constraint systems which satisfy their constraints one by one. This means that a constraint system will satisfy one constraint, assigns the results to the different properties and looks for another constraint to satisfy. During this process, a property may be assigned another value later due to the satisfaction of another constraint. Once a constraint is satisfied, the other constraints in the constraint network may be *narrowed*; the number of ways in which these constraints can be satisfied is reduced due to the previous assignment of certain values to a number of properties. As a result, the constraint system may reach a situation where no unsatisfied constraints can be satisfied anymore because they are all narrowed too much. In that case, the constraint system may fail as there may not exist a solution at all, or the constraint system may backtrack and undo some of its satisfaction work. The combined actions of backtracking and undoing some actions is applied to some of the constraints which are then said to be *widened*; assignments to some of the properties are revoked and the number of ways in which a constraint can be satisfied may increase again. The approach to satisfy constraints one by one is effective in cases where the constraint network is rather sparse; the chances that a constraint system has to backup will become smaller when the network gets more sparse.

In practice, constraint systems exist which use an approach which lies somewhere in between these two extremes. Examples of this are constraint systems that use a technique which splits the constraint network in parts. Each part itself is solved at once, but the different parts are solved one by one. The size of the parts may vary and depend on the status of the individual constraints and the structure of the entire constraint network together.

### 3.3.2 Sequential versus Parallel

The position of a constraint system with respect to the second principle of propagation is determined by whether the information is propagated in a sequential or a parallel way. For this principle it also holds true that except for the two extremes which satisfy all constraints sequentially or in parallel, constraint systems can adopt an approach which does something in between. In that case parts of the constraint network can be satisfied in parallel while the constraints within each part are satisfied sequentially (or the other way around, where parts of the constraint network can be satisfied sequentially while the constraints within each part are satisfied in parallel).

Note that incremental and sequential constraint satisfaction are not the same thing, nor are all-at-once and parallel constraint satisfaction. In the all-at-once case, the constraint system works like a black box where a set of constraints goes in and an assignment to the various properties comes out. This black box can satisfy the constraints in parallel but also sequentially. It is also possible to have an incremental constraint satisfaction technique which satisfies its constraint in parallel. In that case, the constraint system can satisfy several constraints at the same time (i.e. parallel). However, the decision which constraint has to be satisfied next, is made separately for every constraint and this decision depends on the values assigned to the different properties by earlier satisfied constraints.

Details about a mathematical foundation of concurrent constraint systems can be found in [Sarawat 93]. However, this foundation is based on logic constraint systems which, therefore, is beyond the scope of this thesis.

### 3.3.3 Refinement versus Perturbation

Finally, a constraint system can be characterized by whether it uses refinement to satisfy its constraint network or perturbation. In the latter case, (semi-)random values are assigned to the

various properties in the constraint network. Using these initial values, the different constraints are satisfied. If not all the constraints can be satisfied, a new (random) guess is made for some or all of the properties in the constraint network and the constraints are satisfied again.

The approach of refinement uses the result of the previously satisfied constraints to satisfy other constraints. Values of properties computed and assigned by a constraint will not change anymore due to constraint satisfaction. Using refinement, the set of possible assignments for a property will decrease during constraint satisfaction. This means that information, once propagated, cannot be revised.

### 3.4 Satisfaction Techniques

The satisfaction techniques which are used by a particular constraint system are closely related to the type of dependencies which are allowed in the constraint network, the possible structure of the constraint network itself and the propagation technique used. This section will discuss the most common constraint satisfaction techniques. Note that it is possible (and sometimes necessary) to combine those techniques in order for a constraint system to satisfy the constraints in the constraint network.

Some constraint systems use these satisfaction techniques in combination with planning strategies. In that case, the constraint system will first analyze the structure of the constraint network and the different constraints in it, then find a (semi-)optimal solution after which the individual constraints are satisfied in the order defined by the plan using one of the techniques described below. The advantage of making a plan before the constraints are actually satisfied is that the constraint system can use the global structure of the constraint network to find this plan. This approach is especially convenient in cases where constraints can have different priorities. However, in cases where constraints are satisfied in parallel, it is difficult (if not impossible) to find a relative order in which the different constraints should be satisfied [Hintum 95].

#### 3.4.1 Propagation of Known States

This technique is also called *local propagation*. This technique propagates the results of constraint satisfaction from one property to the next one (if both properties are connected by a dependency relation). Propagation of known states is a satisfaction technique which looks for one-step deductions; when the values of enough properties are known so the values of the other properties, referred to in the dependency relation, can be deduced, the constraint system will satisfy the corresponding constraint. Initially those constraints may be satisfied which can deduce the values of other properties by using the value of the property which was just changed. Gradually, the values of more and more properties may be used as the constraint system satisfies more constraints.

This method has one major advantage; it is fast. Furthermore, it can be used with either directional and adirectional dependency relations, which may have cardinalities 0-1, 1-1, 1-m, m-1 and m-m. It is also possible to use priorities for dependency relations when propagation of known states is used and it is possible for the constraint system to support dynamic addition and/or removal of constraints. This technique is also applicable in all possible configurations of incremental/all-at-once, sequential/parallel and refinement/perturbation systems. However, it has one drawback. It may be clear that this technique is not capable of satisfying cyclic dependencies; in a cyclic dependency, the constraint system always lacks the value of at least one property to compute the value of other properties mentioned in the dependency relation.

### 3.4.2 Propagation of Degrees of Freedom

Another technique which can be used to satisfy the different constraints in a constraint object is propagation of degrees of freedom. This technique tries to prune away as many constraints from the constraint network as possible. First those properties (and their dependency relations) are pruned which have the most degrees of freedom; i.e. their value can easily be adapted without making the dependency relations it is involved in inconsistent. For this pruned (and thus less complex) constraint network, the constraint system may again look for properties to prune. Once no more properties can be pruned, the constraint system has to satisfy the remaining constraints. This has to be done using another technique than propagation of degrees of freedom, as this technique can only prune properties. Therefore, propagation of degrees of freedom is often used in combination with propagation of known states. Afterwards, the pruned properties are added to the constraint network again. Properties are added in reverse order to the order in which they were pruned. Each time a property is added to the constraint network, its value is made consistent with the other properties. This is possible, because they had many degrees of freedom.

This technique can be used with directional and adirectional dependency relations which have an arbitrary cardinality and which can have different priorities. Constraints can be added to and removed from the constraint network, but this can not be done while the constraint system is satisfying the constraint network. The order in which properties are pruned is closely related to the structure of the constraint network. Changing this structure during constraint satisfaction will make the satisfaction process error-prone. Note that propagation of degrees of freedom is a technique which seems to need several constraints networks; to realize the pruning and adding of properties, not the actual constraint network, but a copy of the network should be used. This technique is a typical example of a refinement approach; the different properties are only once assigned a value (as they are added to the constraint network again). Furthermore, the different pruning steps (and the reverse adding steps) need to be done sequentially but within each step the different constraints can be satisfied in parallel. Thus, it is a technique which can be used in a mixture of sequential and parallel constraint satisfaction. It can also be used in a mixture of the incremental/all-at-once approach. The different steps have to be done incrementally, one after another. But the satisfaction of the different constraints with each step can be done all-at-once.

### 3.4.3 Relaxation

Relaxation is an example of a technique which uses perturbation. This technique will make an initial guess for (some of) the values of the different properties in the constraint network. Using these initial guesses, the values of the remaining properties (if any) are deduced and an error is estimated. Using this estimated error, the initial guess is adjusted and taken as a new guess. This process continues until the error is lower than a certain threshold.

The main advantage of this technique is that it can be used to satisfy cyclic dependencies. However, this technique also has a number of disadvantages. First of all, this technique is computationally expensive. Also, it can only be used in continuous numeric domains where the result can be approximated. Furthermore, the constraint relations have to be linear in order for this technique to work ([Cournarie et al. 91]). The biggest disadvantage of this technique, however, is the fact that, in the case of an under-constrained constraint network, the solution which is found depends on the initial guess that is used. That means that this technique may produce unpredictable results.

With respect to the various types of dependency relations and constraint networks, the following can be said: relaxation can be used in constraint networks where the dependency relations can be directional as well as adirectional and where they can have an arbitrary cardinality. Constraints may be added to and deleted from the constraint network dynamically and constraints may have



priorities. Relaxation may typically use two constraint networks; one to store the actual values and one to store the temporal values which are computed using the initial guess and which are used to estimate the error of the solution. Relaxation can be used in an incremental as well as an all-at-once approach and in either a sequential or a parallel environment.

### 3.4.4 Prototyping

In [Cournarie et al. 91] a constraint system is described which is based on prototyping. The prototypes in this model (which are considered to be ordinary objects) allow for the shared usage of properties; by delegating the behavior of ordinary objects to the prototype, local modifications to the prototype can have a global impact on all the objects which have delegated (part of) their behavior to that prototype. On the other hand, changes made locally to the objects which have delegated their behavior do not have any effect on other objects or the prototype itself. The problem, however, is the fact that by prototyping alone, only very simple constraints can be specified (i.e. only common properties can be defined). Information is propagated (via the delegation mechanism) from prototype to delegator. Constraint satisfaction is not necessary.

### 3.4.5 Graph and Term Rewriting

Both techniques, Graph Rewriting and Term Rewriting, are based on the dynamic alteration of (a copy of) the constraint network. How this alteration of the network has to be done is defined in so-called *rewrite rules*. Using the rewriting technique, the constraint system starts rewriting that part of the constraint network which has been triggered. As a result of this process, parts of the constraint network may be replaced by the result of this rewriting. Depending on the context in which the constraint system is used, a copy of the constraint system may have to be made so that important information in the original network is not lost after constraint satisfaction ([Leleer 88], [Wilk 91], [Horn 92]).

Information in the constraint network is not really propagated; when a rewrite rule is applied, information is reordered. During this process of reordering some information is first used to simplify the rest of the information and then left out. In the example below, a possible representation of the constraint network " $a := b + c$ " and the possible rewrites are shown. In the first rewrite step, the information that the value of  $b$ , which is 4, and the value of  $c$ , which is 2, have to be added is simplified to the information that the resulting value is 6. All information about  $b$ ,  $c$  and the addition is removed from the constraint network. After the second step, all information about how the value of  $a$  is computed is lost. In return for that loss of information, the whole network was simplified; the value of  $a$  is 6 (note that the figures in this section represent the rewrite-graph, in this case " $a := b + c$ ", and not the constraint network):

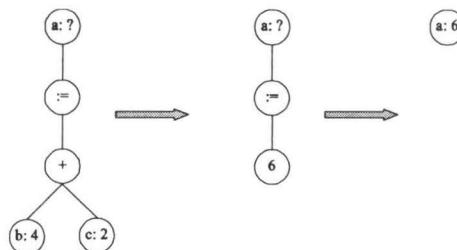


Figure 3.2: Constraint Network Rewriting.

Rewriting of the constraint network takes place as long as the constraint system can find rewrite rules which can be applied to (part of) the constraint network. The advantage of rewrite rules is the that they can use so-called patterns; generic networks which contain variables that can be matched against a part of the actual constraint network. When a pattern is matched (i.e. in the constraint network a subpart can be identified where the non-variable parts of the pattern match the parts of the identified subpart), the values for the variables in the pattern are replaced by the values of the matched constraint network. This feature allows a programmer, for instance, to define how an adirectional dependency relation can be converted into a number of directional dependency relations.

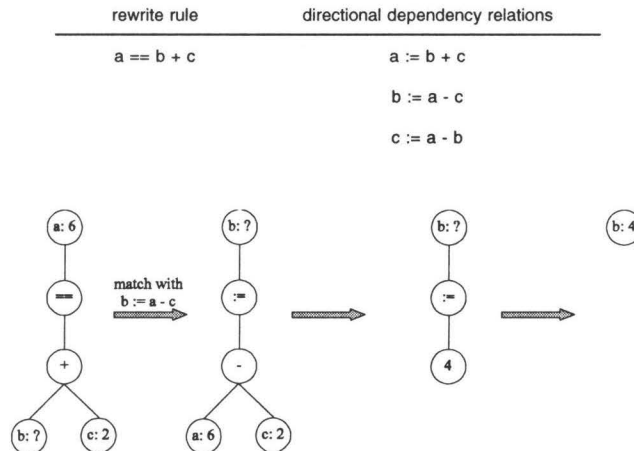


Figure 3.3: Pattern Matching in Constraint Network Rewriting.

The difference between graph rewriting and term rewriting lies in the fact that graph rewriting has the notion of shared sub-networks; within a constraint network arguments of a rewrite rule may point to the same sub-network. When this sub-network is rewritten, it is rewritten once. In the case of term rewriting, for each pointer to a sub-network, a copy of that sub-network is rewritten. This difference in rewriting may lead to different results:

```

square == radius * radius
radius := 5
radius := -5
    
```

In figure 3.4a no sharing of sub-networks is done and the values for the two occurrences of 'radius' are chosen independently of each other. In that case, it is possible that for one occurrence the value of 5 is chosen and that for the other occurrence the value of -5 is chosen. In this case, the result of 'square' may become -25 (which is obviously not what was intended). In figure 3.4b, the situation is presented where the value for the pattern 'radius' is chosen once and that, because the sub-network is shared, the result for 'square' will always be 25 (either the value -5 is chosen for 'radius' and 'square' equals  $-5 * -5$  or the value 5 is chosen and 'square' equals  $5 * 5$ ).



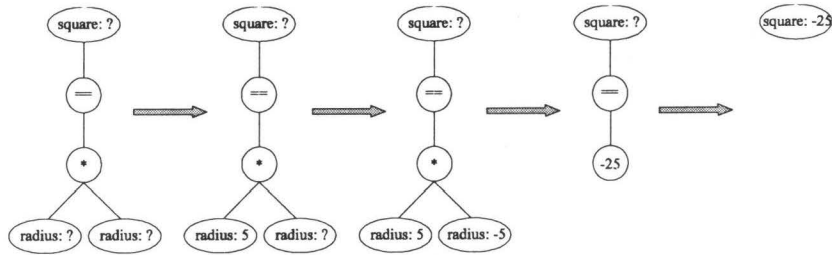


Figure 3.4a: Term Rewriting.

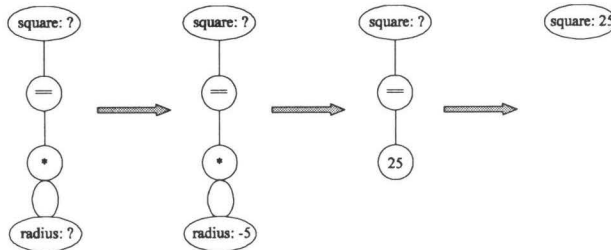


Figure 3.4b: Graph Rewriting.

The rewrite rules have a strong direction; they have a pattern and a substitution network which replaces (part of) the constraint network which matches the pattern. However, because it is possible to specify more than one substitution network for a pattern, adirectional dependencies can be simulated (compare with the situation in §3.2.1 where you had the relation  $a = b + c$  and three possible ways to enforce this relation  $a := b + c$ ,  $b := a - c$  and  $c := a - b$ ). Patterns allow for the matching of any pattern against (part of) the constraint network and, therefore, all cardinalities are supported. It is also possible to specify cycles in the patterns. Thus cycles can be solved by rewriting techniques. Priorities are not supported in general, but it is possible to dynamically add rewrite rules. Rewrite mechanisms are clearly an incremental satisfaction technique; the matching of patterns has to be done one by one; after each rewrite step, any part of the constraint network can be altered so that all the rewrite rules have to be considered again. Nevertheless, it is possible to execute different rewrite step in parallel if the patterns of the different rewrite rules match with subnetworks which do not overlap. Finally, the rewriting technique can be said to be a refinement satisfaction technique; using the rewrite rules, the structure of the constraint network itself and the values of the different properties within it are refined according to the rewrite rules.

#### 3.4.6 Redundant Views

Redundant views (also called alternate view ([Gosling 83])) cannot really be considered a constraint satisfaction technique. However, redundant views can help a constraint system to satisfy constraints. Redundant views exist when information about relations between the various properties is stored more than once (possibly in another way) in different dependency relations.

By combining several dependency relations which constrain the same properties, the value of those properties may be computed using the values of less other properties (variable elimination). This can best be shown by the following example:

The set of dependency relations:

$$\begin{aligned}\text{mid} &== \text{left} + \text{distance} \\ \text{right} &== \text{mid} + \text{distance}\end{aligned}$$

can not be solved when only the values of left and right are known. In that case, the constraint system will end up with the following equations (assume left = 4, right = 8):

$$\begin{aligned}\text{mid} &= 4 + \text{distance} \\ 8 &= \text{mid} + \text{distance}\end{aligned}$$

These equations can only be solved when either the value of mid or distance is known. However, from the original two dependency relations, the following redundant information can be deduced:

$$\text{right} == \text{left} + 2 * \text{distance}$$

Using this dependency relation, the constraint system can satisfy the constraints:

$$\begin{aligned}8 &= 4 + 2 * \text{distance} && \Rightarrow \text{distance} = 2 \\ \text{mid} &= 4 + 2 && \Rightarrow \text{mid} = 6\end{aligned}$$

In the same way redundant views can also be used to solve cyclic dependencies; the redundant information is used to create a new dependency relation where one property from the other dependency relations is eliminated:

The following set of (cyclic) relations:

$$\begin{aligned}a &= b - p \\ b &= c - p \\ c &= a + q\end{aligned}$$

can not be solved when only two values of the set {a, b, c} are known. However, adding the following (redundant) relation to the set, no problems exist anymore to determine the third value:

$$q = 2 * p$$

The motivation behind redundant views is that the more degrees of freedom exist for a variable, the easier it is to solve relations in which that variable occurs. Ideally, the constraint system itself should be able to create redundant views. However, this requires the constraint system to have some knowledge about the semantics of the problem domain. For some areas the semantics may be formalized rather easily (for instance for algebra), but for other areas it may be more difficult to provide the constraint system with the proper formalized semantics (for instance for translating text from one human language into another). Furthermore, the constraint system should be equipped with different semantics for every distinct area for which the constraint system is used. In practice there are two kinds of systems with respect to redundant views:

- those systems that can deduce redundant dependency relations from the given set of relations (and which have mostly only a small application area). An example of such a system is Bertrand ([Leler 88]) which is able to apply the basic rules of algebra on relation specifications and as such is able to find the relation ' $q = 2 * p$ ' for the relations ' $a = b - p$ ', ' $b = c - p$ ' and ' $c = a + q$ '.
- those systems that can support redundant information without having the ability to actually deduce them. An example of such a system is Equate ([Wilk 91]) that is able to choose among several alternative constraint relations to assign a value to a property.

Because redundant views is not a constraint satisfaction technique, its relation to the direction, priority, cardinality, and dynamic behavior of dependencies is not relevant nor is its relation to the various satisfaction techniques.

### 3.5 Constraint Systems

In this section, different constraint systems are discussed. This discussion will be restricted to constraint systems in an object-oriented environment. This restriction results from the fact that the constraint system that will be discussed in the next chapters, is implemented in the object-oriented environment MADE. This section aims to give an overview of some of the most important constraint systems in object-oriented environments which can guide the design of the MADE constraint system.

In the design of the MADE constraint system, the priority lies not on efficiency. Constraint systems will always cause overhead in the computation (no matter how small this may be) and thus the choice to use constraints instead of brute force programming to solve a problem may indicate that it is more important to describe the solution to the problem in an abstract and elegant way than to describe the most efficient way. In the latter case, constraint programming should be avoided and the relations between objects should be programmed directly. However, new, more powerful computers are designed every day and the gap between what is efficient and what is elegant becomes smaller with every new model. As a reason of that, the following constraint systems should not be judged on their efficiency but on their elegance, the expressive power of their dependency relations and the ease with which new dependency relations and new kinds of dependency relations can be added. The MADE constraint system will be designed with the same goals in mind.

Before this overview is given, it is important to realize there are two reasons for the existence of a tension between the object-oriented programming paradigm and constraint programming ([Blake 91]):

- the definition of classes and member functions is done in an imperative way whereas the description of constraints is normally done in a declarative fashion.
- classes should hide as much information as possible from other objects (information hiding) while a constraint system needs to have as much information as possible from a class to satisfy constraints.

The constraint systems discussed in this section primarily deal with the second problem. Properties should be well defined in the interface of an object; Alien (§3.5.1), Equate (§3.5.2), GICS (§3.5.3), Kaleidoscope (§3.5.4), Procol (§3.5.5) and ThingLab (§3.5.6). The first problem is addressed only in Kaleidoscope, Procol and ThingLab.

At the end of each section an example is given of how the definition of the MidPointLine problem would globally look in that constraint system. In the MidPointLine problem three points are defined. Two points are the endpoints of a line and the third point is defined to lie on the middle of the line that is defined by the other two points. Moving either one of the endpoints or the midpoint causes the constraint system to update the other two points such that the midpoint remains on the middle of the line.

#### 3.5.1 Alien

The constraint system Alien ([Cournarie et al. 91]) aims to integrate constraint programming and object-oriented programming. It is based on prototypes and delegation. Prototypes allows for local modifications which have a global impact while delegation allows for the dynamic modification of

individual objects. This approach is considered to be more flexible than the framework of class types and class instances.

Each prototype can export a number of slots. A slot is the name used in Alien to identify a property. Slots behave like ordinary object variables with this difference, that the constraint system is triggered whenever the value of a slot is modified. Constraints are objects which can connect slots of different object to their own *entries*. The entries are the variables of the dependency relations. Each constraint has a satisfaction method which, using these entries, describes how the values of the different slots can be computed. Using slots, Alien has a controlled way to break the information hiding principle of the object-oriented programming paradigm; only designated variables may be changed by the constraint system.

Constraints in Alien are adirectional. However, constraints can be created which are directional. For that purpose, a distinction has to be made between *master entries* and *slave entries*. When the value of a slot, connected to a master entry, is modified, the values of all the other slots involved in the constraint are also updated. When the value of a slot, connected to a slave entry is modified, only the values of slots connected to slave entries are updated.

Alien uses local propagation to satisfy its constraints. This implies that it cannot solve cyclic dependencies between the various slots. It is possible to add slots to an object and to add constraints to the system dynamically. Constraints have a 1-m cardinality; the modification of one slot can affect an arbitrary number of other slots.

Alien uses a sequential, refinement propagation technique; starting at the slot which is modified, constraints are satisfied one by one, while modifying slots of other objects. To deal with over-constrained constraint networks, Alien makes use of priorities. These priorities provide a global control over which constraints are satisfied first; constraints of the same priority are considered all-at-once at the same time by the constraint system whereas the individual constraints within the same priority level are considered one by one. Thus, Alien uses a mixed approach of the incremental and all-at-once propagation technique. Under-constrained constraint networks are dealt with by applying the *principle of least astonishment*.

```

Prototype Line {
  Point point1, point2;           // slots to define the endpoints of the line
  ...
};

Prototype MidPointLine {
  Point midpoint;                 // slot to define the midpoint of the line
  ...
};

Constraint MidPoint {
  // relation: 1*point1 + 1*point2 - 2*midpoint = 0
  Master Line.point1: 1           // moving one point may cause the constraint system to update
  Master Line.point2: 1           // the values of the other points; therefore every slot is
  Master MidPointLine.midpoint: -2 // defined as master.
};

```

### 3.5.2 Equate

The constraint system Equate ([Wilk 91]) combines constraint programming and the object-oriented programming paradigm by demanding that variables of an object (i.e. the properties of the constraints) may not be inspected or modified in any other way than through the interface of that object. For that reason, Equate demands a well defined interface for all objects.

Constraints in Equate are always expressed as equations. Therefore, the problems for which Equate can be used have to have a numerical representation. For constraint satisfaction, this system uses a backward chaining technique in combination with rewrite rules. The way in which constraints are satisfied is very similar to the way in which logical programming languages resolve clauses; decompose large constraints into smaller constraints for which solutions are known and, in the end, combine the various solutions to construct the solution for the large constraint.

Equate uses a rewriting strategy for two purposes; to find a solution directly (i.e. a member function invocation of an object to update the variables of that object) or to convert an equation into an equivalent set of equations. When one equation is replaced by an equivalent set of equations, the equations in this set are rewritten from left to right; i.e. in an incremental way. When all equations are rewritten and a solution is found for all rules, the different solutions have to be combined again. For this purpose, a dependency network is built based upon the various variables used in the solutions. This dependency network is then used to order the calls to the different member functions so variables are used only after they have been assigned the proper (constrained) value.

The Equate system does not call the member functions itself. It will only provide a plan which states in which order different member functions of various objects have to be invoked to make the values of the variables consistent with the constraints imposed on them.

The Equate system can further be characterized as follows. The rewrite rules are directed, can not form cyclic dependencies with each other and cannot be assigned priorities. Rewrite rules can be added and removed dynamically. The cardinality of rewrite rules is m-m; however, the set of variables which are referred to before and after rewriting is the same.

```
Object Line {
    void setpoint1 (int);           // methods to set and get the value of the endpoints
    void setpoint2 (int);
    int getpoint1 ();
    int getpoint2 ();
};

Object MidPointLine : Line {
    void setmidpoint (int);         // methods to set the value of the midpoint
    int getmidpoint (int);
};

midpoint(line) = exp ←            // constraint rewrite rule to set midpoint (1st alternative)
    line.setpoint1 (2*exp - line.getpoint2()) // point1 is defined in terms of midpoint (exp) and point2

midpoint(line) = exp ←            // constraint rewrite rule to set midpoint (2nd alternative)
    line.setpoint2 (2*exp - line.getpoint1()) // point2 is defined in terms of midpoint (exp) and point1

point1(line) = exp ←              // constraint rewrite rule to set point1 (1st alternative)
    line.setpoint1 (exp);          // set point1
    midpointline.setmidpoint ((exp + line.getpoint2()) / 2) // adjust the value of midpoint accordingly

point1(line) = exp ←              // constraint rewrite rule to set point1 (2nd alternative)
    line.setpoint1 (exp);          // set point1
    line.setpoint2 (2*midpointline.getmidpoint() - exp) // adjust the value of midpoint accordingly

point2(line) = exp ←              // constraint rewrite rule to set point2 (1st alternative)
    line.setpoint2 (exp);          // set point2
    midpointline.setmidpoint ((line.getpoint1() + exp) / 2) // adjust the value of point1 accordingly

point2(line) = exp ←              // constraint rewrite rule to set point2 (2nd alternative)
```

```

line.setpoint2 (exp);           // set point2
line.setmidpoint1 (2*midpointline.getmidpoint() - exp) // adjust the value of point1 accordingly

```

### 3.5.3 GICS

In [Rankin 91] the Graphics Interactive Constraint System (GICS) is described. This constraint system is designed for object-oriented graphics systems. GICS puts a more restrictive demand on the interface of objects than Equate; every object which is used in a constraint must have specified a number of predefined member functions in its interface. GICS can make this strong demand as the application area for this constraint system is so narrow.

Objects which can be used in constraints are called *Graphical Elements* (GE) and *Graphical Objects* (GO). GEs are graphic output primitives such as a line, a circle, an arc, etc. Possible (internal) constraints within the GE itself (for instance, fixed line length, fixed angle, fixed end point, etc.) are considered to be the responsibility of the GE itself and will therefore not be maintained by the constraint system GICS. GOs are objects around various GEs connected via external constraints. The properties of these constraints are always points. Modifying the value of such a point via a predefined member function of the interface triggers the constraint system. The external constraints are maintained by GICS; by calling inspection functions and assignment functions of the various GEs, GICS maintains the different constraints for a GO. Within a GO, dependencies between GEs may be cyclic. In that case, GICS uses a relaxation technique to satisfy the constraints. Furthermore, dependency relations are adirectional, do not have priorities and have m-m cardinality. Information is propagated incrementally using refinement.

```

GO Line {
    GE point1;           // endpoint of line
    GE Point2;           // endpoint of line
    ...
};

GO MidPointLine {
    Line lineobject;     // reference to the line
    GE midpoint;         // midpoint of line

    external constraints:
        midpoint = (lineobject.point1 + lineobject.point2) / 2 // changing point1 or point2 automatically updates midpoint
        lineobject.point1 = 2*midpoint - lineobject.point2    // changing midpoint or point2 automatically updates point1
        lineobject.point2 = 2*midpoint - lineobject.point1    // changing midpoint or point1 automatically updates point2
};

```

### 3.5.4 Kaleidoscope

Kaleidoscope ([Freeman-Benson 90]) is a constraint system which uses the notion of time to combine constraint programming and the object-oriented programming paradigm. The emphasis in this integration is on the difference between the declarative nature of constraints and the imperative nature of objects definitions.

In Kaleidoscope, three different types of constraints are defined: equality constraints, primitive constraints and complex constraints. Equality constraints are adirectional constraints between two properties and their values are always the same. Primitive constraints are some predefined constraints over a given set of domains. Primitive constraints may be cyclic, redundant and directional. Complex constraints are constraints over objects whose internal states are already constrained by other lower-level constraints.



Because Kaleidoscope only supports predefined constraints and constraints which are combinations of predefined constraints, it is possible to specify only which constraints should be used in a certain situation (declarative) and not how this constraint should be maintained (imperative). This also shows the weak point of Kaleidoscope; it is not possible to use constraints which cannot be built by combining predefined constraints.

Constraints in Kaleidoscope can be assigned priorities: required, strong, medium, weak and very\_weak. The cardinality of the various constraints varies; equality constraints have a 1-1 cardinality, primitive constraints have a 1-1 or 1-m cardinality and complex constraints may have m-m cardinality. Constraints cannot be added dynamically. The constraint satisfaction is based on refinement and is performed incrementally.

```
Object MidPointLine {
    Line* lineobject

    always: required                               // defines the strength (required) of the constraint relation
        midpoint = (lineobject.point1 + lineobject.point2) / 2 // specify an equality constraint relation
};
```

### 3.5.5 Procol

Procol ([Laffra et al. 91], [Laffra 92]) is an object-oriented programming language which supports constraint programming. Constraints in Procol are declared per object and have a declarative part and a procedural part. The declarative part of a constraint can specify that the procedural part of the constraint has to be executed when a certain member function of a certain other object is called. This declarative part thus specifies what the constraint has to do (invoke another function) and at which point in time (when a specific member function of a specific object has been called). The procedural part of the constraint is a list of statements. This part describes, in an imperative way, how the constraint can be satisfied.

Constraints in Procol are directional, may not be cyclic, have no priorities assigned and cannot be dynamically added to the constraint system. As the properties of the constraints in Procol are the member functions, the cardinality of the constraints is 1-m; the invocation of one member function can lead to the invocation of several other member functions.

Procol uses local propagation. It is also one of the few systems which actually satisfies its constraints completely in parallel.

```
Obj      Line
Declare  Point point1, point2;           // the two endpoints of the line
Protocol ...
Init     ...
Cleanup  ...
Actions  setpoint1 = { ... }             // the methods to which constraints
        setpoint2 = { ... }             // can be attached and which allow
        getpoint1 = { ... }             // to set and retrieve the values
        getpoint2 = { ... }             // of the endpoints.
EndObj   Line

Obj      MidPointLine (object Line lineobject)
Declare  midpoint: Point;
Protocol ...
Init     constraint lineobject.setpoint1 → update1; // trigger constraint 'update1' when point1 is updated
```

```

constraint lineobject.setpoint2 → update1; // trigger constraint 'update1' when point2 is updated
constraint setmidpoint → update2; // trigger constraint 'update2' when midpoint is updated
Cleanup
unconstraint lineobject.setpoint1; // deactivate constraints
unconstraint lineobject.setpoint2; // ...
unconstraint setmidpoint; // ...
Actions
update1 = { midpoint = (lineobject.getpoint1 + lineobject.getpoint2) / 2; } // constraint 'update1'
update2 = { // constraint 'update2'
    delta = midpoint - (lineobject.getpoint1 + lineobject.getpoint2) / 2;
    lineobject.setpoint1 = lineobject.getpoint1 + delta;
    lineobject.setpoint2 = lineobject.getpoint2 + delta;
}
setmidpoint = { ... }
getmidpoint = { ... }
EndObj
MidPointLine

```

### 3.5.6 ThingLab I and ThingLab II

ThingLab I ([Maloney et al. 89]) and its successor ThingLab II ([Borning 81], [Borning et al. 87], [Freeman-Benson et al. 90]) are used for constructing interactive, graphic simulations of experiments in physics and geometry. ThingLab was build to be able to develop basic constraint blocks which, consequently could be used by others to build a particular simulation.

Constraints in ThingLab consist of a *rule* and a set of *methods*. The rule is used to check whether a constraint is satisfied or not and the methods are used to satisfy the constraint when necessary. The different methods are generated automatically by ThingLab using the rule. The rule is adirectional whereas the different methods are directional.

Constraints in ThingLab have m-m cardinality and may be added dynamically. In ThingLab I, constraints may be cyclic, but in ThingLab II this is no longer allowed. ThingLab is capable of solving under- and over-constrained constraint networks.

ThingLab makes use of four comparators to satisfy the different constraint on the various priority levels. It also uses more than one satisfaction technique: propagation of known states, propagation of degrees of freedom and relaxation. To avoid the use of relaxation as much as possible, ThingLab also supports redundant views.

Information, internal to the object, is exported to the constraint system by means of *paths*. A path reflects the hierarchical structure of objects and describes how this hierarchy should be traversed to find the property which is constrained. The constraint system modifies the value of a property by giving the path to the object, mentioned as the root in the path, which then passes the path, from which its own reference is removed, to the new root and so on. When the last object, referenced in the path, is reached, this object will modify the value of the proper variables. This approach ensures that only the object itself changes the values of its own variables instead of the constraint system and thus ensures that the encapsulation of data is preserved as much as possible.

```

Class Line
SuperClasses
    GeometricObject
Part Description
    point1: a Point // an endpoint of the line
    point2: a Point // an endpoint of the line

Class MidPointLine
SuperClasses
    GeometricObject
Part Description

```



line: a Line	// reference to the line
midpoint: a Point	// the midpoint
<b>Constraints</b>	
midpoint = (line.point1 + line.point2) / 2	// specify the constraint relations
// midpoint ← (line.point1 + line.point2) / 2	// these are the three methods which are automatically
// line.point1 ← midpoint*2 - line.point2	// generated by ThingLab and thus do not have be
// line.point2 ← midpoint*2 - line.point1	// specified by the programmer.

### 3.5.7 Summary

In this section, the constraint systems discussed in the previous sections and their aspects are summarized in the following table:

	Alien	Equate	GICS	Kaleidoscope	Procol	ThingLab I & ThingLab II
direction of the relations	adirectional	directional	adirectional	adirectional & directional	directional	adirectional which results in a set of directional relations
cycles allowed	no	no	yes	yes	yes	yes (I) / no (II)
priorities	yes	no	no	5 levels	no	yes
cardinality	1-m	m-m	m-m	1-1, 1-m & m-m	1-m	m-m
dynamic abilities	yes	yes	yes	no	no	yes
# constraint networks	1	2	1	1	1	1
propagation technique	incremental/ sequential/ refinement	incremental/ sequential/ refinement	incremental/ sequential/ refinement	incremental/ sequential/ refinement	incremental/ parallel/ refinement	incremental/ sequential/ refinement
satisfaction technique	local prop.	graph-rewriting	local prop. & relaxation	internal & prototyping	local prop.	local prop., dgrs of freedom, relaxation & redundant views

From this table, it can be concluded that the existing constraint systems for object oriented systems mainly use local propagation as satisfaction technique and mainly use an incremental and sequential propagation technique based on refinement. This is due to the fact that these constraint systems operate within an object oriented environment and changes to properties of constraints are changes to variables of objects; because an object is a black box, changes to these variables can result in (unpredictable) changes of other variables which may also be constrained. This problem can only be solved by using the incremental/refinement approach.

As for the other aspects, some of the systems use adirectional constraint relations, some support cycles and most systems support m-m cardinality and provide dynamic abilities. However, every system has a different mix of aspects and none of them does combine the best of all the different aspects. The MADE constraint will try to combine the best of these aspects into one system. The next chapter will give a detail overview of this MADE constraint system and discuss why certain design decisions were taken.

# CHAPTER 4

## The MADE Constraint System



In the previous chapter, several constraint systems and satisfaction techniques were explained. In this chapter, the focus is on the MADE constraint system. The MADE package itself provides application programmers with a set of object-oriented toolkits (object libraries for, among other things, 2D and 3D graphics, still images, multifonts formatted text, audio and voice sequences, video sequences, animation objects and interaction objects) and utilities (multimedia object editors, a user interface editor, utilities for authoring and help tools) in order to develop interactive multimedia applications. The main objectives of the MADE package are to reduce the multimedia application development by providing high level toolkits and utilities, to impose a better application architecture by using object-oriented technology, to allow a better application integration by providing communication mechanisms between applications and between objects along with a common ergonomics and to guarantee the MADE application portability because the MADE environment itself is portable and hides the platform. The MADE constraint system is one of the toolkits of the MADE package. In the text below both, the conceptual (§4.3) and the implementation level (§4.4) of the MADE constraint system, will be discussed. In §4.5 the performance of the constraint system will be examined. First, however, the MADE environment and the mC++ programming language will be described. Neither the MADE environment nor the mC++ programming language are output of the research that is described in the thesis but they were the given environment in which the research had to be done. The following description of the MADE environment is only here to help the reader get a better understanding of the MADE constraint system when reading the subsequent sections.

### 4.1 The Rationales for Creating the MADE Environment

One of the most spectacular developments in computing technology of the past few years is the appearance of multimedia. Glossy multimedia applications are shown all over the place and on a wide range of different platforms. All major workstation hardware vendors feel the need to come to technical fairs with stunning demonstrations mixing graphics, video, imaging and sound.

However, if we look behind the shiny facades of these applications, we often get a confusing and somewhat disappointing impression. The fact is that most of these programs rely very heavily on the underlying hardware, they are very often highly non-portable, and they lack a unifying programming concept. It is only after several years of development that multimedia research can enter into a more mature phase. Appropriate programming concepts and models must be found and fleshed out in order to have more portable multimedia applications. On a longer term, suitable official and/or de facto standards need to be developed and adopted by the programming community. This fact has been recognized by ISO and work on multimedia document standards has been under way for some time; the SGML extensions known as HyTime ([Newcomb et al. 91]) and the ODA extensions ([Hoepner 92]) are concerned with both multimedia and hypermedia document descriptions. Note that neither of these standards activities concentrate on presentation mechanisms.

The European Communities' ESPRIT III project MADE has set up the ambitious goals of defining and implementing a portable object-oriented development environment for multimedia applications. The particularity of this project is that it does not aim only at the definition of programming tools to handle, e.g., video and audio, but it also identifies a number of more general objects (i.e. tools) that are necessary to develop advanced multimedia applications. One of the important challenges in this project, therefore, has been the definition of a (general) programming model which will allow for the proper use of all the different functional capabilities that should be supported by MADE. A new model had to be defined as models promoted by well-known tools like, e.g., C++, simply did not fully satisfy the requirements of multimedia programming.

Advanced multimedia programming raises a number of non-trivial problems. Apart from the obvious hardware challenges (see, e.g., [Fox 92]), certain demands of multimedia are also difficult to honor within the framework of traditional operating systems, e.g., Unix<sup>TM</sup>. The concerns of the

MADE project are different. Namely, provided that the underlying hardware and operating system layers are appropriate, what is the ideal programming framework, i.e., what is suitable 'mental model' for multimedia programming? In the MADE model there are two central concepts: *active objects* and *delegation*.

### 4.1.1 Active Objects in the MADE Model

Synchronization of different media is known to be a particular demanding task. Conceptually, different media (i.e., a video sequence and a corresponding sound track) should be considered as parallel activities that have to reach specific milestones at distinct and possibly definable synchronization points. In many cases, specific media types may be directly supported in hardware. In some cases, using strictly specified synchronization schemes, the underlying hardware can take care of synchronization. However, a general object model aimed at multimedia should offer the possibility to describe synchronization in general terms as well. This requirement directly leads to the adoption of *active objects* as one of the cardinal concepts in the MADE model.

The definition of active objects in MADE is fairly traditional. The advent of popular 'object-oriented' paradigms like C++, often makes it necessary to remind their users that one of the archetypes of object-oriented programming, Smalltalk-80 ([Goldberg 84]), makes use, at least conceptually, of active objects. The definition of the active objects in MADE follows the same ideas, using paradigms for message passing and for the control of received messages that are well known in the concurrent programming paradigm.

The only unusual feature in the MADE model, compared to traditional message passing protocols, is the introduction of *sampled messages*. Consider the well-known idea of sampling a logical input device, e.g., mouse position values. A separate object modelling (or directly interfacing) a mouse can send thousands of motion notification messages to a receiver object, and this latter can just 'sample' these messages using the sampled message facility. More generally, the introduction of sampled messages makes it possible to model (using small, dedicated active objects) different sorts of interrupt handling, which is often necessary in multimedia programming.

The introduction of active objects gives an efficient and elegant solution to the problems raised by media synchronization. Synchronization of different media can be expressed in terms of reference points within each media type (reference points might be video frames, audio samples, etc), and these can be readily modelled using the notion of event synchronization ([Guimarães et al. 92a], [Guimarães et al. 92b]). Using active objects, event synchronization appears to be no more and no less than synchronization of concurrent processes, i.e., concurrent active object in the case of the MADE model.

The idea of using active objects for multimedia programming is, *per se*, not new. Indeed, a group at the University of Geneva has used the same notion for their multimedia research ([Gibbs et al. 92], [May et al. 92]). However, their approach to active objects is more restrictive than the one used in the MADE model. They essentially define source, filter and sink objects only, and build their applications based on these concepts alone. Although these simple active objects can be described easily in the MADE model, there is no reason to restrict active objects in this way. On the contrary, a richer functionality leads to application and programming facilities that cannot really be achieved by other means.

### 4.1.2 Delegation in the MADE Model

Interaction among multimedia objects is inherently very dynamic. Logical links are created among objects, but these links are temporary in nature. One of the most important and, at the same time,

most complex examples for these links is constraint management. Constraints may be set up among objects, which then affect their behavior. These constraints are dynamic: a user may set them up and release them at a later point.

Ideally, setting up and managing constraints should be 'internal'. This means that if a third party object communicates with constrained objects, this third party object should *not* have to know whether or not the object it is communicating with is subject to a constraint. This requirement is reminiscent of classical inheritance in traditional object-oriented systems: a third party object does not know whether a specific message it sends is served by a method of its direct target, or by a method that this target object inherits from another object. There is a major difference, though: inheritance is static (i.e. it is defined when the type of the object is defined) and what we need for interactive multimedia applications is a dynamic capability. Hence the second major concept in the MADE model is *delegation*.

The precise semantics of delegation is based on the concept of *prototypes* as opposed to classes. A prototype is an object that represents the *default* behavior for a set of similar objects. Any object can play the role of a prototype. New object can re-use part of the knowledge stored in a prototype. To create an object that shares knowledge with a prototype, one has to construct an *extension* object, which has a list of personal behavior idiosyncratic to the object itself, and a list of prototypes. The list of prototypes of an object can change during its lifetime. When an extension object receives a message, it first attempts to respond to the message using the behavior stored in its personal part. If this is not possible, it has to find a prototype stored in the extension which would respond to the message. This last step is the *delegation* of the message. The exact semantics of delegation is described elsewhere (and beyond the scope of this thesis). Classical *inheritance* is a closely related concept of that of delegation; it is simply a 'static' form of delegation. This means that an object *A* may be defined to *inherit* the behavior of object *B*, which is conceptually equivalent to the fact that *A* delegates its behavior to *B* once and for all.

The concept of delegation, although not really widely known and used, is not a new concept ([Borning 86], [Lieberman 86]). It has also been proposed for use with graphical systems and in animation. It has, however, never become widely accepted by the computer graphics community. The only example of its usage within the context of multimedia (that the author is aware of), is for the Event Script language of the Athena Muse project ([Hodges et al. 89])<sup>1</sup>. This, in spite of the fact that delegation is, in some sense, a more powerful concept than the concept of inheritance and opens up modelling possibilities which are not cleanly describable by other means. Delegation in combination with active objects leads to other potential benefits. Concurrency can be easily exploited via delegation between active objects (for example, an object might clone itself and delegate part of its behavior to the clone) allowing both objects to proceed concurrently. A caller of the original object need not to be aware of the exploitation of concurrency in this case and is not impeded by an redirection of the message since delegation always takes the shortest route between caller and delegatee.

## 4.2 The mC++ Programming Language

The MADE Constraint System is written using the mC++ programming language. This language is based on C++ but provides a rich assortment of additional features which may help the programmer to create multimedia presentations more easily. The mC++ programming language is

---

<sup>1</sup> Event Script is, however, *not* meant to be a general-purpose programming environment; it is used primarily for creating user-interface event handlers.

part of the MADE environment and is an implementation of the MADE model as described in the previous section.

To be able to obtain a better understanding of the implementation of the constraint system, the relevant concepts of the mC++ programming language will be explained. Some of the design decisions taken for the constraint system as well as the reason why some functions in the constraint system are modelled the way they are find their origin in the MADE object model. In all these cases, communication, delegation and object classes as defined in the MADE object model have had their influences. For an elaborated discussion on mC++ and the underlying MADE object model the reader is referred to [Arbab et al. 93a], [Arbab et al. 93b], [Heeman et al. 93], [Herman et al. 94].

### 4.2.1 MADE Objects and Communication

Objects are defined as instances of class types. In mC++ class types can be defined as either *active* or *passive*. Instances of active class types are said to be active objects whereas instance of passive class types are said to be passive objects. Every active object has its own virtual processor and its own thread of control<sup>2</sup>. In practice, threads can all run on one processor or they can be distributed over several processors. Because every active object has its own thread of control, every active object is able to control its own behavior and when and how it communicates with other objects. Passive objects have no own thread of control. Therefore, they have no control over when to communicate with other objects. They do have some control over how to communicate with other objects. In both cases, communication between two objects, has to take place via the same, shared memory.

#### 4.2.1.1 Communication

Communication with active objects takes place by *message passing*. From the outside, messages can be sent to an active object. Internally, such an object has two different message regimes. Each message, which can be accepted by the object, has its own dedicated *message receptor*. This message receptor defines whether a message is "queued" or "sampled". Whether a message will be queued or sampled is internal to the receiving object.

Queued messages are placed in the receptors' queue until the message can be serviced. Until then, the object that has sent the message will be suspended. Messages in the queue will be served "fairly" so that no message is indefinitely denied a chance for service. Queued messages may have parameters and they may return a result to the sender. Queued messages in mC++ are what is considered in the literature as synchronous communication.

Sampled messages are not queued. A message receptor for sampled messages can only contain one message. A newly arriving message replaces any current message in the receptor. The object that sent the message will not be suspended. As soon as it has sent its message, it will continue. This implies that a sampled message cannot return a result to the sender. They may still have some parameters associated with the message. Sampled messages are thus a special form of what is considered to be asynchronous communication. The actual servicing of a sampled message is postponed until the active object is able to do so.

---

<sup>2</sup> In the current implementation of the Made Object Model on the IRIX 5.3 (SGI) systems, only 40 threads are available. On SunOS4.x and SunOS5.x (SUN) there is no limit on the number of threads.

Message receptors can be assigned a priority. If different message receptors have pending messages and the object can service a new message, the object will choose to accept the message with the highest priority from the message receptor. Choices between messages of receptors with the same priority is made nondeterministic. When the active object services a new message it will invoke a member function with the same name as the message<sup>3</sup>.

Communication with passive objects is done via the invocation of the member functions of these objects and the values of the parameters passed during these invocations.

### 4.2.1.2 Prototypes and Delegation

In mC++, both the notions of prototypes and delegation ([Borning 86], [Lieberman 86]) are supported. For each class type which is defined in a mC++ program, there exists a prototype type and one instance of this prototype type (the *prototype*). This prototype is always present, regardless of whether there exists an instance of the class type. Prototype types of active class types are active as well and prototype types of passive class types are passive. MADE objects can use the member functions defined in their prototypes<sup>4</sup>; the prototype defines the *default* behavior of a set of similar objects. A MADE class type represents an *extension* of the prototype type, in that methods may be defined which are not defined in the prototype type and are particular to this class type.

Closely related to the notion of prototype is delegation. Whereas prototypes can thus be used to define the common behavior of all instances of a particular class type, delegation can be used to specify different behavior of specific instances of a class type. In mC++, all objects (active and passive) have the possibility to dynamically designate (part of) their behavior to other objects. To do so, an active object has to *delegate* all messages for a certain message receptor to the message receptor of another object. The object which delegates its messages as well as the object which sends a message that is delegated will, after the messages are delegated, not be aware of this. The object to which the messages are delegated are able to determine whether the messages were delegated and, if so, which object has delegated the messages. This information may be used to communicate with the delegator object.

A passive object will delegate a member function invocation to the invocation of a member function of another object. Note that active objects can only delegate their messages to other active objects, whereas passive objects can delegate invocations of their member functions to passive and active objects. In the latter case, the member function invocation is converted into a message call. Delegation is only allowed for public and protected member functions.

### 4.2.2 MADE Object Classes

Besides the normal C++ class, mC++ defines three additional (MADE) classes. One of these classes is an active class. The other two classes are passive classes. With the exception of the normal C++ classes, all other classes can use delegation and may have prototypes. Communication with ordinary C++ classes is done using normal member function invocation.

---

<sup>3</sup> In the remainder of this text the notions of message and member function (with respect to an active object) will be considered to be synonyms.

<sup>4</sup> The prototypes of an object are the instances of the prototype types of all the super-class types of the object.



### 4.2.2.1 Active Objects

Active objects run in their own thread of control. They have control over which messages to accept and when to accept them. The general syntax of an active object declaration is as follows (*T* stands for an arbitrary type and *ARGS* for a number of arbitrary arguments):

```
Active AObject {  
  
    // Class Prototype declarations  
    Proto private:  
        // private data and member functions  
        T proto_priv_mf (ARGS);  
    ...  
    Proto protected:  
        // protected data and member functions  
        T proto_prot_mf (ARGS);  
    ...  
    Proto public:  
        // public data and member functions  
        T proto_publ_mf (ARGS);  
    ...  
  
    // Class Extension declarations  
    private:  
        // private data and member functions  
        T priv_mf (ARGS);  
    ...  
    protected:  
        // protected data and member functions  
        T prot_mf (ARGS);  
    ...  
    virtual void main (); // must be defined  
    public:  
        // public data and member functions  
        T publ_mf (ARGS);  
    ...  
  
    Sampler:  
        publ_mf, prot_mf;  
    Priority:  
        (publ_mf, prot_mf), (priv_mf);  
    Delegate_access:  
        prot_mf;  
};
```

The *Proto public*, *Proto protected* and *Proto private* sections are optional. In these sections, the member functions of the prototype type are defined.

The optional *Sampler* section is comma-separated list of message names which describes which messages will be sampled by the message receptor.

The optional *Priority* section is a comma-separated list of groups of message names. The priority section describes the relative order of the different messages. The groups are listed in decreasing priority. Messages within the same brackets have equal priority.

The optional *Delegate\_access* section describes the protected member functions which may be accessed by a delegatee (i.e. an object to which a message is delegated). This comma-separated

list contains only the names of protected member functions; public member functions are already accessible by default.

#### 4.2.2.2 Mutex Objects

Mutex objects are passive objects. They provide mutual exclusive servicing of their public member functions. On protected and private member function no mutual exclusion is provided.

```

Mutex MObject {

  // Class Prototype declarations
  ...

  // Class Extension declarations
  ...

  Sync:
    ( boolean condition ) : publ_mf;
    ( boolean condition ) : publ_mf;
  Priority:
    (publ_mf, prot_mf), (priv_mf);
  Delegate_access:
    prot_mf;
};

```

In the optional *Sync* section a finer control over the execution of the member functions is provided. For each public member function a condition can be specified which is evaluated every time a member function invocation can be serviced. A member function invocation can be serviced when no other member function is executed at that time. After this evaluation, the object will choose among the pending member function calls which conditions have evaluated to the value 'true'. This choice is based upon the priority scheme.

A mutex object may also define a *priority* section and a *Delegate\_access* section similar to the ones for active objects.

#### 4.2.2.3 Unprotected Objects

Unprotected objects provide no mutual exclusive access (and thus are unprotected against concurrent access) on their member functions and are normal C++ objects with a few extra features; they can delegated their member functions to other mC++ objects and other mC++ objects can delegate their member functions to unprotected objects. Unprotected class types also have prototype types.

Unprotected objects may also have a *Delegate\_access* section.

#### 4.2.2.4 Thread Control and Message Servicing

In case active objects communicate with other active objects or when passive object communicate with active objects, mC++ uses a 'rendez-vous' mechanism and syntax which is a simplified version of concurrent C and is inspired by the 'rendez-vous' mechanisms of Ada ([Gehani et al. 90], [Naiditch 95]).

Whenever a message is sent to an active object, the message is serviced in the thread of the receiving active object; an active object controls the servicing of incoming messages itself. The

sending object (either active or passive) is suspended until the receiving active object has finished servicing the message (in case of a sampled message, the receiving object will service the message immediately and the sending object will not be suspended at all).

In case an active object communicates with a mutex or an unprotected object, the member function of that mutex (unprotected) object is executed in the thread of the calling active object. More generally, the member functions of mutex and unprotected objects are always executed in the thread of their caller. Thus if an active object **A1** invokes a member function of a passive object **P1** and if, during the execution of that member function, a member function is invoked of the passive object **P2**, the member function of **P2** is serviced in the thread of object **A1**, because the member function of **P1** was also executed in the thread of object **A1**. If, in its turn, object **P2** sends a message to an active object **A2**, the message is serviced in the thread of object **A2**.

Special rules apply in case of the mutual exclusion on the execution of member functions of mutex objects. The mutual exclusion property only holds for different threads. That means that when an active object **A** has invoked a member function of mutex object **M**, no object which runs in another thread than the one of **A** can invoke a member function of **M** at that moment (it will be suspended). However, when, during the execution of the member function of **M**, a member function of another passive object **P** is invoked, this object **P** can invoke (again) member functions of **M** because **P** runs in the thread of **A** due to which no mutual exclusion on the member functions of **M** is provided for **P**.

### 4.2.3 mC++ Runtime Methods

Besides the three additional class types, mC++ also provides some extra functionality. Some of these extra methods are important for the implementation of the MADE constraint system. Only this functionality will be discussed here. The two important parts are the Dynamic Call Interface and the Delegation Interface. There are also some general methods which are of importance to the MADE constraint system to maintain its administration of its constraints.

#### 4.2.3.1 Dynamic Call Interface

The Dynamic Call Interface (DCI) makes it possible to invoke member functions of mC++ objects. What sets the DCI apart is that it can be determined dynamically which member function of which object to invoke. Member functions can be identified by their name and by the type of their arguments. The advantage of the DCI lies in the fact that in this way the choice of how the communication between objects takes place can be delayed until runtime. This means that in mC++ objects can be created which can adapt their behavior (with respect to communication) at runtime. The DCI is only available for public and protected member functions.

The DCI supports member functions which have parameters and the DCI is also capable of returning the result of the member function. A very powerful feature in this is the ability to check whether a certain object has a certain member function defined in its interface. This does not mean that an active object will accept the message or that the member function of a passive object can be executed at that moment (due to mutual exclusion, callers may be suspended), but only that there exists a message receptor for this message or that entry is defined for this member function in the interface of the object.

◦ **void MCallMethod (MObject\* destObj, char\* idString, ... /\* possible parameters \*/)**

This function invokes the member function *idString* of object *destObj*. Additional parameters may be specified in case the member function takes arguments. The first additional parameters is used to store the result (if needed) of the member function. Therefore this

argument should be a pointer to a variable which is large enough to contain the result. The remaining additional parameters are the arguments of the member function.

◦ **MBool MinqMethod (MObject\* destObj, char\* idString)**

This method returns MTRUE when the object *destObj* has defined the member function *idString* in its interface. Otherwise, this function returns MFALSE.

```
void EXPORT registerDependentObject (MObject* d, CO* c, char* dtem, char* dtrm) {
...
if (dtem != 0) {
    int p1, p2;
    p1 = strcspn (dtem, " ");
    p2 = strcspn (dtem, "(");
    char* strg = (char*) MMALLOC (strlen(dtem)+30);
    strcpy (strg, "void setInstanceClass_");
    strncpy (strg+22, dtem+p1+1, p2-p1-1);
    strcpy (strg+22+p2-p1-1, "(void*)");
    MCallMethod ((c->prototypeFromName(c->nameOfMCclass())), strg, c);
    MFREE (strg);
}
else {
...
};
...
};
```

#### 4.2.3.2 Delegation Interface

In mC++, a few additional features are included in the Delegation Interface which expand the functionality of the delegation mechanism. These features make it possible to dynamically extract information about the delegation. This means that a function can obtain information about the member function that was delegated and the object that has delegated it.

◦ **void MObject::delegate (char\* idStrDlgr, MObject\* dlgte, char\* idStrDlgte, MBOOL status)**

This function allows to set or cancel a delegation. This is done by changing the value for *status*. In case this parameter has the value MTRUE, the delegation of member function *idStrDlgr* of the current object is delegated to member function *idStrDlgte* of object *dlgte*. The signature of *idStrDlgr* and *idStrDlgte* have to be the same. In case the value of *status* is MFALSE, the delegation is canceled.

◦ **Delegator**

Every function has the possibility to inspect the variable *Delegator*. If the current function is invoked due to a delegation (i.e. the function is the destination of a delegation), the value of *Delegator* points to the object which has delegated one of its member functions to the current function. This variable can be used, together with the *Delegator\_access* entry of a class declaration, to access (protected) member functions of the delegator. If the current function is invoked, but not through a delegation, *Delegator* is equivalent to the *this* pointer.

◦ **Delegator\_function**

The variable *Delegator\_function* points to the function which has been delegated to the current function. This variable can be used to access the original delegated member function.

◦ **Originator::**

This object points to the same object as *Delegator*. However, *Originator* is especially useful in combination with *Delegator\_function* in a function which was the destination of a delegation action; when the statement *Delegator->Delegator\_function (...)* is used, the current function is invoked due to the active delegation of member function

*Delegator\_function* of object *Delegator* to the current function. If, instead, *Originator::Delegator\_function (...)* is used, the last specified delegation of *Delegator\_function* is ignored. Because of this, the recursive invocation of the current function is circumvented. This same mechanism is applied to all functions *f* when called by *Originator::f*.

```
Proto int constraint_object::dep_shadow_func_getClockValue () {  
    char* m = ((constraint_object *) InstanceClass_getClockValue()->name_dep_shadow_func_getClockValue());  
    MCGO->dep_dispatcher (Delegator, m);  
    int t = Originator::Delegator_function parameters;  
    return t;  
};
```

### 4.2.3.3 General Messages

In the mC++ programming language all MADE classes support some general member functions that allow other objects to obtain information about prototypes and class hierarchies. These , however, will not be discussed here and the reader is referred to the MADE literature for more details.

## 4.3 The Concepts of the MADE Constraint System

In this section, the concepts of the MADE constraint system will be explained. For that purpose, in §4.3.1 the specific requirements for constraint systems in multimedia environments are reviewed. Out of this review, the different design decisions for the MADE constraint system are deduced and consequently discussed in §4.3.2. In §4.3.3 the naming conventions as adapted in the MADE constraint system as well as the symbols and notations used in the remainder of this text are discussed.

### 4.3.1 Constraints in Multimedia Environments

In this section, the relevance of the different types of constraint systems to multimedia applications is examined. Multimedia applications try to define, manipulate and present spatial and temporal information. To do so these applications generally have to deal with two notions of information: *multimedia data* and *multimedia information* [Bulterman 94].

- Multimedia data consists of raw media chunks (physical entities) and defines what is to be presented. Examples of multimedia data are graphics (gif, jpeg, tiff), movies (flic and mpeg), sound-samples (voc, wav, snd), graphic output primitives (line, circle, arc), text, etc.
- Multimedia information defines the context in which the multimedia data has to be used (logical entities). Examples of multimedia information are the location on the screen, the sample rate, the order in which multimedia data is presented, etc.

Constraints in multimedia are mainly used to support the maintenance of the multimedia information and to keep this information consistent throughout the presentation. In most multimedia applications, the multimedia data is a given entity whereas the multimedia information is used to glue the multimedia data together in a smooth and pleasant way. In the remainder of this section we will focus on the constraint management of multimedia information.

Because a lot of different disciplines come together in multimedia (graphics, synchronization, animation, collision detection, interface design, etc.), a lot of different types of constraints may be used in a multimedia presentation. Instead of creating different constraint systems for every area, one constraint system should be defined which can be used by the multimedia systems. This constraint system should be able to support the constraints for each of these different disciplines. This implies that on the one hand the constraint system should be a general constraint system so it can satisfy all kinds of constraints, but that on the other hand it is also powerful enough to satisfy all the different constraints needed in a multimedia presentation. When the presentation is based on interaction with the user, even more is demanded from the capacities of the constraint system; information propagation and constraint satisfaction are then dependent on the actions of the user and may thus change whenever the user takes some actions. As seen in Chapter 3 most of the current constraint systems are designed to solve problems in a very narrow area; consequently these systems cannot be used in multimedia environments and a new approach for multimedia constraint systems is necessary.

Multimedia is thus an area in which many different kinds of objects need to be managed simultaneously and where the maintenance of the relations between all those objects can become a complex and difficult task. Consequently,

- it is more advantageous if the multimedia constraint system runs in parallel with the ongoing presentation so that the presentation as a whole is disturbed as little as possible due to actions of the constraint system. This will ensure a presentation of the multimedia data that is as smooth as possible.
- it is desirable if the constraints which work on different parts of the presentation are solved in parallel with each other so that the sub-presentations are disturbed as little as possible due to ongoing satisfaction processes in other parts of the presentation. As different parts of the presentation may run in parallel with each other, the constraint system should disturb this parallelism as little as possible and satisfy the constraints per part of the presentation in parallel.
- it is necessary for a constraint system to be able to interactively change the status of the constraints in the constraint network. Multimedia presentations are often interactive and the actions of the user may have their effect on the status of the different constraints; which constraints are needed, when are they needed and for how long are they needed.
- it must be possible to change the constraint network dynamically because of the same reason as in the previous item. The interaction with the user may lead to a situation where new constraints or new dependency relations are in order (or can be removed as they are no longer needed). Therefore, it must be possible to add/remove objects to/from the constraint network.
- it would be helpful if constraints could be specified in the most declarative way possible. A declarative description lies closer to the notion of constraints than an imperative description. Therefore the possibility to use declarative descriptions may reduce the complexity of describing the constraints for the already complex network of relations. It should be the task of the constraint system to translate these declarative constraints into the equivalent imperative instructions for the computer.

Another point which may increase the complexity of constraint maintenance is that there may be very complicated (indirect) relations between the different objects in a multimedia application. These relations may have priorities, be cyclic, be directional or adirectional, have arbitrary cardinality or combine some or all of these qualities:

- The notion of priorities in multimedia constraint systems is essential. Within a multimedia presentation, the different objects in the presentation are always competing with each other to get the best resources and to be displayed in the best possible way. As computers are (at the moment) not powerful enough to provide all the required



resources at the same time to realize this perfect presentation, these resources need to be shared between the different objects. An elegant way to do this, is to assign priorities to the requests for resources. A multimedia constraint system must be able to support, among other things, this competition and thus should be able to support the priority assignment if necessary. The precise semantics of the priority systems (what to do in case of equal priorities, how to avoid deadlocks or starvation due to long high priority computations) is not relevant at this stage and will therefore not be discussed in detail.

- Cyclic dependencies between the objects in multimedia applications are easily created. Very simple and common applications already require the support of cyclic dependencies. An example where cyclic dependencies are used is the case of the multiple views (a particular piece of information is presented in a number of different ways). Changing an arbitrary view (i.e. changing the actual information which is presented) may affect the other views.
- Much of the expressive power of a constraint system is determined by the way in which changes to one object can be propagated to other objects.
  - A multimedia constraint system should be able to support directional dependencies as well as adirectional dependencies. An example in which directional dependencies are desired is in the case where a grid needs to be modeled. The position of the objects depends on the grid size but the grid size should not be dependent on the location of the various objects. The multiple view example is an example where dependencies should be adirectional; changing the information in an arbitrary view causes other views to be adapted accordingly and vice versa.
  - The constraint system should also support dependencies with cardinality m-m. The multiple view is, as is suggested by its name, an example of a 1-m dependency relation: the same piece of information is presented in a number of ways. However, more general multiple view examples will require the more stronger m-m cardinality as it is not uncommon that the information which is to be presented is made out of information collected from several objects.

Also an important point is that it must be possible to store multimedia presentations (i.e. the multimedia information) in some kind of database. In that case, a certain presentation can be saved and retrieved from the database later so it is possible to do a replay, store a history of the different phases of the presentation or to allow for a pause-and-resume like operation. As a result of that, it must be possible to store the status of the different constraint objects and the constraint network itself to a database as they are an integral part of the presentation.

From the observations outlined above and from the information presented in the previous chapter, the following evaluation of the different satisfaction methods for multimedia applications can be made:

- The method known as **Propagation of Known States**, in contrast to the other methods, can solve constraints in a parallel environment. Because multimedia applications often use parallel execution schemes, propagation of known states seems a logical choice. Propagation of known states is very well suited to interact with the user of a multimedia application on the different actions to be taken by the constraint system. Changes to objects due to constraint satisfaction occur one by one and the user can easily follow the propagation (and changes of the different objects) within the constraint network.
- **Propagation of Degrees of Freedom** is a method which requires the constraint network and the constraints in it not to be altered during constraint satisfaction. When a constraint is satisfied, no other constraints may be added, deleted or (de)activated in the constraint network. This restriction means that this method can hardly be used in a parallel

environment. The main application would have to be stopped during constraint satisfaction. In multimedia applications this would mean that first the presentation is stopped, then the constraints are satisfied and finally the presentation is resumed. Such an approach would lead to very jerky presentations.

Another drawback of this method is that interaction would be more problematic as whole subgraphs are satisfied at once. This would mean that the user cannot get a clear insight in what is going on in the constraint network (if such an insight is desired).

- The method of **Relaxation** requires its constraints to be numerical. However, constraints in multimedia applications can be applied to all kinds of objects and the demand to force all multimedia constraints to be translated to numerical dependencies would restrict a multimedia constraint system too much. Relaxation tries to change the current situation (in which some constraints are invalidated) smoothly into a situation where all constraints are satisfied (which, hopefully, leads to a smooth adaptation of the presentation). However, relaxation is a time-consuming technique to obtain constraint satisfaction and this would leave the presentation's constraints in an invalid situation for too long.
- **Redundant Views** can be used very well in a multimedia constraint system. They may even help the constraint system find solutions for complex problems. However, it is the programmer who has to provide the multiple views. The constraint system cannot, in general, deduce redundant views itself (this would lie in the scope of AI).
- **Prototyping** can also be used in multimedia constraint maintenance. However, prototypes only provide support for simple constraints; all objects generated from a particular prototype can share the same behavior. It can mainly be used for some kinds of equivalence constraints (variables having the same value, functions having the same functionality, etc.).
- **Graph and/or Term Rewriting** may be used by a constraint system in a multimedia environment. However, rewriting, like relaxation, is a very time-consuming technique.

Considering the above mentioned requirements and observations, it seems that an incremental approach to constraint maintenance is probably the only technique which can, in real life, deal with the problems of constraints in multimedia applications. Trying to solve the whole constraint network all at once may be too time-consuming or even impossible.

The requirements for the constraint system as presented in this section were inspired by the fact that the system should operate in a multimedia environment. However, because the multimedia constraint system should be so versatile, the resulting system would be applicable in many other areas too.

#### 4.3.2 Design Decisions in the MADE Constraint System

The MADE constraint system was designed with the remarks, made in the previous section, in mind. Therefore, most of these aspects can be found in the constraint system. However, some of them have been realized differently in the MADE constraint system than suggested earlier.

In the constraint system two levels can be distinguished; one level defining the static structure and one level defining the dynamic structure of the constraint system. The static structure is defined by a collection of constraint objects that specify how a certain constraint problem should be solved. The dynamic structure is defined by the relations between the different properties and the constraints defined in the static structure.

Using another view, the global MADE constraint system can be divided into six parts (CUI, MCGO, CGO, CO, CO\_WRAP, ROUTER). This division is a result of the requirements imposed on multimedia constraint systems and the way in which two of the main aspects of the MADE constraint



system are realized; parallel constraint satisfaction and flexibility. CUI, MCGO, CGO and ROUTER are used to manage the dynamic structure whereas CO and CO\_WRAP manage the static level of the constraint system.

- CUI: the access to the MADE constraint system is provided by the *Constraint User Interface*. All constraint related operations which are issued from an application program should go through the CUI. The CUI will redirect the requested operation to the appropriate MCGO object and/or ROUTER object.
- MCGO: the *Meta Constraint Graph Object*. This part controls and coordinates all the operations which may affect (part of) a constraint network. In theory, several constraint networks may coexist and thus several MCGO objects could coexist. In the MADE system there is only one constraint network and thus only one MCGO object.
- CGO: the actual constraint network may be divided into several independent clusters. A *Constraint Graph Object* manages one such cluster. In the MADE system there may be several CGO objects present; one for each cluster.
- CO: the *Constraint Object* provides access to the individual constraints. Although the CO objects are part of the constraint system, they can be accessed without using the CUI. This may be the case when the status information of the CO objects has to be changed or retrieved.
- ROUTER: the *ROUTER* object stores information about the links defined between constrained and constraint objects in the constraint network. There exists one ROUTER object for each MCGO object.
- CO\_WRAP: the *Constraint Object WRAPper* makes it possible to solve the different constraint objects in parallel.

In the next sections, each of the different parts is discussed in more detail.

### 4.3.2.1 The Constraint User Interface (CUI)

The sole purpose of the CUI is to provide the application programmer with a clean and easy to use interface to the constraint toolkit. To do so, the CUI hides most of the details of how the constraint network is implemented. The CUI's most important task is to redirect the function calls made to the CUI to the appropriate MCGO object.

The CUI accepts function calls to build, destroy and save (part of) the constraint network. These calls can be made at any time. This means that the constraint network can be altered dynamically; constraints can be added and deleted at any moment in time. Of every configuration of the network, an image can be saved in the database. Furthermore, a constraint network can be retrieved from the database at any moment. This functionality allows for interactive and dynamic constraint management.

### 4.3.2.2 The Meta Constraint Graph Object (MCGO)

The main function of the Meta Constraint Graph Object is the management of the different CGO objects. Also the redirection of function calls is an important task; function calls made to the MCGO object are redirected to CGO objects. The reason to have another redirection layer below the CUI is threefold:

- 1) the MCGO only deals with the (de)construction of the network itself whereas the CUI provides an interface to both, the network (MCGO objects) and the information about the links in the network (ROUTER objects).

- 2) everything managed by one MCGO can be considered a separate constraint network whereas the CUI may connect to several MCGO objects and thus to several different constraint networks.
- 3) the MCGO is used as (lower level) entry point to the constraint system by the CO objects.

The reason to put an extra layer on top of CGO objects had to do with flexibility. It is possible that within a single constraint network, a particular cluster of objects has no direct or indirect connections to another cluster of objects. As they are not connected, the maintenance of each of these clusters is totally independent of each other. Therefore, their maintenance can be split over different managers. As each manager has only one (smaller) cluster to maintain, they may do this more efficient. This is what the CGO objects actually do; each of them maintains a separate cluster of the total constraint network. Whenever a link is created which connects two different clusters, these clusters are grouped under one CGO object. Similar, when a bridge (i.e. a link which is the only (in)direct connection between two groups of objects) in a cluster is removed, the cluster is divided into two new clusters, each one maintained by a different CGO object.

The MCGO object serves as a dispatcher that dispatches the incoming function calls over the existing CGO objects. For that purpose, the MCGO object has to have a list of all the CGO objects in the MADE constraint system. Because the MCGO does not have any knowledge about which constrained objects are located in which cluster, the MCGO object will simply dispatch the incoming function calls to all CGO objects. The CGO objects themselves will then check whether the function call should be serviced by them or not. This approach was motivated by the following observations:

- details w.r.t. constraint satisfaction should be kept as local as possible (and thus also the data needed to perform this constraint satisfaction)
- the implementation of the CGO objects would become simpler as the task of that object would be limited (the CGO objects do not have to deal with parallelism)

The difference between the different clusters managed by CGO objects and the different networks managed by MCGO objects is that two clusters may merge into one new big cluster whereas the constraint networks managed by different MCGO objects can never be merged. Furthermore, the merging and splitting of the different clusters is done automatically by the constraint system whereas the division of the objects over the different MCGO objects has to be done by the application programmer.

Whether constrained and constraint objects should be placed in the same constraint network under one MCGO object or in several constraint networks under multiple MCGO objects depends mostly on the problem that needs to be solved.

The MCGO object does not run parallel with the application program. Function calls which cause the constraint network to be modified will suspend the caller until the modifications to the network are complete. This restriction has been made to avoid situations where a second update of the network causes conflicts in the network because it was not stable yet due to the first update. This typically occurs when clusters are merged or split.

### 4.3.2.3 The Constraint Graph Object (CGO)

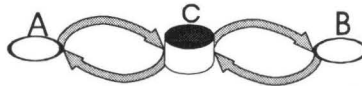
The CGO objects are the objects which perform the actual modifications to the constraint network. Every CGO object maintains a part of the whole constraint network. For flexibility reasons, the complete constraint network is divided into different clusters. Constrained and constraint objects in one cluster have no connections with constrained or constraint objects in other clusters. This property makes it possible to maintain the different clusters independent of each other. This also makes it possible to exploit the concurrent character (i.e. the ability to have separate threads of control) of the MADE programming environment. Each CGO object in the MADE constraint system

runs in its own thread of control (i.e. is an active object) so that updates to the different clusters can be done in parallel.

CGO objects do nothing more than store information about the structure of the cluster and activate CO\_WRAP objects. To prevent inconsistencies in the stored information, access to the information maintained by a particular CGO object may be denied temporarily. For instance, it is not possible to do constraint satisfaction while the structure of that same cluster is updated or vice versa update the structure of a cluster while satisfying some constraints. However, when a certain CGO object has blocked (some of) its information, the information of other CGO objects may still be accessible.

The information stored by the CGO objects is used to provide an incremental satisfaction technique: local propagation of known states. The information is organized in such a way that propagation can be easily done in single steps (from constrained object via the constraint object to another constrained object).

It is allowed to create cycles in a cluster. Cycles are automatically detected by the CGO object when created. It is not possible to create adirectional relations in the CGO object. However, adirectional relations (between two objects **A** and **B**) can be simulated by a cyclic relation. For that purpose two relations should be defined; one between a constrained object **A**, a constraint object **C** and another constrained object **B** and the other between object **B**, constraint object **C** and object **A**. Note that object **A** as well as object **B** both play the role of source and destination for the same constraint object **C**. The MADE constraint system provides several ways to check the direction of the propagation (either from object **A** to object **B** or vice versa from object **B** to object **A**):



#### 4.3.2.4 The Constraint Object WRAPPER (CO\_WRAP)

The CO\_WRAP objects make it possible to satisfy the constraints in parallel with the application. CO\_WRAP objects are active objects which have their own thread of control. The sole purpose of the CO\_WRAP objects is to provide an environment where it is possible to execute the statements needed to satisfy a constraint object in their own (and thus concurrent) thread of control. As the MADE constraint system does not analyze these statements, no additional support is given by the constraint system itself to gear the synchronization of the actions of the different constraint objects to one another. This means that, beside for satisfaction purposes (see §4.4.7), the MADE constraint system does not consider any synchronization aspects between the satisfaction processes of the different constraint objects. If such a synchronization is desired, it can be programmed explicitly by the constraint programmer (see §4.3.3 and §4.4.1.2). The reader is referred to [Bergmans 94] and [Bergmans et al. 96] for more details on this subject; a more detailed discussion of this topic is beyond the scope of this thesis.

#### 4.3.2.5 The Constraint Object Class (CO)

The constraint object class provides the basic functionality which is needed by the constraint system to perform constraint satisfaction. The application programmer can use this class as the basis from which more complex constraint classes can be created. The approach to take a class as the fundamental building blocks for constraints allows for the building of constraint libraries; a new

constraint class can define the constraint functions which have to be used to satisfy a specific problem or a number of problems for a specific domain. Once the constraint class has been defined, its instances can be regarded as declarative solutions to a constraint problem. Because the MADE constraint system is implemented in mC++, the lowest level of the constraint class hierarchy has to be imperative.

Having a class-oriented approach also accommodates different kinds of programmers. Those that only have a basic knowledge about the constraint system and just want to use ready-made constraint objects and those that have enough specialized and detailed knowledge about constraint programming to make constraint classes themselves which are tailored for a particular situation.

One of the most important parts of a constraint class are the constraint functions. Constraint functions are member functions of the CO object which contain a list of statements which describe the actions which have to be taken to satisfy a constraint object. The constraint object class provides special constructs to assist the constraint programmer with the writing of constraint functions. These constructs allow the programmer to get a finer control over cyclic dependencies and to use priorities.

### 4.3.2.6 The Router

The Router contains information about the links between the different constrained objects on the one hand and the constraint objects on the other hand. This information is important for the constraint system as, among other reasons, it provides the possibility to work with constraint priorities and to support m-m cardinality between (in)dependent objects and constraint objects.

### 4.3.3 Naming Conventions, Symbols and Notation

To make a discussion about the MADE constraint system easier to understand, different parts of the constraint network, as used in the MADE constraint system, have been given certain names. This section will introduce these naming conventions. Furthermore, to illustrate more clearly the ideas behind the constraint network, graphical symbols will be used to visualize what is said in the text. These symbols will be introduced and explained in this section too.

The MADE Constraint System makes use of one constraint network. Construable objects in MADE can be added to the constraint network in two ways: as *dependent objects* or as *independent objects*. Independent objects are objects which may *trigger* a constraint object; i.e. they may indicate to a specific constraint object that the constraint maintained by that constraint object may need to be re-satisfied because changes to a property<sup>5</sup> of the independent object may have an impact on the properties of other objects. These latter objects are called the dependent objects. The names dependent and independent are relative to a specific constraint object; an independent object of one constraint object can be a dependent object for another constraint object. In this way, a whole network of dependent, independent and constraint objects can be constructed.



this symbol represents a dependent object. It can be recognized by the dark left side of the ellipse. This part of the constraint system belongs to the triggering subsystem.

---

<sup>5</sup>

A *property* in the sense of the MADE constraint system is a status variable which is set (and retrieved) by invoking a member function of the object. Variables which are set directly can never be a property.



this symbol represents an independent object. It can be recognized by the dark right side of the ellipse. This part of the constraint system belongs to the triggering subsystem.



this symbol represents an object which is an independent object to one constraint object and a dependent object to another constraint object. This part of the constraint system belongs to the triggering subsystem.



this symbol represents a constraint object. This part of the constraint system belongs to the satisfaction subsystem.

The *constraint objects* themselves can also be added to the constraint network in two ways; either as *eager* constraints objects or as *lazy* constraint objects. The difference between an eager and a lazy constraint objects becomes clear when an independent object triggers a constraint object. Whenever a constraint object is triggered, the changes to the properties of the independent object are *propagated* by that constraint object to the dependent object. An eager constraint object will propagate the changes of the properties of an independent object immediately to the dependent objects<sup>6</sup>. Lazy constraint objects can be used as some kind of buffer to the propagation. These objects will only propagate changes of an independent object to a dependent object if the dependent object explicitly requests such an update<sup>7,8</sup>.

The constraint network in MADE is merely a means to propagate changes of the independent objects to the dependent objects. Because MADE provides an active, object-oriented environment, the objects in mC++ have to obey the Object-Oriented Paradigm; changes to properties have to be made through the object's interface. Thus every time a constrained property is changed, a member function of the independent object has to be invoked. This fact is used in the mechanism to trigger the constraint object. The member function of the independent object which changes the property due to which a constraint object is triggered is called a *triggering member function*. Dependent objects perform a *renewed triggering* action on the constraint object via the *renewed-triggering member function*. Performing a renewed triggering action on a constraint means that a dependent object makes a request (i.e. send a message or invoke a member function) to the constraint object

---

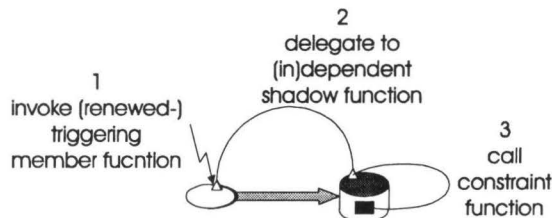
<sup>6</sup> eager constraints are especially useful when changes of independent objects have to have an immediate effect (for instance, when a representation of a property needs to be displayed on an output device).

<sup>7</sup> whenever a request is issued but no propagation is buffered, the dependent object will proceed as if the request was never made.

<sup>8</sup> lazy constraints can be useful when, for instance, figures have to be aligned to a grid. When the grid-size changes, only newly placed figures use the new grid-size and are aligned to the new grid; already placed figures are unchanged. In this case, the figures to be placed are the dependent objects to the constraint object and the grid-size a property of some kind of grid object. Every time a figure is placed, it makes a request to the constraint object to update its current position so that it is aligned to the grid. At that moment, the lazy constraint will propagate changes, made to the grid-size, to the figure (i.e. the dependent object).

to update its properties, if necessary. Renewed-triggering member functions are necessary only in case of lazy constraint satisfaction.

The actual maintenance of the constraints, adapting the properties of the dependent objects when the properties of the independent objects have been altered, is implemented using *delegation*. The triggering member functions of the independent objects and the renewed-triggering member functions of the dependent objects are delegated to special member functions of the constraint object (so-called *shadow functions*); when the application invokes the (renewed-)triggering member function, the runtime system will invoke the shadow member function instead. One of the actions of the shadow function will be to execute the original (renewed-)triggering member function. The purpose of these shadow functions is to enable the constraint system to keep its administration of the constraints in the constraint network up to date. There are two types of shadow functions: *independent shadow functions* (shadow functions for the triggering member functions) and *dependent shadow functions* (shadow functions for the renewed-triggering member functions). The shadow function also activates a *constraint function*. In the constraint function, the programmer specifies the actions which must be performed to maintain the dependency relation. Each constraint object must have at least one constraint function associated with it.



this symbol represents a member function of an object. This can be either a (renewed-)triggering member function or it can be a shadow function. In case of ambiguity, the type of the shadow function may be indicated by an accompanying 'i' (independent) or 'd' (dependent). This part of the constraint system belongs to the triggering subsystem.



this symbol represents a dependency relation between an independent object and a constraint object or between a constraint object and a dependent object. This part of the constraint system belongs to the propagation subsystem.



this symbol represents a delegation from one member function to another member function. Delegation always goes from a triggering member function to an independent shadow function or from a renewed-triggering member function to a dependent shadow function. This part of the constraint system belongs to the triggering subsystem.

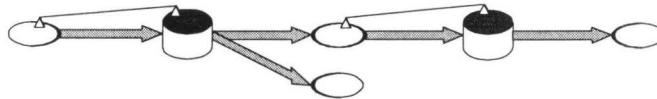
### 4.3.4 Network Configurations

In the MADE constraint system, different kinds of configurations are possible. The basic configuration is where an independent object is connected to a constraint object which, in its turn, is connected to a dependent object. The independent object must provide a triggering member function which triggers an independent shadow function of the constraint object. An renewed-triggering member function only has to be included for lazy constraints. However, because the type

of a constraint object (lazy or eager) can change dynamically, an renewed-triggering member function is also obligatory when, during its lifetime, a constraint object becomes lazy:



On this basic configuration different variations exists. These different variations will be discussed in the next sections. Finally, it is possible to construct arbitrary constraint networks by using these variations; objects which are dependent objects of one constraint object may be an independent object to another constraint object:



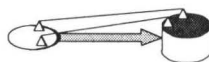
The reason that all the different configurations that are allowed in the MADE constraint system are presented here is that they show that almost any configuration can be specified and thus that, in this respect, the MADE constraint system imposes almost no restrictions. This is important as the expressive power of a constraint system is determined to a large extent by the configurations that can be specified in the constraint network.

### 4.3.4.1 Independent Connections

Independent objects can be connected to a constraint object in several ways. In this section, all possible configurations will be shown. However, two configurations are prohibited:

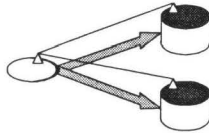
- an independent object may not have one triggering member function which triggers the same constraint object by two different independent shadow functions. For the MADE constraint system, the pair (independent object, triggering member function) has to be unique per constraint object; there are no good semantic reasons to require such situations.
- an independent object may not have more than one triggering member function which triggers the same independent shadow function of a constraint object. The pair (constraint object, independent shadow function) has to be unique per independent object; otherwise it is not possible to notify all the necessary constraint objects that the triggering member function is invoked when this configuration is used.

An independent object may have several triggering member functions which trigger different independent shadow functions of the same constraint object. In this case, invocation of different triggering member functions will trigger the same constraint object. However, during the process of satisfying the constraint, the constraint object can identify which independent shadow function was triggered. And by knowing that, it is possible to identify the triggering member function (or a group of triggering member functions, depending on the precise structure of the constraint network) which triggered the constraint object. This allows the system to perform different actions depending on which member function of the independent object was invoked. This configuration can also be used when the two triggering member functions have a different signature but have to trigger the same constraint object or when two properties of the same independent object constrain the properties of the same dependent object:

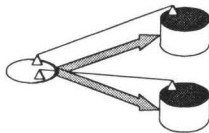




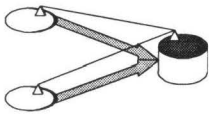
It is also allowed to have one member function trigger more than one independent shadow function of different constraint objects. This allows for the starting of different constraint satisfaction paths at the same time by only a single triggering member function invocation. This is typically used when a certain property of an independent object is used to constrain the properties of several dependent objects:



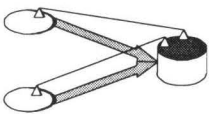
An independent object may have two different triggering member functions which are delegated to the different independent shadow functions of two different constraint objects. This configuration may be used when several properties of the same independent object are used to constrain the properties of different dependent objects:



Different independent objects may activate the same constraint object via the same independent shadow function. This avoids the need to have different independent shadow functions for the triggering member functions of different independent objects when 1) in the constraint function no distinction is made between the different triggering objects/member functions and 2) the triggering member functions have the same signature:



Different independent objects may activate the same constraint object but by triggering a different independent shadow function. The idea behind this scheme is similar to the one where one independent object could have several triggering member functions trigger different independent shadow functions of the same constraint object. Depending on the shadow function triggered, the satisfaction process of the constraint may perform different actions. This implies that, depending on the independent object which activated the constraint object, alternative satisfaction steps might be taken. This configuration is also necessary when the two different independent objects with different signatures trigger the same constraint object:



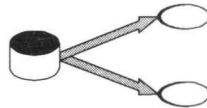


### 4.3.4.2 Dependent Connections

Just as not all possible configurations were allowed to connect an independent object to a constraint object, two configurations exist which are not allowed for connecting dependent objects to a constraint object:

- a dependent object may not have two renewed-triggering member functions which perform a renewed triggering action on the same constraint object via the same dependent shadow function. The pair (constraint object, dependent shadow function) has to be unique per dependent object; otherwise, notification of the necessary constraint objects, that the renewed-triggering member function is invoked, is not possible.
- a dependent object may also not have one renewed-triggering member function which performs a renewed triggering action on two different shadow functions of the same constraint object. The pair (dependent object, renewed-triggering member function) has to be unique per constraint object; it is enough that one renewed-triggering member function performs a renewed triggering action on only one dependent shadow function.

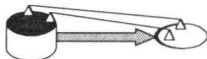
A constraint object may have dependency relations with more than one dependent object. This allows information to be propagated to different paths in the constraint network. This is typically the case when different dependent objects are constrained by the same constraint object:



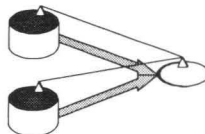
A dependent object may have an renewed-triggering member function to trigger the dependent shadow function. This allows the dependent object to perform a renewed triggering action on the constraint object in case of lazy constraint satisfaction. In that case, constraint satisfaction is only performed when the renewed-triggering member function is invoked:



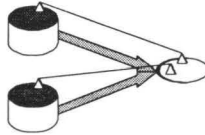
A dependent object may also have two different renewed-triggering member functions which trigger two different dependent shadow functions of the same constraint object. This configuration is necessary when the signature of the two renewed-triggering member functions is different or when two different properties are constrained by the same constraint object:



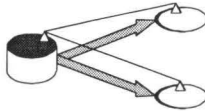
A dependent object may have an renewed-triggering member function which performs a renewed triggering action on several dependent shadow functions from different constraint objects. This allows for lazy constraint satisfaction where the dependent object may be constrained by more than one constraint object:



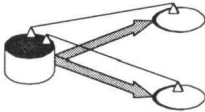
Also allowed is the situation where two different renewed-triggering member functions of the same dependent object perform a renewed triggering action on two different constraint objects. In that case, two different properties of the same dependent object can be constrained by the different constraint objects:



Several dependent objects may have renewed-triggering member functions which perform a renewed triggering action on the same dependent shadow function of the same constraint object. This allows for lazy constraint evaluation in the case where several dependent objects are constrained by the same constraint object:



The last configuration allowed is where two different dependent objects each have an renewed-triggering member function which performs a renewed triggering action on the same constraint object but via different dependent shadow functions. This only makes sense when the renewed-triggering member functions have a different signature. When the signature is the same, the configuration is allowed, but it can also be realized using the previous configuration:



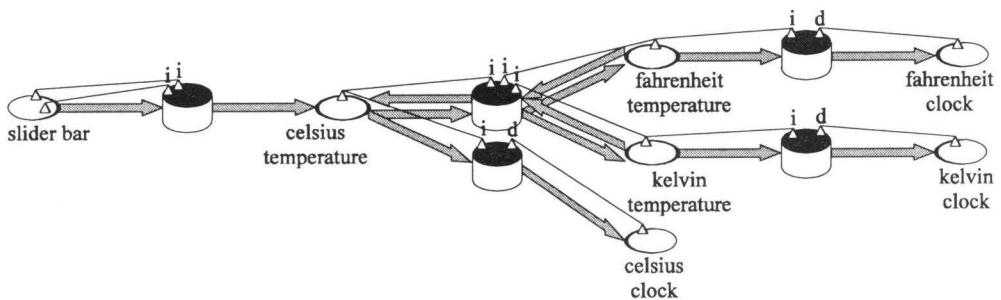
#### 4.4 The Implementation of the MADE Constraint System

In the previous section the concepts of the MADE constraint system were discussed. In this section, more details about the MADE constraint system will be presented. The different sections will show how the MADE constraint system can be used in an mC++ program and what it actually does. The organization of this section is guided by the different actions which can be performed using the constraint network; creating a constraint class (§4.4.1), initializing the constraint system (§4.4.2), building up the constraint network (§4.4.3), breaking it down (§4.4.4), triggering a constraint object (§4.4.5), performing a renewed triggering action on a constraint object (§4.4.6), satisfying the constraint network (§4.4.7 and §4.4.8), saving (part of) the network in the database (§4.4.9) and give a notification when (part of) the network is satisfied (§4.4.10).

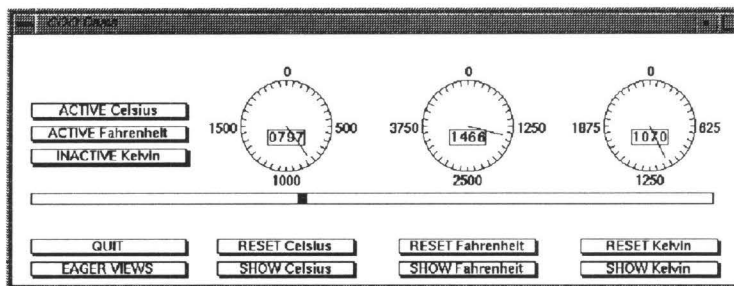
This section demonstrates at a more detailed level the different design aspects which are important when realizing a general constraint system as described in the previous sections. Important in this section is the way in which the constraint system can manipulate and use

information about its triggering subsystem to propagate information through the constraint network and to satisfy constraints.

In the remainder of this chapter, a simple example will be used to show the different aspects of the MADE constraint system. The example which is chosen is a temperature converter. This temperature converter converts degrees Celsius, degrees Fahrenheit and degrees Kelvin into each other. The program is constructed using three objects which hold the temperature according to the Celsius, Fahrenheit or Kelvin system and three objects which show these values on the screen (in the form of a clock). The constraints are constructed as follows:



A small demo program was designed which allowed the user to play with the different values and the status of the different constraint objects. The interface of this program is shown below. The demo application shows the three clocks, each presenting the temperature in one of the three systems; the left clock shows the degrees Celsius, the middle one the degrees Fahrenheit and the right clock shows the degrees Kelvin. The slider bar in the window allows the user to set the Celsius temperature. Via the slider, the user can send his input to the program. The *quit* button is used, obviously, to quit the program. All other buttons can be used to interactively change the status of the different constraint objects.



Below the three clocks are six buttons. With each clock two buttons are associated. One which *resets* the value of the temperature it represents to 0 (the other values get updated accordingly). The other button forces the clock to update the display and show the current value of the temperature it represents. These latter buttons can be used in combination with the button in the lower left corner of the screen (eager views). That button makes the constraint objects between the temperature objects and the clock objects eager (when the button reads 'lazy views') or lazy (when the button reads 'eager views'). The *show* buttons below the clocks can then be used to perform a renewed triggering action on the constraint objects. The buttons on the left of the clocks are used to

activate/inactivate the constraints between the temperature objects and the clock objects. Once such a constraint is inactivated (i.e. the button says *INACTIVE* -), the clock will show no updates whatsoever (not even after pressing the *show* button), because the temperature object will not be able to propagate its changes to the clock objects.

The interface of this demo thus allows to test three important features of the MADE constraint system: eager constraint satisfaction, lazy constraint satisfaction and the dynamical updating of the constraint network.

In the remainder of this chapter, this demo program will be used to show how the principles, that were explained in the previous chapter, can be applied to a real situation. For this example, it is assumed that the following three objects are already declared (they act as dependent and independent objects in the example and are one of the many possible ways in which this data can be represented). Note that no special requirements are made with respect to their implementation (except that they are MADE classes). The definitions of these classes is left out as they are not relevant to the discussion here:

<pre> Unprotected Degree { public:     Degree ();     ~Degree ();      void setTemp (int i);     int getTemp ();  private:     int _value;  }; </pre>	<pre> Unprotected Clock { public:     Clock (int maxVal);     ~Clock ();      void setClock (int i);     int getClock ();  private:     int _max;     int _value;  }; </pre>	<pre> Unprotected Slider { public:     Slider (int maxVal);     ~Slider ();      void setSlider (int i);     void setSliderFromScreen ();     int getSlider ();  private:     int _max;     int _value;  }; </pre>
---	--	--

#### 4.4.1 Creating a Constraint Class

A central role in the constraint system is played by the constraint objects. These constraint objects are instances of so-called constraint classes. Constraint classes are derived from the, by the MADE constraint system provided, CO class. Once a constraint class is defined, constraint objects, and thus constraints in general, can be created as often as desired with only a simple declaration.

The examples in this section show how the constraint side of the triggering subsystem is defined and that only a very limited amount of information is needed (w.r.t. the complex task that has to be performed by the triggering subsystem) to give this definition. This section also shows how parts of the satisfaction subsystem can be defined using constraint functions and what kind of support is needed to be able to make the satisfaction process powerful enough to handle all kinds of constraints.

##### 4.4.1.1 Constraint Class Declarations

The construction of a constraint class generally follows the following rules:

```

Mutex NewConstraintClass : public CO {
public:
    NewConstraintClass ( ... );
    ~NewConstraintClass ( ... );

    void oneConstraintFunction ();

```

```

void twoConstraintFunction ();

// declare public member functions
...

void UpdateSnaptree (MSnaptree&);

declareIndepShadowFunc (resultType, ISFname, (argumentTypes));
declareDepShadowFunc (resultType, DSFname, (argumentTypes));

};

```

The parts of this declaration in bold print are the special parts necessary for a constraint class declaration. First of all, the new constraint class must be derived publicly from the CO class. The new constraint class will inherit some functionality from CO class for which the constraint system assumes that it is present in every constraint class.

The constraint functions must be declared publicly and must have the signature "void CO::\* ()". Per constraint class several constraint functions may be declared. Only one constraint function may be active at a time.

The member function *UpdateSnaptree* is optional and can be defined in case the new constraint object has declared new variables which have to be put in (updated from) the database when the constraint object is stored (retrieved).

The macros *declareIndepShadowFunc* and *declareDepShadowFunc* are used to automatically declare the (in)dependent shadow function as well as some other functions which are necessary for the constraint system to perform a (renewed-)triggering action on the constraint object.

### 4.4.1.2 Constraint Class Definitions

Besides the class declaration, a definition of the various member functions also has to be provided. For constraint classes the following definitions have to be provided on top of the normal definitions:

```

NewConstraintClass::NewConstraintClass ( ... ) : COEager { ... };

void NewConstraintClass::oneConstraintFunction () {
    if (defined (ISFname)) ...
    undefine (ISFname);
    if (lastUpdate (ISFname)) ...
    if (cmpUpdate (ISFname1, ISFname2) == 0) ...
    update (ISFname);
    char* name = cfTriggerObject->nameOfMCclass();
    if (i < cfCycleCount) ...
    Cycle (
        CycleDo (c1) {
            ...
        };
        CycleDo (c2) {
            ...
        };
        CycleBreak;
    );
};

```

```
defineIndepShadowFunc (resultType, NewConstraintClass, ISFname, (argumentList), (argumentNames));
defineDepShadowFunc (resultType, NewConstraintClass, DSFname, (argumentList), (argumentNames));
```

The constructor can be given an initiator which is either *COEager* or *COLazy*. Depending on this initiator, the defined constraint will be either eager or lazy (the type of the constraint can be changed during run-time). In case no initiator is given, the constraint is assumed to be lazy.

In the constraint functions, the constraint programmer may use several special constructs. Most of these deal with the identification of the triggering member function. These functions are listed below to show the reader which kind of information can be retrieved from the information propagation subsystem of the constraint system and how this information can be used to guide the satisfaction process described in the constraint function. This list is not intended to be complete; it provides some basic functionality which makes it possible to describe the most common constraint problems. This is also the basic set which is implemented in the MADE constraint system. However, it is very well possible to come up with additional functions for this list:

- **cfTriggerObject**  
Holds a pointer to the independent object which triggered the current constraint object the last time.
- **defined (ISFname)**  
Holds the value of 1 when the independent shadow function *ISFname* is 'defined'. A shadow function is said to be defined when an independent object has triggered the constraint object via this shadow function and has not been 'undefined' since.
- **undefine (ISFname)**  
Undefines the independent shadow function *ISFname*.
- **lastUpdate (ISFname)**  
Holds the value of 1 when the constraint object was triggered via the independent shadow function *ISFname* the last time. In case no shadow function is defined, the returned value will always be 0.
- **update (ISFname)**  
Allows manipulation of the administration of the various triggering shadow functions. When an *update* is done on a particular shadow function, this shadow function is then considered to have been used in the last triggering of the constraint object. Thus { *update* (F); *lastUpdate* (F); } always returns 1 whereas { *update* (F); *lastUpdate* (G); } always returns 0.
- **cmpUpdate (ISFname1, ISFname2)**  
Compares the relative order in which *ISFname1* and *ISFname2* are used in the triggering of the constraint object. If *ISFname1* has been used more recently than *ISFname2*, or if *ISFname2* is undefined and *ISFname1* is defined, the result of this construct will be 1. If *ISFname2* has been used more recently than *ISFname1*, or if *ISFname1* is undefined and *ISFname2* is defined, the result will be -1. If both shadow functions are undefined, the result will be 0.
- **cfCycleCount**  
Holds the current iteration of the execution of the constraint function in a cycle. In constraint objects which do have cyclic dependencies, the value of *cfCycleCount* is increased every time the constraint function is executed during the iteration of the cycle. Initially, the value of *cfCycleCount* is 1. The value is reset to 1 after the complete cycle is satisfied so that it can be used during the next satisfaction of the cycle. In constraint

objects which do not have cyclic dependencies, the value of *cfCycleCount* will always be 1.

- **Cycle**

Provides the constraint programmer with an abstract construct to manage cycles in the constraint function. The *Cycle* construct has a number of *CycleDo* statements as arguments. The last argument may be a *CycleBreak*, but this argument may also be omitted.

- **CycleDo**

The *CycleDo* statement in its turn has an argument and a body. The argument is a number which has to be larger than 0. The body will be executed whenever the value of the first argument is not greater than the current value of *cfCycleCount*. During each cycle, only the body of one *CycleDo* argument will be executed. The *CycleDo* arguments are checked from first to last argument and therefore, they should be ordered in ascendance by the value of their first argument.

- **CycleBreak**

The *CycleBreak* argument can break the cycle; i.e. the cycle is considered to be satisfied and the constraint system will continue to satisfy the remaining part of the constraint network.

### 4.4.1.3 Examples of Constraint Class Declarations and Definitions

The demo program uses constraint objects of three different constraint classes. The first type of constraint object is an instance of the constraint class *SliderToDegree*, which propagates a value of an instance of the *Slider* class to the value of an instance of the *Degree* class. The constraint class has two independent shadow functions; these functions will act as the shadow functions for the two member functions, *setSlider* and *setSliderFromScreen*, of the class *Slider*. The purpose of the constraint function of *SliderToDegree* is to update the value of an instance of the *Degree* class:

```
Mutex SliderToDegree : public CO {
public:
    SliderToDegree (Slider*, Degree*);
    ~SliderToDegree ();

    void constraintFunction ();

    declareIndepShadowFunc (void, setSliderValue, (int));
    declareIndepShadowFunc (void, setSliderValueScreen, (int));

private:
    Slider* sliderObject;
    Degree* degreeObject;
};

SliderToDegree::SliderToDegree (Slider* s, Degree* d) : COEager { ... };
void SliderToDegree::constraintFunction () { ... };

...

defineIndepShadowFunc (void, SliderToDegree, setSliderValue, (int i), (i));
defineIndepShadowFunc (void, SliderToDegree, setSliderValueScreen, (int i), (i));
```



Note that the *SliderToDegree* constraint class is an eager constraint class by the specification of the *COEager* initiator in the constructor definition of the class. Also, note that the constraint function has the demanded "void CO::\*()" signature.

The second constraint class resembles the *SliderToDegree* constraint class. It also propagates a value from one object to another object. However, this constraint class, *DegreeToClock*, has a dependent shadow function. This is required as instances of the class *DegreeToClock* may become lazy when the user presses the *lazy views* button in the application:

```
Mutex DegreeToClock : public CO {
public:
    DegreeToClock (Degree*, Clock*);
    ~DegreeToClock ();

    void constraintFunction ();

    declareIndepShadowFunc (void, setTemp, (int));
    declareDepShadowFunc (int, getClockValue, ());

private:
    Degree* degreeObject;
    Clock* clockObject;
};

DegreeToClock::DegreeToClock (Degree* d, Clock* c) : COEager { ... };
void DegreeToClock::constraintFunction () { ... };

...

defineIndepShadowFunc (void, DegreeToClock, setTemp, (int i), (i));
defineDepShadowFunc (int, DegreeToClock, getClockValue, (), ());
```

Note that the constraint objects of this class are initially eager (according to the *COEager* specification in the initiator). However, it is possible that the instances are made lazy because a dependent shadow function is available.

The third and last constraint class is *CFK*. This constraint class provides the constraint function which actually converts the different temperatures among the three instances of the class *Degree*. The constraint class is an eager constraint class and the instance of this class will be a cyclic constraint object; each of the three instances of the *Degree* class which are connected to the constraint object will be independent and dependent objects at the same time of that same constraint object. Furthermore, this class has three different independent shadow functions; one for each independent object. These different independent objects are used in the constraint function to determine whether Celsius has to be converted to Fahrenheit and Kelvin, Fahrenheit has to be converted to Celsius and Kelvin or Kelvin has to be converted to Celsius and Fahrenheit<sup>9</sup>.

```
Mutex CFK : public CO {
public:
    CFK (Degree*, Degree*, Degree*);
    ~CFK ();
```

<sup>9</sup> In this case, this could also be done by using *cfTriggerObject*. Instead of `lastUpdate (setTemp?)` the condition could be expressed like `cfTriggerObject == ...Object`. In this case this is possible because the triggering objects are all different.



```

void constraintFunction ();

declareIndepShadowFunc (void, setTempC, (int));
declareIndepShadowFunc (void, setTempF, (int));
declareIndepShadowFunc (void, setTempK, (int));

private:
    Degree *celsiusObject, *fahrenheitObject, *kelvinObject;
};

CFK::CFK (Degree* c, Degree* f, Degree* k) : COEager { ... };
void CFK::constraintFunction () {
    Cycle (
        CycleDo (1) {
            if (lastUpdate (setTempC)) { /* F = 9/5 * C + 32; K = C + 273 */
            else if (lastUpdate (setTempF)) { /* C = (F-32) * 5/9; K = (F-32) * 5/9 + 273 */
            else if (lastUpdate (setTempK)) { /* C = K - 273; F = (K-273) * 9/5 + 32 */
            };
            CycleBreak;
        };
    };

    defineIndepShadowFunc (void, CFK, setTempC, (int i), (i));
    defineIndepShadowFunc (void, CFK, setTempF, (int i), (i));
    defineIndepShadowFunc (void, CFK, setTempK, (int i), (i));
}

```

In all the code presented above, the functionality of the various constraints was all located within the same CO object; maintenance of a constraint is now local to a CO object which may contribute to the maintainability of the constraint program.

### 4.4.2 Initializing the Constraint System

Besides creating the necessary constraint classes, the constraint system also has to be initialized. Afterwards, the administration has to be cleaned up again. The CUI provides two functions: `initConstraintToolkit` and `closeConstraintToolkit`; these functions respectively create an initially empty constraint network and destroy all remnants of a constraint network. They are typically used at the beginning and at the end of an application which uses the constraint toolkit and their purpose is to hide a number of implementation specific details at startup and shutdown. An empty constraint network consists of a ROUTER object with no information and a MCGO object with no CGO objects and no information.

### 4.4.3 Building Up the Constraint Network

Once constraint classes are declared and the administration for the constraint system is set up, a constraint network can be build up. For this purpose, three functions are available:

- `void registerDependentObject (MObject* d, CO* c, char* rf =0, char* dsf =0)`
- `void registerIndependentObject (MObject* i, CO* c, char* tf, char* isf =0)`
- `void registerConstraintObject (CO* c, char* cf)`

The order in which these functions are applied is restricted by only one rule; the constraint object may only be registered after all its dependent and independent objects are registered. There is, however, no strict rule on the order in which independent and dependent objects are registered. Their registration may even be intertwined with registration of dependent and independent objects of other constraint objects. The fact that the order in which the dependent, independent and constraint

objects is loose, does not imply that the construction of the constraint network is an unstructured process; a careful analysis of the constraint problem is needed to determine which property has to be connected to which other property using which constraint. Once these relations are clear, the actual building of the constraint network is trivial. In the next sections, each of above mentioned three functions and their working will be described in more detail. In the demonstration program, the following piece of code is used to initialize and build up the constraint network. This is typically done at the beginning of the *main* function:

```

1 main () {

    MInitialize ();
    InitConstraintToolkit ();

5   Slider* sliderInstance;

    Clock *clockInstanceC, *clockInstanceF, *clockInstanceK;

10  Degree *degreeInstanceC, *degreeInstanceF, *degreeInstanceK;

    SliderToDegree* constraintObjectSD = new SliderToDegree (sliderInstance, degreeInstanceC);
    DegreeToClock* constraintObjectDCC = new DegreeToClock (degreeInstanceC, clockInstanceC);
    DegreeToClock* constraintObjectDCF = new DegreeToClock (degreeInstanceF, clockInstanceF);
15  DegreeToClock* constraintObjectDCK = new DegreeToClock (degreeInstanceK, clockInstanceK);
    CFK* constraintObjectCFK = new CFK (degreeInstanceC, degreeInstanceF, degreeInstanceK);

    registerDependentObject (degreeInstanceC, constraintObjectSD);
    registerIndependentObject (sliderInstance, constraintObjectSD,
20     "void setSlider(int)", "void setSliderValue(int)");
    registerIndependentObject (sliderInstance, constraintObjectSD,
        "void setSliderFromScreen(int)", "void setSliderValueScreen(int)");
    registerConstraintObject (constraintObjectSD, "void constraintFunction()");

25  registerDependentObject (degreeInstanceC, constraintObjectCFK);
    registerDependentObject (degreeInstanceF, constraintObjectCFK);
    registerDependentObject (degreeInstanceK, constraintObjectCFK);
    registerIndependentObject (degreeInstanceC, constraintObjectCFK,
        "void setTemp(int)", "void setTempC(int)");
30  registerIndependentObject (degreeInstanceF, constraintObjectCFK,
        "void setTemp(int)", "void setTempF(int)");
    registerIndependentObject (degreeInstanceK, constraintObjectCFK,
        "void setTemp(int)", "void setTempK(int)");
    registerConstraintObject (constraintObjectCFK, "void constraintFunction()");

35  registerDependentObject (clockInstanceC, constraintObjectDCC, "int getClock()", "int getClockValue()");
    registerIndependentObject (degreeInstanceC, constraintObjectDCC, "void setTemp(int)");
    registerConstraintObject (constraintObjectDCC, "void constraintFunction()");

40  registerDependentObject (clockInstanceF, constraintObjectDCF, "int getClock()", "int getClockValue()");
    registerIndependentObject (degreeInstanceF, constraintObjectDCF, "void setTemp(int)");
    registerConstraintObject (constraintObjectDCF, "void constraintFunction()");

    registerDependentObject (clockInstanceK, constraintObjectDCK, "int getClock()", "int getClockValue()");
45  registerIndependentObject (degreeInstanceK, constraintObjectDCK, "void setTemp(int)");
    registerConstraintObject (constraintObjectDCK, "void constraintFunction()");

    // constraints are now installed and can be used in the application
    ...

50  closeConstraintToolkit ();
    MTerminate ();

};

```

In this code fragment, the bolded parts are necessary to build up the constraint network. In the beginning of the main application, in line 3 and 4, the MADE environment and the constraint system are initialized. At that moment the MADE constraint system has an empty constraint network and the ROUTER object contains no information. At the end of the program, line 51 and 52, the constraint system and the MADE environment are closed down. These lines, 3, 4, 51 and 52, have to be included for all applications which use the MADE constraint system.

At line 18, the construction of the constraint network starts. All the code, from line 18 up to and including line 46, is part of the construction of the constraint network for the demonstration program. Note that in the approach taken in the MADE constraint system, all constraint network related actions can be concentrated at one location. This may provide a better support for the maintenance of the constraint aspects in the application.

In the next three sections it is shown how the triggering subsystem (§4.4.3.1 and §4.4.3.2) is connected and works together with the constraint satisfaction subsystem (§4.4.3.3).

### 4.4.3.1 Registering a Dependent Object

When a dependent object is registered, it has to be specified which object acts as *dependent object* (d) to which *constraint object* (c) and what the signatures are of the *renewed-triggering member function* (rf) and the *dependent shadow function* (dsf). Because only dependent objects of lazy constraints need renewed-triggering functions, the renewed-triggering function may be omitted<sup>10</sup>. In that case no information is stored in the Router object. If a renewed-triggering member function is specified, this specification should follow the proper rules<sup>11</sup>. The same holds for the specification of the *dsf*. If a value for *dsf* is omitted, *dsf* is assumed to have the same value as *rf*.

When a dependent object is registered, *rf* is delegated to *dsf*. Furthermore, the ROUTER object stores the following information:

name shadow function	name trig. function	constraint object	dependency object	type
dsf	rf	c	d	DEP

This information is essential for the proper triggering of the various constraint objects. Details on how this information is used for the triggering subsystem are given in §4.4.6. After the information is stored in the ROUTER object, the dependency relation between the dependent object and the constraint object has to be stored in one of the CGO objects. This information is needed to implement information propagation in the constraint network and it is stored using a simple list which states which dependent objects are directly linked to which constraint objects, which dependent objects are indirectly linked to which constraint objects and vice versa.

---

<sup>10</sup> In that case, the value for *dsf* must also be omitted.

<sup>11</sup> These rules are specified in the document **Specification of the MADE Object Model and of the mC++ language** ([Heeman et al. 93]), under the section Member Function Identification: "[...] A *member function identification* is a string version of the full member function signature (without the class name). [...] No whitespace should appear in the signature string (except between the return type of the function and the function name) [...]. If the return value is a pointer (eg, `int* f()`), then the string to be used should be `"int* f()"` (as opposed to `"int *f()"`). [...]"

#### 4.4.3.2 Registering an Independent Object

The registration of independent objects is more straightforward than the registration of dependent objects. With independent objects, a triggering member function always has to be specified. Independent objects should propagate information and therefore the *triggering member function* (*tf*) should specify where the information should be propagated to. The specification of the *independent shadow function* (*isf*) may be omitted in which case the value of *isf* is assumed to be the same as *tf*.

When an independent object is registered, the member function specified by *tf* is delegated to the member function specified by *isf*. Besides that, the following information is stored by the ROUTER object:

name shadow function	name trig. function	constraint object	dependency object	type
isf	tf	c	i	INDEP

Again this information is needed to implement the triggering subsystem in the MADE constraint system (see also §4.4.5). After the information is stored in the Router, the dependency relation between the independent object and the constraint object has to be stored in one of the CGO objects. The information is stored using a simple list which states which constraint objects are directly linked to which independent objects, which constraint objects are indirectly linked to which independent objects and vice versa.

Finally, a counter has to be initialized which is going to be used to provide priority handling for constraint objects.

#### 4.4.3.3 Registering a Constraint Object

The registration of a constraint object is nothing more than specifying that the *constraint function* (*cf*) is going to be the active one. A constraint object may have several candidate constraint functions of which only one at a time can be the active one. This feature can be used to change the behavior of a constraint object dynamically; the active constraint function can be changed at any time.

#### 4.4.3.4 Examples of Adding Constraint Objects to the Network

In this section, it is shown how the MADE constraint system builds up the network for the constraint objects of the demonstration program (in §4.4.3.4.1 object *SliderToDegree*, in §4.4.3.4.2 object *CFK* and in §4.4.3.4.3 object *DegreeToClock* ) and stores important information.

##### 4.4.3.4.1 Adding the SliderToDegree Constraint Object

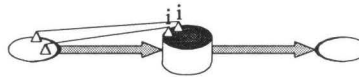
This section discusses the set up of the *constraintObjectSD* object. The associating code is located at line 18 up to and including 23 of the function *main* as shown above. The first instruction (line 18), registers a dependent object *degreeInstanceC* with constraint object *constraintObjectSD*. Because this constraint object is eager and will never become lazy, no renewed-triggering member function (and thus no dependent shadow function) has to be specified for the dependent object.

In line 19-20, 21-22 and 23 two different member functions (*setSlider* and *setSliderFromScreen*) of the same independent object (*sliderInstance*) are registered with the constraint object *constraintObjectSD*. Because both member functions of the independent object have the same

signature but the constraint function of the constraint object may care which member function triggered the constraint object, both triggering member functions have to be delegated to a different independent shadow function. In both cases, the destination to which the triggering member function has to be delegated is specified.

Finally, in line 23 the active constraint function of the constraint object is set to "*constraintFunction*". At this moment, the constraint system contains the following information:

Router				
name shadow function	name (renewed-)triggering member function	constraint object	dependency object	type
void setSliderValue(int)	void setSlider(int)	constraintObjectSD	sliderInstance	INDEP
void setSliderValueScreen(int)	void setSliderFromScreen(int)	constraintObjectSD	sliderInstance	INDEP
CGO				
Constraint Object	Dependent Objects connected directly	Dependent Objects connected indirectly	Independent Objects connected directly	
constraintObjectSD	degreeInstanceC	degreeInstanceC	sliderInstance	
(in)dependent Object	Constraint Objects connected directly	Constraint Objects connected indirectly	Eager Constraint Objects connected indirectly	
sliderInstance	constraintObjectSD	constraintObjectSD	constraintObjectSD	
degreeInstanceC				



#### 4.4.3.4.2 Adding the CFK Constraint Object

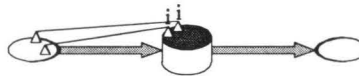
In line 25-34, the *constraintObjectCFK* constraint object is set up. First, each of the three instances of the *Degree* class are registered as dependent objects to *constraintObjectCFK*. Again, as this constraint object will never become lazy, no renewed-triggering member functions have to be specified.

Next, the same three objects are registered as independent objects to that same constraint object. This action makes the constraint object cyclic and a flag will be raised in the constraint object to record this fact. Because all independent objects are of the same type and all three register the same member function as triggering member function, the constraint object will not be able to tell which object triggered the constraint object. Therefore, the triggering member function of each independent object is delegated to a different independent shadow function. This makes it possible for the constraint function (using the functionality of *lastUpdate* and the like) to find out which action has to be performed; i.e. which temperature system has to be converted to which other temperature systems (see also the code in §4.4.1.3).

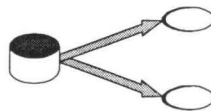
Finally, the active constraint function for *constraintObjectCFK* is registered. The information in the constraint system is updated accordingly.

Note, that by first registering the object *degreeInstanceC* (which was already a dependent object of *constraintObjectSD*) as dependent object to *constraintObjectCFK*, all objects are put in the same cluster of the constraint network (i.e. are managed by the same CGO object). A different situation would occur when first one of the other objects (*degreeInstanceF* or *degreeInstanceK*) would be registered as dependent object. In the case that *degreeInstanceF* and *degreeInstanceK* are registered before *degreeInstanceC* (thus when line 25 would be placed behind line 27 and the execution of the *main* function would only have reached line 27), the following situation would occur (tables are omitted):

CLUSTER 1:



CLUSTER 2:



If, in this situation, *degreeInstanceC* is registered too, the same situation as described earlier in this section is reached. This is because after registration of *degreeInstanceC*, the two clusters are merged.

#### 4.4.3.4.3 Adding the DegreeToClock Constraint Object

The final part of the construction of the constraint network is done in line 36 up to and including line 46. In this part the same actions are repeated three times for the different instances of the *DegreeToClock* class (line 36-38 for *constraintObjectDCC*, line 40-42 for *constraintObjectDCF* and line 44-46 for *constraintObjectDCK*).

This section will only discuss lines 36-38 as the explanation for the other two parts is similar. First, in line 36, a dependent object is registered together with a renewed-triggering member function and its dependent shadow function. The specification of the renewed-triggering member function and the dependent shadow function are necessary because, by interaction of the user which may press the 'eager/lazy views' button, the instances of the *DegreeToClock* class can become lazy. In line 37, an independent object *degreeInstanceC* is registered. In this case, no independent shadow function is specified, so the constraint system assigns the value "void setTemp(int)" to it. Furthermore, in this case, the member function *setTemp* of *degreeInstanceC* is delegated for the second time. So these lines show that one triggering member function can trigger two constraint objects (in this case *constraintObjectCFK* and *constraintObjectDCC*). In line 38, the active constraint function for *constraintObjectDCC* is specified.

The information stored in the constraint network after the whole constraint network is build, is the following:

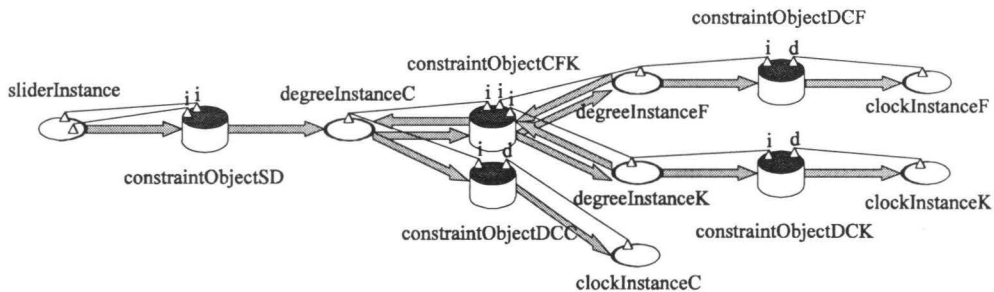
Router				
name shadow function	name (renewed-)triggering member function	constraint object	dependency object	type
void setSliderValue(int)	void setSlider(int)	constraintObjectSD	sliderInstance	INDEP
void setSliderValueScreen(int)	void setSliderFromScreen(int)	constraintObjectSD	sliderInstance	INDEP
void setTempC(int)	void setTemp(int)	constraintObjectCFK	degreeInstanceC	INDEP
void setTempF(int)	void setTemp(int)	constraintObjectCFK	degreeInstanceF	INDEP
void setTempK(int)	void setTemp(int)	constraintObjectCFK	degreeInstanceK	INDEP
int getClockValue()	int getClock()	constraintObjectDCC	clockInstanceC	DEP
void setTemp(int)	void setTemp(int)	constraintObjectDCC	degreeInstanceC	INDEP
int getClockValue()	int getClock()	constraintObjectDCF	clockInstanceF	DEP
void setTemp(int)	void setTemp(int)	constraintObjectDCF	degreeInstanceF	INDEP
int getClockValue()	int getClock()	constraintObjectDCK	clockInstanceK	DEP
void setTemp(int)	void setTemp(int)	constraintObjectDCK	degreeInstanceK	INDEP

CGO			
Constraint Object	Dependent Objects connected directly	Dependent Objects connected indirectly	Independent Objects connected directly
constraintObjectSD	degreeInstanceC	degreeInstanceC, degreeInstanceF, degreeInstanceK, clockInstanceC, clockInstanceF, clockInstanceK	sliderInstance
constraintObjectCFK	degreeInstanceC, degreeInstanceF, degreeInstanceK	degreeInstanceC, degreeInstanceF, degreeInstanceK, clockInstanceC, clockInstanceF, clockInstanceK	degreeInstanceC, degreeInstanceF, degreeInstanceK
constraintObjectDCC	clockInstanceC	clockInstanceC	degreeInstanceC
constraintObjectDCF	clockInstanceF	clockInstanceF	degreeInstanceF
constraintObjectDCK	clockInstanceK	clockInstanceK	degreeInstanceK
(in)dependent Object	Constraint Objects connected directly	Constraint Objects connected indirectly	Eager Constraint Objects connected indirectly
sliderInstance	constraintObjectSD	constraintObjectSD, constraintObjectCFK, constraintObjectDCC, constraintObjectDCF, constraintObjectDCK	constraintObjectSD, constraintObjectCFK, [ constraintObjectDCC ], [ constraintObjectDCF ], [ constraintObjectDCK ]

degreeInstanceC	constraintObjectCFK, constraintObjectDCC	constraintObjectCFK, constraintObjectDCC, constraintObjectDCF, constraintObjectDCK	constraintObjectCFK, [ constraintObjectDCC ], [ constraintObjectDCF ], [ constraintObjectDCK ]
degreeInstanceF	constraintObjectCFK, constraintObjectDCF	constraintObjectCFK, constraintObjectDCC, constraintObjectDCF, constraintObjectDCK	constraintObjectCFK, [ constraintObjectDCC ], [ constraintObjectDCF ], [ constraintObjectDCK ]
degreeInstanceK	constraintObjectCFK, constraintObjectDCK	constraintObjectCFK, constraintObjectDCC, constraintObjectDCF, constraintObjectDCK	constraintObjectCFK, [ constraintObjectDCC ], [ constraintObjectDCF ], [ constraintObjectDCK ]
clockInstanceC			
clockInstanceF			
clockInstanceK			

Note that *constraintObjectDCC* is placed between '[' ]' under the heading 'Eager Constraint Objects'. This has to do with the fact that *constraintObjectDCC* can be lazy. In case the constraint is lazy, it is removed from the 'Eager Constraint Objects' list. The same holds for *constraintObjectDCF* and *constraintObjectDCK*. The information described in the two tables above is, except for the specification of the constraint functions, all the information the constraint system needs to perform the constraint satisfaction. This information totally describes the constraint network; the Router contains information for the triggering subsystem and the CGO's for the information propagation. Furthermore, the pointers in both, the Router and the CGO, provide access to the information of the satisfaction subsystem (stored in the various constraint functions). The diagram below visualizes the same information as is described in the table above:



#### 4.4.4 Breaking Down the Constraint Network

Just as the constraint network can be build up, (parts of) the constraint network can also be broken down. This breakdown can be temporary or permanent. The possibility to build up the constraint network or to break it down at runtime is a very powerful feature of the MADE constraint system; the benefits of dynamic alteration of the constraint network were already discussed in Chapter 3.



### 4.4.4.1 Temporary Breakdown

It is possible to inactivate individual constraint objects temporarily. In that case, the constraint system will treat these constraint objects in a different manner. In case of network building or permanent network breakdown, the inactive constraint objects are treated as normal active constraint objects. In case of propagation, the inactive constraints are treated as if they were not registered (and thus non-existent in the constraint network). Of course, temporarily inactivated constraint objects can also be activated again.

### 4.4.4.2 Permanent Breakdown

In the case of a permanent breakdown, information about the different objects is removed from the ROUTER object and the CGO objects. Dependent, independent and constraint objects can be removed individually or as a group.

When unregistering a dependent object from the constraint network, one connection or a group of connections can be removed. This can be controlled by values which identify the *renewed-triggering member function* and the *shadow function* of connection that has to be removed. If one of these identifiers or both identifiers are omitted, it is assumed that all applicable values may be assumed for the omitted identifier. This allows for the removal of whole groups of connections.

When the dependent object is left with no connections at all, it will be removed from the constraint network as well. The same holds for constraint objects; when no independent or dependent objects are left for a particular constraint object, the constraint object itself is removed from the constraint network. If some (in)dependent objects are still present, the active constraint function of a constraint object will change to an empty constraint function (i.e. a constraint function with an empty body). Because the dependency relations of the constraint object have been changed, the former active constraint function is replaced by another active constraint function for safety reasons.

The unregistering of an independent object is similar to the unregistering of a dependent object.

When a constraint object is unregistered from the constraint network, all links to this constraint object are removed. This means that all information regarding dependency relations is removed from the CGO object (independent as well as dependent). Also all information with respect to the constraint object in the ROUTER object is removed.

If, as a result of the removal of the constraint object and its links, dependent or independent objects remain which are left with no links to other constraint objects, these objects will be unregistered from the constraint network as well.

Important with respect to the breakdown of the constraint network is that the rest of the constraint network (after partial) breakdown is made consistent (which is important for correct constraint satisfaction). For this purpose, the information stored in Router and the CGO's is important again to guide this process.

### 4.4.5 Triggering a Constraint Object

Whereas the previous sections dealt with the (de)construction of the constraint network, the next sections will discuss the use of the constraint network. In this section, the triggering of a constraint object will be discussed. The triggering of a constraint object may start a constraint network satisfaction process; in case the triggered constraint is invalidated, the constraint system will start

constraint satisfaction on the appropriate part of the constraint network. Details on constraint satisfaction can be found in §4.4.7.

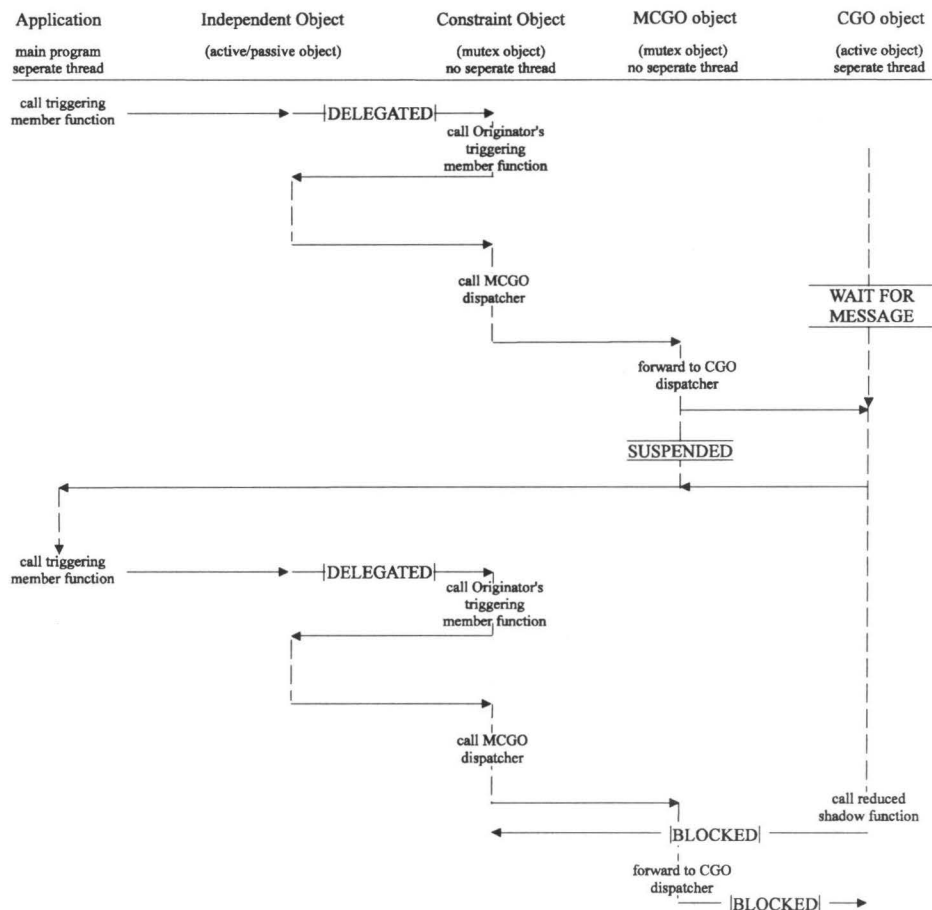
When a triggering member function is invoked, the message is delegated to an independent shadow function of the prototype of the *CO* class. In this case there are two possibilities; the triggering member function is connected to only one independent shadow function, or the triggering member function is delegated to several independent shadow functions. In the first case there is not really a problem. However, in the second case, a problem occurs as in MADE only one delegation can be active at a time. Therefore, the triggering member function can only notify one constraint object of the fact that the triggering member function was invoked. However, it is necessary to notify all the constraint objects which are connected to the independent object via the triggering member function. Therefore, each original shadow function is split in two parts; one part is used as the destination of the delegation which will execute the original triggering member function (using the `Originator::Delegator_function`) and then (note the sequential execution which is very important) instruct the constraint system to execute the *reduced shadow functions* of all the other constraint objects to which the independent object was connected via the triggering member function. Information about which constraint objects to address is retrieved from the ROUTER object (§4.4.4.1). To avoid a deadlock when the constraint system is triggered twice in a row and the CGO object is still calling the reduced shadow functions, the shadow functions themselves are a member function of the prototype of the *CO* class.

The diagram below shows a way in which such a deadlock situation could occur if both, the shadow function and the reduced shadow function, would be member functions of the constraint object (more details on how the process of triggering a constraint object actually works is described below whereas the semantics of thread control were explained in §4.2.2.4). This thesis does not contain any exact proof that in the design of the MADE constraint system no deadlock can occur. This has to do with the fact that the proof that no deadlock can occur is NP hard; it would be beyond the scope of this thesis to present extensive proofs. Intuitively it can be made clear that no deadlocks will occur. Deadlocks can only occur during the communication of two different threads in the constraint system. This means that deadlock can only occur when the application has to communicate (indirectly) with the CGO objects and thus when a constraint is triggered (or, as will be described in the next section, when a renewed-triggering action is performed). During these (renewed-)triggering actions, the application does not (in)directly invoke member functions of objects which are also invoked by the CGO object. Therefore neither the application nor the CGO object will call upon objects which are serviced in another thread and which will call upon objects serviced in their own thread. As a result of that, no deadlock can occur during these (renewed-)triggering actions.

A reduced shadow function, will update the administration of the constraint object; it will invalidate the constraint object and update the history about which independent shadow functions were used to trigger the constraint objects. By using one shadow function and several reduced shadow functions, it is possible to have one triggering member function which is connected to several independent shadow functions of different constraint objects. In this way, the original triggering member function will only be called once while the administration of all the appropriate constraint objects is updated.

Finally, the pair (independent object, triggering member function) is put on a list to be candidates for constraint satisfaction and the constraint system is requested by the independent shadow function to start a new satisfaction process. An optimization is made in this process. If a candidate is already on this list, the constraint system will not start a second satisfaction process; because there are candidates for constraint satisfaction, the constraint system is still busy satisfying (part of) the constraint network (in a separate thread). And because the candidate is already on the list, it will automatically be satisfied by the current satisfaction process. In this case, changes to the different

objects have occurred quicker than the constraint system was able to satisfy the total constraint network.



In the demonstration program, a constraint object is triggered when the user changes the position of the thumbnail on the slider bar. When the thumbnail is moved, the member function *setSliderFromScreen* is invoked. At that moment, several actions are performed by the constraint system. A scheme of this process is shown in the diagram below. In this diagram, function calls are represented by horizontal arrows. Also the return of the execution of a member function is represented by a horizontal arrow. In the diagram there are two threads present; one for the application and one for the (active) CGO object. The other objects are passive and therefore their member functions are executed in the thread of their caller.

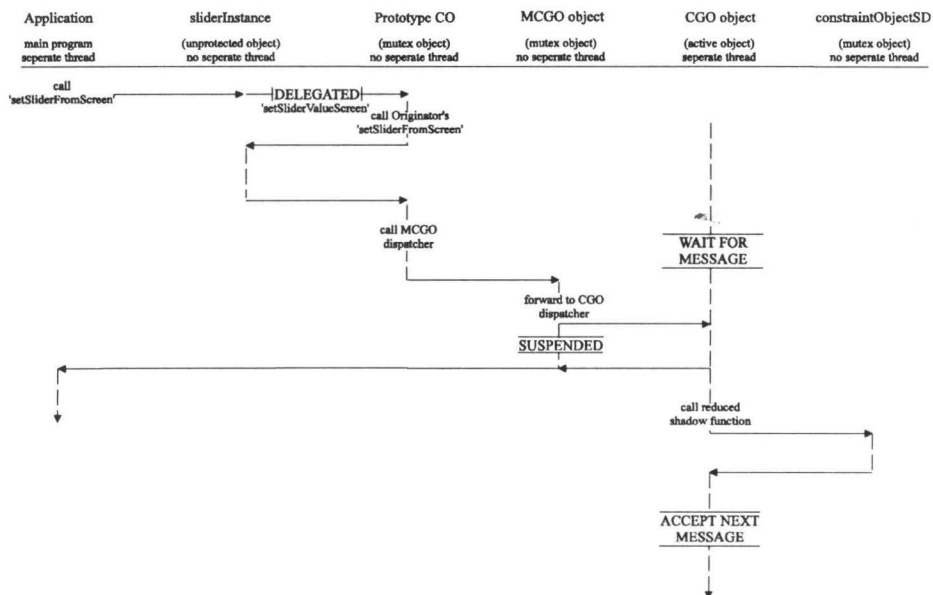
When the main application invokes the member function *setSliderFromScreen* of the object *instanceSlider*, the thread of the application will execute this member function. The invocation of *setSliderFromScreen* is delegated to the independent shadow function *setSliderValueScreen* of the

prototype of class *CO* at which point the constraint system is triggered. This is according to the specification of line 21-22 of the main application. Also the shadow function *setSliderValueScreen* is executed in the thread of the main application.

The independent shadow function will first call (still in the thread of the main application) the triggering member function (i.e. *instanceSlider->setSliderFromScreen(...)*). The information which triggering member function has to be called, can be retrieved from the *Originator::Delegator\_function* information provided by the mC++ runtime library. After the triggering member function has returned control to *constraintObjectSD* the constraint object will call the MCGO dispatcher in order to notify all the constraint objects for which *instanceSlider->setSliderFromScreen(...)* is a triggering member function. Also this call is done in the thread of the main application.

The MCGO dispatcher will forward the call to the appropriate CGO object. Immediately after that, the CGO object will return control to the main application. From that moment on, the remaining part of the triggering process and the actual satisfaction of the different constraint objects can be done in parallel with the main application; the main application does not have to wait before the constraints are satisfied but can continue with its execution after the CGO object is notified.

The remaining actions which have to be performed by the constraint system, to complete the triggering process, is to call the different reduced shadow functions. Because *instanceSlider->setSliderFromScreen(...)* only triggers *constraintObjectSD*, the CGO object only has to invoke the reduced shadow function of *constraintObjectSD*. This information can be retrieved from the ROUTER object, because the reduced shadow function has the appropriate information: the pointer to the original Delegator of the independent shadow function and the name of the triggered independent shadow function. After the constraint object has returned control to the CGO object, the CGO object has finished the dispatching process and also its contribution to the triggering process. This also completes the triggering process.



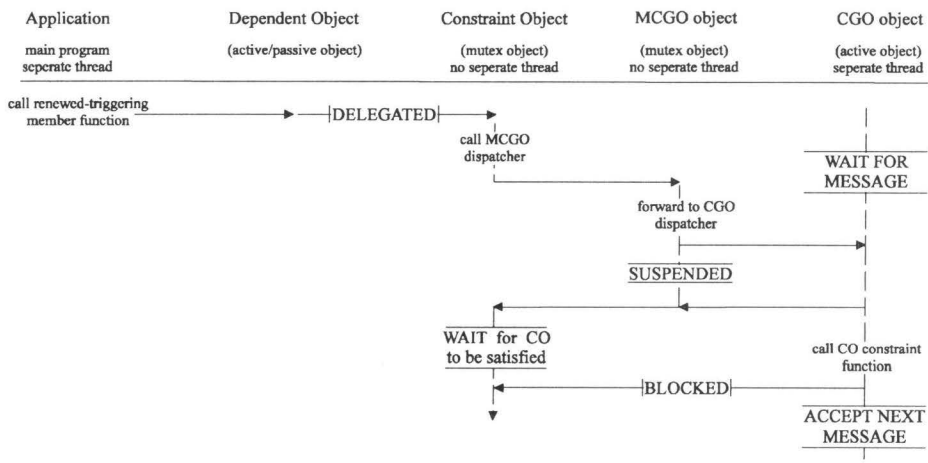
Note that due to the concept of delegation and the Dynamic Call Interface (used to invoke the reduced shadow functions), the triggering of the constraint system can be done totally transparent to the main application.

### 4.4.6 Performing a Renewed-Triggering Action on a Constraint Object

The process of performing a renewed-triggering action on a constraint object uses a different approach than the triggering of a constraint object although parallels can be found.

The dependent shadow function is not a member function of the constraint object itself but of its prototype. The reason here is also to avoid deadlocks. This can be made clear in the following way. The original renewed-triggering function has to be called after the constraint functions of the constraint objects it is connected to are satisfied. Therefore, the shadow function must first call upon the dispatcher of the MCGO object to propagate its request for constraint satisfaction and then wait for the satisfaction process to be completed. Furthermore, the shadow function may not return control to its caller until the original renewed-triggering function is executed (otherwise, the original intended execution order would be disturbed).

If the dependent shadow function is a member function of the constraint object itself, the constraint object, while blocking its other (mutex) member functions by not returning control to the dependent shadow function (its caller), waits to become valid. At the same time, the CGO object tries to validate the constraint, but is blocked by the mutex mechanism. This can be avoided if the dependent shadow function is a member function of the prototype of the constraint object. This prototype can easily be accessed when the original constraint object is known. Furthermore, as the prototype is another object than the constraint object itself, the CGO object would not be blocked anymore when trying to satisfy the constraint:



In case of the dependent shadow function (as opposed to the independent shadow function), no separate administration per constraint object has to be maintained. Therefore, there is no need to have something like a reduced dependent shadow function. It is sufficient to identify all the lazy constraint objects which have the renewed-triggering object as dependent object (this information can be found using the data stored in the Router object: §4.4.3.2). From these constraint objects, the constraint function has to be executed. Because the dependent shadow function is a member

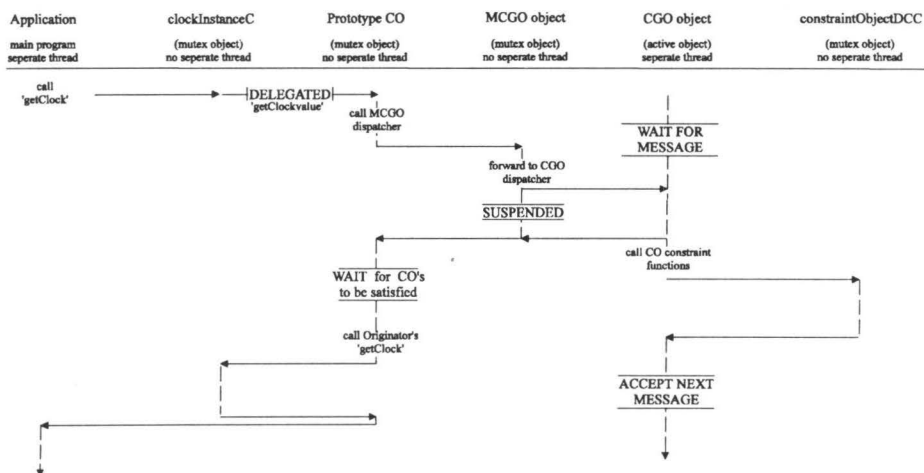
function of the constraint class prototype, the constraint functions of the different constraint classes can be executed without any problems of deadlock. When a constraint object is renewed-triggered, its constraint function will not be executed using a special CO\_WRAP object in order to achieve parallelism (see also §4.4.7.1). As the dependent object has to wait for the constraint object to become valid, there is not much to gain from parallel constraint satisfaction. Constraint satisfaction will only occur when the constraint is invalid. This can only be the case with lazy constraints, as eager constraints are satisfied immediately after they are invalidated. Therefore, when a dependent object performs a renewed triggering action on an eager constraint, the constraint system will perform no extra actions than only execute the original renewed-triggering member function.

Once all the constraint functions of the constraint objects, which can be renewed-triggered by `Delegator->Delegator_function`, are satisfied, the original renewed-triggering member function may be executed. Only after that, the renewed-triggering member function may finish its execution and return control to the dependent object. The dependent object is suspended during that time to ensure that the retrieved status information of the dependent object, which is constrained by some constraint object, is not out of date too soon.

In the demonstration program a constraint object can be renewed-triggered when the user presses the *SHOW* - button. When this button is pressed, the current value of the appropriate instance of the *Degree* class is presented via its associated clock. The diagram below shows the scheme of the process when the *SHOW CELSIUS* button is pressed.

When the user presses the button, the function `getClock` of the object `clockInstanceC` is invoked in the thread of the application. This function call is delegated (according to line 36 of the main application) to the dependent shadow function `getClockValue` of the prototype of class *CO* (also in the thread of the application). The first action of the dependent shadow function is to call the dispatcher of the MCGO object.

The MCGO dispatcher forwards the call to the CGO object. After the CGO object has serviced the message of the MCGO object, the MCGO object returns control to the prototype. The prototype will now wait until all the constraint objects, for which `clockInstanceC->getClock()` is a renewed-triggering member function, are validated. When all those constraint objects are validated, the prototype will call `getClock` of `clockInstanceC` (using the `Originator::Delegator_function` function of the mC++ runtime library) and return control to the main application.



While the prototype waits for the different constraint objects to become valid, the dispatcher of the CGO object will call the constraint functions of the different constraint objects. Because `clockInstanceC->getClock()` triggers only one constraint object (from the information stored by ROUTER object it shows that the only match with (Delegator, dependent shadow function) is *constraintObjectDCC*), only the active constraint function of that constraint object (according to line 38 of the main application this is *constraintFunction()*) will be called. Once this is done, the contribution of the CGO object to the renewed triggering action is finished. Also here, the use of delegation and the Dynamic Call Interface allow for transparent renewed-triggering actions.

### 4.4.7 Constraint Satisfaction and Information Propagation

Constraint satisfaction is realized by executing the different constraint functions of the appropriate constraint objects. The execution of the constraint functions themselves are not that interesting. Still some remarks can be made about this process. The main attention in this section will be on the propagation of changes of properties through the constraint network; how does the MADE constraint system select which constraint objects should execute their constraint function and how their executions are ordered in time to provide an efficient satisfaction process. This section will also illustrate that the aim of the MADE constraint system to support parallel constraint satisfaction makes this task even more complex.

The propagation and satisfaction mechanisms are illustrated on the basis of the demonstration program. The propagation of information through the constraint network starts after a constraint object is triggered. Note that, as a result of performing a renewed triggering action on a constraint object, another constraint object may be triggered. This situation may occur when the constraint function of the renewed-triggered constraint object invokes a triggering member function of other constraint objects. However, it is not always true that a renewed triggering action starts a constraint satisfaction process.

The demonstration program allows the user to change the status of the different constraint objects used in the program. First of all, the user can activate and inactivate the different constraints and, secondly, the user can make the views either eager or lazy. The example that will be used here assumes that the instances of the constraint class *DegreeToClock* are active and eager. This means that the constraint system has the information as described at the end of §4.4.3.4.3. After the function *setSliderFromScreen* has triggered the constraint object *constraintObjectSD*, this constraint has been invalidated by the reduced shadow function and (*sliderInstance*, "void setSliderFromScreen(int)") has been put on the list of candidates for constraint satisfaction. Furthermore, it is assumed that the constraint system is not busy satisfying any other constraints.

#### 4.4.7.1 Executing the Constraint Function

In the MADE constraint system, a constraint object (and thus the constraint it represents) is satisfied by executing the constraint function of that particular constraint object. The execution of this constraint function will be done in parallel with the main application and even in parallel with the constraint system itself.

In the previous sections it was explained that different clusters of the constraint network which have no connections with each other, are maintained independently of each other and parallel to each other. This is realized by the CGO objects which have their own thread of control. It may be clear that the satisfaction process of a cluster within one CGO object cannot possibly affect the clusters in other CGO objects. Because there are no connections between objects in the clusters of different CGO objects, changes to the status of an object in one cluster do not affect objects in another cluster.

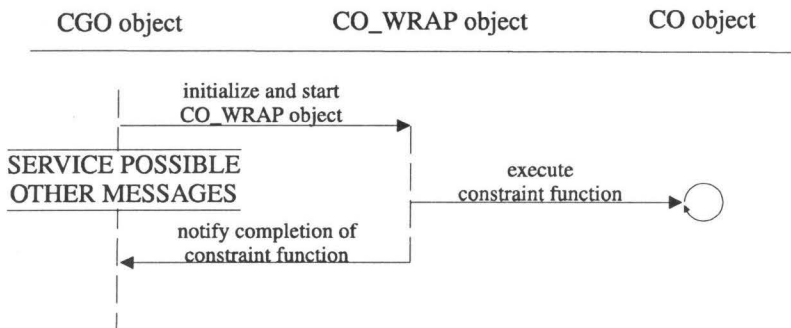


This approach is extended to the satisfaction process within each CGO object and a similar reasoning can be set up for constraint objects, within the same cluster, which have to be satisfied; if there is no path, following the dependency relations in the direction from independent object to constraint object to dependent object, from one constraint object to the other constraint object, the satisfaction process of the two constraint objects can be executed in parallel. This means that several constraint objects within the same cluster can be solved in parallel. That these kinds of situations occur in practice will be shown in more detail in the example used in this section.

Thus, once it is established that a number of constraints can be solved in parallel, separate threads have to be allocated; one for each constraint function that has to be executed. These threads are encapsulated in the different instances of the CO\_WRAP class. CO\_WRAP objects are active objects. Their sole purpose is to execute the constraint function. Given a pointer to a constraint object and its active constraint function, the CO\_WRAP object will execute the constraint function in its own thread of control and afterwards, when the execution of the constraint function is finished, notify the appropriate CGO object that the constraint is satisfied.

In the current implementation of the MADE constraint system, CO\_WRAP objects are created and deleted every time a constraint function is executed. This means that every time this process begins, a thread has to be initialized and, afterwards, killed. The creation of a new thread of control can be an expensive and time-consuming operation. Therefore, an optimization could be made by letting the CGO objects share a pool of idle CO\_WRAP objects. Every time a CGO object is in need of a CO\_WRAP object, it takes one out of this pool of idle ones. Only if this pool is empty, a new CO\_WRAP object may be created. Afterwards, when the CGO has no need anymore for the CO\_WRAP object, this CO\_WRAP object may be placed back in the pool. An extra restriction may be placed on the number of total active CO\_WRAP objects; a new CO\_WRAP object may only be created when the number of CO\_WRAP objects in use by the different CGO objects does not exceed a certain threshold. On the other hand, when the pool of CO\_WRAP objects remains filled with rather a lot of idle CO\_WRAP objects, the constraint system could consider killing some of them in order not to claim too much resources. The management over the pool of CO\_WRAP objects would then be done best by a separate active object. It should be a separate object to maintain the highly parallel character of the satisfaction process of the different CGO objects and it should be active to give the object his own control over the life cycle of the different idle CO\_WRAP objects in the pool.

At this point the possibility of deadlocks has to be considered again: CGO and CO\_WRAP objects all have their own thread of control and they do communicate with each other. However, the communication between these objects is strictly controlled by the constraint system itself and no communication between CO\_WRAP objects and the application can occur. Therefore, the danger of deadlock is not present here.





### 4.4.7.2 Making Execution Plans

The propagation mechanism of the MADE constraint system which determines which constraint function to execute next or in parallel, is guided by the *execution plans* that the system has made. Execution plans can be compared with caching; some of the work done to determine the execution order of the various constraint functions is stored for later re-use.

Because constraints can be satisfied in parallel, execution plans cannot describe the absolute order in which constraint functions should be executed. As a matter of fact, there is very little that can be said about the order at all. This is due to the fact that it is hard to get any information about the time it takes to satisfy a constraint and thus to synchronize the satisfaction processes of the different constraints with each other and the main application. Therefore, the MADE constraint system uses execution plans in a different way. An execution plan consist of pointers to constraint objects which could be invalidated by the invocation of a certain triggering member function of a particular independent object (an execution plan is therefore associated with the pair (independent object, triggering member function)). In general, an execution plan contains pointers to those constraint objects which obey the following rule:

```
(the constraint object is eager) AND
(the constraint object is active) AND
(
  ( the independent object associated to the execution plan is an independent
    object of the constraint object
  ) OR
  ( the constraint object can be (recursively) reached by following an already
    found constraint object via one of its dependent objects to the next
    constraint object
  )
)
```

Some pointers may occur in the execution plan more than once. In that case, there is more than one path which leads from the triggering object (only via other eager, active constraint objects) to the duplicated constraint object. In such situations, a pointer to such a constraint object is added to the execution plan as often as there are such (different) paths. The pointers in the execution plan will be used to check which execution functions of which constraint objects may *not* be executed yet. Note that the execution plan serves the opposite purpose as is normally the case. As long as a pointer to a constraint object is present in the plan, the constraint function of that object may not be executed.

Execution plans are associated with the pair (independent object, triggering member function). This seems to be the most logical association; even more logical than associating the execution plans with constraint objects. This has to do with the fact that independent objects may set off constraint satisfaction and may even trigger several constraint objects at the same time. In the case that execution plans are associated with constraint objects, one triggering action could activate several execution plans. That would mean that more administration would be necessary.

Execution plans are made only when they are needed (i.e. when an independent object has triggered a constraint object) and a valid plan does not yet exist. Once a execution plan is created, it is stored for later re-use. Note that, during the construction of the constraint network, no execution plans are created as these plans are only needed when the constraint network needs to be satisfied. However, during the (temporary) breakdown of the constraint network, all plans which are affected

by the breakdown will be destroyed; when the structure of the constraint network changes, the validity of the execution plans is endangered.

When an execution plan is created, the constraint system will check if a plan, associated with the independent object and the triggering member function, exists. If not, an execution plan is created which guarantees that no constraint object has to be satisfied twice (i.e. during the satisfaction of the constraint objects of the execution plan, no constraint object becomes invalidated after it is satisfied once) and which tries to solve as many constraints in parallel as possible. This approach will give a semi-optimal solution to the problem at hand; the MADE constraint system does not make any assumptions with respect to the time needed to satisfy a particular constraint or load-balancing. In practice, it may be better to delay the satisfaction of a constraint object, even if it could be satisfied in parallel with the other ongoing satisfaction process, to get a better load-balance. However, these metrics depend very heavily on the underlying hardware structure and are also outside the scope of this thesis.

In the demonstration program the CGO object will take the pair (*sliderInstance*, "void setSliderFromScreen(int)") from the candidate list and start to look for its execution plan. The first time the CGO looks for this plan, it will not find it, and a new execution plan will be created. This plan will contain pointers to the following constraint objects (the information in this section is based on the tables presented in §4.4.3.4.3):

constraintObjectSD, constraintObjectCFK, constraintObjectCFK, constraintObjectCFK, constraintObjectCFK,  
constraintObjectDCC, constraintObjectDCC, constraintObjectDCF, constraintObjectDCK

Note that the constraint object *constraintObjectCFK* has four occurrences in the execution plan and *constraintObjectDCC* two. This is because there are four different paths from the triggering object to this *constraintObjectCFK* and two to *constraintObjectDCC*:

sliderInstance - constraintObjectSD - degreeInstanceC - constraintObjectCFK  
sliderInstance - constraintObjectSD - degreeInstanceC - constraintObjectCFK - degreeInstanceC - constraintObjectCFK  
sliderInstance - constraintObjectSD - degreeInstanceC - constraintObjectCFK - degreeInstanceF - constraintObjectCFK  
sliderInstance - constraintObjectSD - degreeInstanceC - constraintObjectCFK - degreeInstanceK - constraintObjectCFK  
  
sliderInstance - constraintObjectSD - degreeInstanceC - constraintObjectDCC  
sliderInstance - constraintObjectSD - degreeInstanceC - constraintObjectCFK - degreeInstanceC - constraintObjectDCC

The execution plan can be retrieved from the information stored in the ROUTER object and the CGO object. First of all, using the information from the ROUTER object, it is possible to find all the constraint objects which are triggered by (*sliderInstance*, "void setSliderFromScreen(int)"). This means that from the table in the ROUTER object those entries should be selected which match:

name shadow function	name (renewed-)triggering member function	constraint object	dependency object	type
-	void setSliderFromScreen(int)	-	sliderInstance	INDEP

In the demonstration program, the result would be:

name shadow function	name (renewed-)triggering member function	constraint object	dependency object	type
void setSliderValueScreen(int)	void setSliderFromScreen(int)	constraintObjectSD	sliderInstance	INDEP

thus, one constraint object is found: *constraintObjectSD*. Using this result, the tables of the CGO object are searched. The algorithm that does this follows the directly connected dependent and constraint objects. A constraint object is not processed twice; a dependent object is processed as often as it is encountered:

*constraintObjectSD* is directly connected to *degreeInstanceC*

*degreeInstanceC* is directly connected to the eager constraint objects *constraintObjectCFK* and *constraintObjectDCC*

*constraintObjectCFK* is directly connected to *degreeInstanceC*, *degreeInstanceF* and *degreeInstanceK*

*constraintObjectDCC* is directly connected to *clockInstanceC*

*degreeInstanceC* is directly connected to the eager constraint objects *constraintObjectCFK* and *constraintObjectDCC*

*degreeInstanceF* is directly connected to the eager constraint objects *constraintObjectCFK* and *constraintObjectDCF*

*degreeInstanceK* is directly connected to the eager constraint objects *constraintObjectCFK* and *constraintObjectDCK*

*clockInstanceC* is directly connected to nothing

*constraintObjectCFK* already processed

*constraintObjectDCC* is directly connected to *clockInstanceC*

*constraintObjectCFK* already processed

*constraintObjectDCF* is directly connected to *clockInstanceF*

*constraintObjectCFK* already processed

*constraintObjectDCK* is directly connected to *clockInstanceK*

*clockInstanceC* is directly connected to nothing

*clockInstanceF* is directly connected to nothing

*clockInstanceK* is directly connected to nothing

All the underlined constraint objects, together with the objects found from the information stored in the ROUTER object, form the new execution plan as presented above.

### 4.4.7.3 Satisfaction Using Execution Plans

When the MADE constraint system starts satisfying (part of) a cluster of the constraint network (as a parallel process to the application in one of the CGO objects), it will first search for an execution plan. If such an execution plan does not exist or is not valid anymore, a new plan, as described in the previous section, is made.

When starting the satisfaction process, only those constraint objects will be considered which are contained at least once in the execution plan; other constraint objects will not be affected by the independent object which started this satisfaction process. The MADE constraint system uses four lists to perform the administration for the satisfaction process:

- **execution set:**

This set contains pointers to the constraint objects which still have to be satisfied. Initially, the set contains a copy of the execution plan, without all duplicate pointers.

- **block set:**

This set contains pointers to the constraint objects whose constraint function may not yet be executed. This set may contain duplicate pointers. Initially it contains an exact copy of the execution plan. At the start of the satisfaction process, for each constraint object, for which the triggering object is the independent object, one pointer is removed from the block set.

- **cyclic set:**

This set contains pointers to cyclic constraint objects which have been encountered during the satisfaction process and which have not yet been satisfied. Thus, initially, the set is empty.

The *root* of the cyclic set is the pointer to the constraint object which is used as starting point in the cycle from where information in the cycle is propagated from and which can be used to check whether the cycle has been traversed once.

◦ **current cycle set:**

This set contains pointers to the cyclic constraint objects which have been visited in the current iteration of the cycle if a cycle is satisfied. If no cycle is satisfied at a moment in time, the set is empty. Thus, initially, the set is empty.

Once these four lists are created using the execution plan, the satisfaction process continues according to the following rules:

- 1) Determine the pointers which are contained in the execution set, but not in the blocked set.
- 2) From those pointers select the ones which do not point to cyclic constraint objects;  
if such pointers do not exist (the constraint system has to deal with cycles)
  - if the current cycle set is not empty (one complete iteration of the cycle is finished)
    - if one of the constraints in the current cycle is invalidated
      - ◆ remove *root* from the block set,
      - ◆ remove the current cycle set from the cyclic set,
      - ◆ add the current cycle set to the execution set and
      - ◆ make the current cycle set empty.
    - if none of the constraints in the current cycle is invalidated
      - ◆ remove the current cycle set from the cyclic set and the block set,
      - ◆ remove those pointers from the block set which point to constraint objects which have an independent object which is also a dependent object of one of the constraint object from the current cycle set and
      - ◆ make the current cycle set empty.
  - if the execution set is not empty
    - select a pointer from the cyclic set. This pointer will be called the *root* of the cycle,
    - remove the *root* and all its duplicates from block set
- if such pointers do exist
  - start the parallel execution of their constraint functions by starting up the CO\_WRAP objects.
- 3) Whenever a constraint function is satisfied, check whether it belongs to a cyclic constraint object;  
if not
  - remove the pointer to the constraint object from the execution set,
  - remove one pointer for each constraint object, which has an independent object which is also a dependent object of the just satisfied constraint object, from the blocked set,
  - each pointer which is removed from the blocked set and points to a cyclic constraint object, must be added to the cyclic set.
- if so
  - remove the pointer to the constraint object from the execution set,
  - add the pointer to the current cycle set,
  - add this pointer to the blocked set as often as the number of independent objects it has,
  - remove one pointer for each cyclic constraint object, which has an independent object which is also a dependent object of the just satisfied constraint object, from the blocked set,
  - each pointer which is removed from the blocked set must be added to the cyclic set.

This process loops around point 1-3 and only stops when the execution set is empty. In that case, all the constraint objects, which could be affected by the triggering independent object, are satisfied.

In the remainder of this section the different states are shown which the demonstration program would reach during constraint satisfaction according to the rules described above. This process shows how the MADE constraint system accomplishes the satisfaction of a fair amount of constraint objects in parallel while each (non-cyclic) constraint function is only executed once. First of all, the four satisfaction administration lists are created:

```
execution set = (constraintObjectSD, constraintObjectCFK, constraintObjectDCC, constraintObjectDCF,
                constraintObjectDCK)
block set =     (constraintObjectSD, constraintObjectCFK, constraintObjectCFK, constraintObjectCFK,
                constraintObjectCFK, constraintObjectDCC, constraintObjectDCC, constraintObjectDCF,
                constraintObjectDCK)
cyclic set =    ()
current cycle set = ()
```

These four sets are used to start the satisfaction process of the demonstration program. The first thing that is done, is to remove all the constraint objects from the *block set* for which *sliderInstance*, the triggering object, is an independent object:

```
execution set = (constraintObjectSD, constraintObjectCFK, constraintObjectDCC, constraintObjectDCF,
                constraintObjectDCK)
block set =     (constraintObjectCFK, constraintObjectCFK, constraintObjectCFK, constraintObjectCFK,
                constraintObjectDCC, constraintObjectDCC, constraintObjectDCF, constraintObjectDCK)
cyclic set =    ()
current cycle set = ()
```

At this point, constraint object *constraintObjectSD* may be satisfied, as this constraint object is contained in the *execution set* and not in the *block set* and because this constraint object is not cyclic. During the execution of the constraint function of this constraint object no other constraint function of another constraint object can be satisfied in parallel as all the constraint objects in the *execution set* are, according to the information in the CGO object, also contained in the *indirectly connected Constraint Objects* of the *directly connected Dependent Objects* of *constraintObjectSD*. Therefore, the constraint system has to wait for the constraint function of *constraintObjectSD* to finish before further actions can be taken.

Once the constraint function of *constraintObjectSD* has finished, the constraint objects *constraintObjectCFK* and *constraintObjectDCC* have been invalidated by the constraint function of *constraintObjectSD*, the constraint object *constraintObjectSD* is validated and the administration sets are updated:

```
execution set = (constraintObjectCFK, constraintObjectDCC, constraintObjectDCF, constraintObjectDCK)
/* remove constraintObjectSD from the execution set as its constraint function has been executed */
block set =     (constraintObjectCFK, constraintObjectCFK, constraintObjectCFK, constraintObjectDCC,
                constraintObjectDCF, constraintObjectDCK)
/* remove constraintObjectCFK from the block set as one of its independent objects (degreeInstanceC)
   is a dependent object of constraintObjectSD */
/* remove constraintObjectDCC from the block set as one of its independent objects
   (degreeInstanceC) is a dependent object of constraintObjectSD */
cyclic set =    (constraintObjectCFK)
/* add constraintObjectCFK to the cyclic set as this constraint object is cyclic and removed from block
   set */
current cycle set = ()
```

At this stage all constraint objects which are contained in the *execution set* are also blocked. Therefore, a constraint object from the cyclic set is chosen. Because the only choice is *constraintObjectCFK*, this constraint objects is the new *root* of the current cycle. The administration set is updated accordingly:

```

execution set = (constraintObjectCFK, constraintObjectDCC, constraintObjectDCF, constraintObjectDCK)
block set () = (constraintObjectDCC, constraintObjectDCF, constraintObjectDCK)
               /* remove all occurrences of the new root from the block set */
cyclic set = (constraintObjectCFK)
current cycle set = ()

```

Now, the constraint function *constraintFunction* of constraint object *constraintObjectCFK* may be satisfied. During the satisfaction of this constraint object, no other constraint objects can be satisfied in parallel as all the constraint objects in the *execution set* are, according to the information in the CGO object, also contained in the *indirectly connected Constraint Objects* of the *directly connected Dependent Objects* of *constraintObjectCFK*. Once the constraint function has finished its execution, the administration sets are updated. Note that the constraint objects *degreeInstanceF* and *degreeInstanceK* are invalidated:

```

execution set = (constraintObjectDCC, constraintObjectDCF, constraintObjectDCK)
               /* remove constraintObjectCFK from the execution set as its constraint function has been executed */
block set () = (constraintObjectCFK, constraintObjectCFK, constraintObjectCFK, constraintObjectDCC,
               constraintObjectDCF, constraintObjectDCK)
               /* add constraintObjectCFK to the block set as often as it has independent objects (3 in this case) */
cyclic set = (constraintObjectCFK)
current cycle set = (constraintObjectCFK)
               /* add constraintObjectCFK to the current cycle set */

```

At this moment no constraint objects can be satisfied as all the constraint objects in the *execution set* are blocked. However, because the *current cycle set* is not empty and the constraint object *constraintObjectCFK* is invalidated, a new iteration of the cycle is started. The invalidation of the constraint object was caused by calling the member functions *setTemp* of both, the *degreeInstanceF* and *degreeInstanceK* objects, in the constraint function of *constraintObjectCFK*. These functions are two triggering member functions of *constraintObjectCFK* which invalidated that same constraint object. In this situation the administration lists are updated:

```

execution set = (constraintObjectCFK, constraintObjectDCC, constraintObjectDCF, constraintObjectDCK)
               /* add current cycle set (i.e. constraintObjectCFK) to the execution set */
block set () = (constraintObjectDCC, constraintObjectDCF, constraintObjectDCK)
               /* remove the root (i.e. constraintObjectCFK) from block set */
cyclic set = ()
               /* remove current cycle set (i.e. constraintObjectCFK) from cyclic set */
current cycle set = ()
               /* make the current cycle set empty */

```

At this stage the constraint function of *constraintObjectCFK* can be executed again. Because this is the second iteration, the constraint function will only execute a *CycleBreak* command. So, *constraintObjectCFK* will be not be updated again. This leads to the following two updates of the administration lists (the constraint object *constraintObjectCFK* is now validated):

```

execution set = (constraintObjectDCC, constraintObjectDCF, constraintObjectDCK)
               /* remove constraintObjectCFK from the execution set as its constraint function has been executed */
block set () = (constraintObjectCFK, constraintObjectCFK, constraintObjectCFK, constraintObjectDCC,
               constraintObjectDCF, constraintObjectDCK)
               /* add constraintObjectCFK to the block set as often as it has independent objects (3 in this case) */
cyclic set = ()
current cycle set = (constraintObjectCFK)
               /* add constraintObjectCFK to the current cycle set */

```

```
execution set = (constraintObjectDCC, constraintObjectDCF, constraintObjectDCK)
block set () = ()
/* remove current cycle set (i.e. constraintObjectCFK) from the block set */
/* remove constraintObjectDCC from the block set as one of its independent objects
(degreeInstanceC) is a dependent object of constraintObjectCFK */
/* remove constraintObjectDCF from the block set as one of its independent objects (degreeInstanceF)
is a dependent object of constraintObjectCFK */
/* remove constraintObjectDCK from the block set as one of its independent objects
(degreeInstanceK) is a dependent object of constraintObjectCFK */

cyclic set = ()
current cycle set = ()
/* make the current cycle set empty */
```

At this moment the constraint functions of three constraint objects (*constraintObjectDCC*, *constraintObjectDCF* and *constraintObjectDCK*) can be solved in parallel; all three constraints objects are invalidated and none of the constraint objects in the *execution set* are, according to the information in the CGO object, also contained in the *indirectly connected Constraint Objects* of the *directly connected Dependent Objects* of either *constraintObjectDCC*, *constraintObjectDCF* or *constraintObjectDCK*.

After all three constraint functions have been executed, the constraint objects are validated. The administration lists contain the following information:

```
execution set = ()
/* remove constraintObjectDCC from the execution set as its constraint function has been executed */
/* remove constraintObjectDCF from the execution set as its constraint function has been executed */
/* remove constraintObjectDCK from the execution set as its constraint function has been executed */

block set () = ()
cyclic set = ()
current cycle set = ()
```

Now, because the *execution set* is empty, all the constraint objects, which were affected by the triggering of the constraint object, are satisfied and the propagation and satisfaction process has finished.

### 4.4.8 Satisfaction and Propagation in other Configurations

In this section, the same rules for constraint satisfaction and propagation are used as described as in the previous section. However, in this section, two different configurations are assumed. In the first configuration, the constraints objects *constraintObjectDCC*, *constraintObjectDCF* and *constraintObjectDCK* are considered to be active, but lazy. Therefore the information stored by the CGO object is somewhat different from the information described in the previous section. In this case all the information which was placed between '[ ]' in the table presented in §4.4.3.4.3, is removed from this table. The constraint network remains the same. The execution plan which is made is therefore somewhat different:

*constraintObjectSD* is directly connected to the eager constraint object *constraintObjectCFK*

*constraintObjectCFK* is directly connected to *degreeInstanceC*, *degreeInstanceF*, *degreeInstanceK*

*degreeInstanceC* is directly connected to the eager constraint object *constraintObjectCFK*

*degreeInstanceF* is directly connected to the eager constraint object *constraintObjectCFK*

*degreeInstanceK* is directly connected to the eager constraint object *constraintObjectCFK*

*constraintObjectCFK* already processed

*constraintObjectCFK* already processed

*constraintObjectCFK* already processed



Therefore, the execution plan becomes:

```
constraintObjectSD, constraintObjectCFK, constraintObjectCFK, constraintObjectCFK, constraintObjectCFK
```

The execution of the different constraint functions remains the same as in the previous section where only the last step, executing the constraint functions of *constraintObjectDCC*, *constraintObjectDCF* and *constraintObjectDCK* in parallel is omitted. For that reason, the values of the different degrees get updated, but the clocks will not represent this updated information. For that same reason, it is possible to press the *SHOW* - buttons to get the value of a temperature on the clock. When this button is pressed, the function *getClock* is invoked. This function is delegated (line 36, 40 and 44) to *getClockValue* of either *constraintObjectDCC*, *constraintObjectDCF* or *constraintObjectDCK*. One of these constraint objects is then renewed-triggered. If the constraint object is invalidated, the constraint function of the renewed-triggered constraint object is executed as yet. In that case the constraint object is also validated. In case the constraint object was already valid, the constraint function is not executed (as there is no need to). Note that in this demonstration program the renewed triggering actions of one of these objects does not trigger other constraint objects as none of the objects *clockInstanceC*, *clockInstanceF* and *clockInstanceK* is an independent object to any constraint object.

In the second configuration it is assumed that the constraints objects *constraintObjectDCC*, *constraintObjectDCF* and *constraintObjectDCK* are inactive; they will not be considered during the constraint propagation and satisfaction process. In that case it also of no importance whether the constraints are lazy or eager. The information stored by the constraint system is equal to that described in §4.4.3.4.2.

The execution plan and the actions taken by the constraint system are identical to the ones described in the first configuration of this section. The only difference lies in the actions performed when one of the *SHOW* - button is pressed. In that case, nothing happens. As the constraint objects *constraintObjectDCC*, *constraintObjectDCF* and *constraintObjectDCK* are inactive they cannot be renewed-triggered; the functions *getClock* are not delegated to the shadow functions anymore. Invocation of these functions will thus only result in the execution of these functions without the intervention of constraint objects or constraint system.

Note that in both configurations the work that has to be done by the constraint system is minimized; lazy constraints are not satisfied until an explicit request is made (the invocation of a renewed-triggering member function) and inactive constraint objects will stop the propagation of information in the constraint network as the constraint relation does not have to be maintained (at the moment).

#### 4.4.9 The Constraint Network and the Database

The MADE constraint system makes it possible to store and retrieve individual constraint objects as well as the complete constraint networks in/from a database. As there is a need to store multimedia presentations it must also be possible to store the (state of the) constraint network; therefore the MADE constraint network has been designed in such a way that storage and retrieval of (part of) the constraint network is possible. The MADE environment uses the following terminology with respect to the database:

- **save:**  
store status information of the different MADE objects in the database.
- **retrieve:**  
update the status information of a MADE object with the information stored in the database.



◦ **restore:**

create a new **MADE** object and fill its status information with the information stored in the database.

When an individual constraint object is stored, its dependent and independent links are stored as well. After an object is stored, the database object will return a unique identifier, the *MOid*. This *MOid* can be used when a constraint object has to be restored or retrieved.

The information stored in the database is the information in the snaptree of the CO object and the information added to the snaptree for each individual constraint class. The following information from the snaptree of the constraint object completely describes the state of a constraint object:

Valid	the flag to indicate whether the constraint object is valid or not
UserActive	the flag to indicate whether the constraint object is active or not
Lazy	the flag to indicate if the constraint object is lazy or eager
Cyclic	the flag to indicate whether the constraint object lies on a cycle
dboCf	the signature of the active constraint function
dboIndeps	a list with information regarding the independent objects of the constraint object
dboDeps	a list with information regarding the dependent objects of the constraint object
dboMapping	a signature for identification of a constraint object

The item *dboIndeps* points to a list in the database. An item in this list is constructed for every combination (independent object, triggering member function). Each item in the list contains the following information:

dtrm	the signature of the triggering member function
dtem	the signature of the independent shadow function
oid	an identifier for the independent object
noid	an identifier for the next item on the list

Similar, *dboDeps* points to a list with an item for every combination of (dependent object, renewed-triggering member function):

dtrm	the signature of the renewed-triggering member function
dtem	the signature of the dependent shadow function
oid	an identifier for the dependent object
noid	an identifier for next item in the list

Using the information as described above, it is possible to restore the state of a constraint object. In case an individual constraint object is retrieved, first all its existing dependent and independent

links are removed before the status information for this constraint object is restored. After that, new links are created. In case an individual constraint object is restored, a new constraint object is created which has no existing links and therefore the new links can be created immediately.

The storage and retrieval of a constraint network is nothing more than the sequential storage and retrieval of the individual constraint objects in the constraint network. Independent and dependent object are not stored when a constraint network is stored because the constraint system is not allowed to change the status of the (in)dependent objects (when the constraint network is retrieved/restored) without explicit knowledge of the application. Therefore, the (in)dependent objects of a constraint network cannot be retrieved/restored by the constraint system itself; as a result of that there is no point in storing anything else but a reference to these objects when the constraint system is stored.

For the storage (as well as the retrieval and restoration) of the constraint network, in principle no extra information has to be stored which is not already stored by individual constraint objects; from the information stored by individual constraint objects, the complete constraint network can be reconstructed. However, it is necessary to know which constraint objects, stored in the database, belong to the constraint network. Therefore, a set of all the constraint objects in the constraint network is stored in the database:

dboSet	a set of all constraint objects in the current constraint network
--------	---

The restoration of a constraint network introduces a new problem. Once a constraint network is restored, the application has no handles to the different individual constraint objects. This might not always be necessary, but sometimes it is desirable to get control over the individual constraint objects again. For this purpose, special constructs are available which allow an application to define handles which can be used upon restoration of the constraint network (compare *dboMapping* in the snaptree of the constraint object).

#### 4.4.10 Notification of Constraint Satisfaction

The MADE constraint system also provides the ability to obtain information about the satisfaction process other than via the individual constraint objects. Because many details of the constraint satisfaction process are hidden by the MADE constraint system, additional functions have to be provided to retrieve some information about this process again in a controlled manner. A example of information of the satisfaction process which can be important to know is whether information propagation has already reached a certain object in the constraint network. Due to the fact that constraint satisfaction may be done in parallel by the MADE constraint system and because it is hard to predict the order in which certain objects are satisfied, the only way to get this kind of information is to ask the constraint system.

For this purpose, two functions are defined: *notify* and *denotify*. Once the *notify* function is invoked, the constraint system will increase the value of a variable (passed as an argument) every time (part of) the constraint network is satisfied in which a particular object (also passed as an argument) was contained. This behavior (of updating the variable) takes effect immediately; if the constraint system is busy satisfying (part of) the constraint network while the function *notify* is invoked and if during that satisfaction process the indicated object is used as a dependent object, the value of the specified variable is already incremented. The function *denotify* will cancel the behavior introduced by *notify*.

These functions allow a program to check whether a change to a particular (independent) object has been propagated to a specific object. By checking the value of the variable, it is possible to wait

until a constraint object has updated a particular object and perform the desired action only after that moment. The choice to use a variable instead of a member function invocation of the object which watches the state of the information propagation in the constraint network is based on the following observations: the value of the variable can be checked at any time (thus also at a later, more convenient moment) whereas only the message of an active object can be sampled. This means that, by using a variable, all objects (including Mutex, Unprotected and C++ objects) can use the information with regard to constraint satisfaction in a sampled way. If desired, also a busy-loop can be created to act on the satisfaction (of part) of the constraint network as soon as the value of the variable changes.

### 4.5 Performance of the MADE Constraint System

In the section on the rationales for creating the MADE environment (§4.1), it was already mentioned that mC++ is based on C++. In practice, the relation between mC++ and C++ is even closer; every mC++ program is translated (by the mC++ translator) into an equivalent C++ program before it is given as input to a C++ compiler. During this translation, the mC++ translator makes sure that all mC++ specific features (i.e. the features which are not available in C++ such as delegation, active objects, etc.) are mapped to (in some cases by the translator newly created) C++ functions and C++ objects such that the semantics of the original mC++ program are preserved in the new C++ program.

To be able to say anything about the performance of the constraint system, it is necessary to understand how this translation of the mC++ translator takes places with respect to active objects, delegation and the dynamic call interface (the three most important mC++ specific features which are used by the constraint system). After this translation scheme is explained in §4.5.1, the execution plan from §4.4.7.2 will be used in §4.5.2 to discuss the overhead and performance of the constraint system in general. It is important to note that the performance of the constraint system will primarily be measured in the number of C++ function calls that are needed to satisfy the constraint network. This measurement unit is chosen because it can be used as an indication for the complexity of a process. The reason not to chose a time unit has to do with the fact that there is no real mC++ compiler and thus time measurements on the executable representation of the mC++ program will not lead to representative comparisons. Furthermore, the MADE constraint system is not an integral part of the language mC++, but is programmed, using mC++, as a toolkit. This again would lead to time measurements which cannot really be compared with languages where the constraint system is an integral part of that language.

#### 4.5.1 The Translation from mC++ to C++

This section explains part of the translation process of a mC++ program to a C++ program. In the text below, only the aspects with respect to active objects, delegation and the dynamic call interface will be discussed in detail. During the translation, mC++ code is transformed into C++ code and specialized macros are added to the text by the mC++ translator where appropriate. The resulting C++ program (i.e. the source code after the mC++ translator has finished its translation) makes use of a number of such macros. These macros are part of the so-called MADE C++ API. More details on this C++ API can be found in [Arbab et al. 93a]. To make the process of translation clearer, the following code, which declares an active object, will be taken as starting point:

```
Active AnArbitraryActiveObject {  
    ...                                     // constructor functions, destructor functions, main function, etc.  
}
```

```

public:
    char aPublicMemberFunction (float argY);

    ...

};

char AnArbitraryActiveObject::aPublicMemberFunction (float argY) {
    ...
};

```

The first step of the translation from a mC++ program to a C++ program consist of adding a number of macros to the class declaration of `AnArbitraryActiveObject`. These macros (typeset in bold) will, in a next step, create additional auxiliary functions to support things like message delegation, message priorities, sampled messages, dynamic call support, etc.:

```

class AnArbitraryActiveObject: virtual public MActive {
    MDECL_ACTIVE_CLASS(AnArbitraryActiveObject);

public :

    char aPublicMemberFunction ( float argY );

    MDECL_DACTIVE(AnArbitraryActiveObject, char, aPublicMemberFunction,
        MCchar_aPublicMemberFunction__float_, 1, (float ), 0);
};

MIMPL_ACTIVE_CLASS_START(AnArbitraryActiveObject, 1, (MActive ))
    MINIT_MSG(AnArbitraryActiveObject, aPublicMemberFunction, MCchar_aPublicMemberFunction__float_);
MIMPL_ACTIVE_CLASS_END(AnArbitraryActiveObject, 1, (MActive ))

```

At the beginning of the class declaration, a macro is inserted which takes care of the declaration of several functions, variables and tables which are needed by the (active) class to support the special mC++ features. Furthermore a macro is added to the class declaration for every (proto) member function that is encountered by the mC++ translator. These latter macros are needed to define several auxiliary functions to make it possible that a mC++ member function can be delegated, can be used in a dynamic call, can be used in a sampled way or can be assigned a priority with respect to the servicing of the message. Outside the class declaration additional macros are inserted which take care of the initialization of the different member functions and their auxiliary functions.

The different auxiliary functions do not only have to be declared, they also have to be defined. Therefore, a macro is inserted in the C++ code for every member function definition that is encountered in the mC++ code. Note that in the code below the name of the original member function is preceded by 'M\_'. The macro will define another function which has the same signature (i.e., name and type of parameters) as the original member function, which serves as a shell around the original member function and will invoke `M_aPublicMemberFunction`. This approach provides the mC++ runtime code with hooks on the member function to take care of delegation, dynamic calls, priorities etc. Furthermore, two arguments are added to the argument list of the original member function: `MObject* Delegator` and `MMsgID dtorMsg`. These two arguments correspond to *Delegator* and *Delegator\_function* as described in §4.2.3.2:

```

MIMPL_DACTIVE(AnArbitraryActiveObject, (char), aPublicMemberFunction,
    MCchar_aPublicMemberFunction__float_, 1, (float ), ((double)));

```

```
char AnArbitraryActiveObject::M_aPublicMemberFunction(MObject *Delegator, MMsgID dtorMsg, float argY ) {
    If (Delegator); if (dtorMsg); //dummy stats to suppress warnings
    ...
};
```

In the next step, each of these macros is expanded by the C++ compiler. During this expansion several auxiliary member functions are created. The names of the auxiliary functions are derived from the original member function by prepending it with a unique prefix (the name of this prefix depends on the task of the auxiliary function) and the name of the class and by appending an encoded parameter list. In general, the following auxiliary functions are created:

- void MCIniter\_*ClassName*\_**MemberFunctionName**\_*EncodedParameterList* (
 

MObject\* localProto)

This auxiliary function will initialize several tables and flags which have to be set up for the other auxiliary functions. These tables are defined for the prototype of *ClassName* which is pointed to by *localProto*.

- ... **MemberFunctionName** (...)

This auxiliary function replaces the original member function (which was renamed *M\_MemberFunctionName*) and serves as a hook for the delegation. This function first tests whether the function is delegated or not. If so, the delegatee is called. Otherwise *MCCaller\_ClassName\_MemberFunctionName\_EncodedParameterList* is called (and, indirectly, the original member function).

- ... MCCaller\_*ClassName*\_**MemberFunctionName**\_*EncodedParameterList* (void\* voidObj,
 

MObject\* dtor, void\*, MMsgID dtorMsg, ...)

This auxiliary function is used to guarantee the mutual exclusive access of member functions of either an active or a mutex object. *voidObj* points to the object which member function *MemberFunctionName* should be called, *dtor* points to the delegator (if present) and *dtorMsg* points to the original member function which was delegated. The remaining arguments are the same as those of the original member function *MemberFunctionName*. When no other member function of *voidObj* is entered (mutual exclusive access), this function will invoke *M\_MemberFunctionName*.

- MBool MCGuardCaller\_*ClassName*\_**MemberFunctionName**\_*EncodedParameterList* (
 

MObject\* o)

This auxiliary function is used to evaluate the guards of a *Sync* section of a mutex object *o* to determine whether the original function *MemberFunctionName* may be called or not.

- void MCDynCaller\_*ClassName*\_**MemberFunctionName**\_*EncodedParameterList* (
 

MObject\* o, va\_list args)

This auxiliary function is used when the original member function *MemberFunctionName* is called using the dynamic call interface. The purpose of this function is to extract the arguments in *args* and pass them on to the original function of object *o*.

- void MCSetDelegFlag\_**MemberFunctionName**\_*EncodedParameterList* (MObject\* o,
 

MBool set)

This auxiliary function is used to set a flag to indicate (*set*) whether or not *MemberFunctionName* of object *o* should be delegated or not.

With respect to the running example, the following code would result after macro expansion:



```

void MCDynCaller_AnArbitraryActiveObject_aPublicMemberFunction_MCchar_aPublicMemberFunction__float_ (
    MObject* o, va_list args) {
    ...                                     // extract argument from args
    if (retPtr)
        *retPtr = obj->aPublicMemberFunction((float)arg0);           // (indirectly) call original mC++ member function
    else
        (void) obj->aPublicMemberFunction((float)arg0);               // (indirectly) call original mC++ member function
};

char AnArbitraryActiveObject::aPublicMemberFunction (float arg0) {
    if (MChasDeleg_aPublicMemberFunction_MCchar_aPublicMemberFunction__float_) { // If mC++ function is delegated
        ...
        return dteCaller(dte, (MObject*)this, (void*)this,           // call delegatee
            "AnArbitraryActiveObject_aPublicMemberFunction" , arg0 );
    }
    return MCCaller_AnArbitraryActiveObject_aPublicMemberFunction_MCchar_aPublicMemberFunction__float_ (
        (void*)this, (MObject*)this, (void*)this,                   // else (indirectly) call original
        "AnArbitraryActiveObject_aPublicMemberFunction" , arg0 );   // mC++ member function
};

char MCCaller_AnArbitraryActiveObject_aPublicMemberFunction_MCchar_aPublicMemberFunction__float_ (
    void* voidObj, MObject* dtor, void*, MMsgID dtorMsg , float arg0 ) {
    ...
    MBool doLeave = obj->MC_Enter(obj,                                // check on mutual exclusive access
        "AnArbitraryActiveObject_aPublicMemberFunction");           // and block semaphore for access
    char retValue = obj->M_aPublicMemberFunction ( dtor, dtorMsg , arg0 ); // call original mC++ member function
    if (doLeave) obj->MC_Leave();                                         // release semaphore for access
    return retValue;
};

char AnArbitraryActiveObject::M_aPublicMemberFunction(MObject *Delegator, MMsgID dtorMsg, float argY) {
    if (Delegator); if (dtorMsg);
    ...
};

void AnArbitraryActiveObject::
    MCIniter_AnArbitraryActiveObject_aPublicMemberFunction_MCchar_aPublicMemberFunction__float_ (
        MObject* localProto) {
    ...                                     // initialize tables of object
};

void MCSetDelegFlag_AnArbitraryActiveObject_aPublicMemberFunction_MCchar_aPublicMemberFunction__float_ (
    MObject* o, MBool set) {
    ...                                     // set delegation flag
};

```

Given the set of auxiliary functions described above, a number of remarks with respect to the performance of the mC++ runtime system can be made. These remarks are important for the explanation in the next section in which the performance of an arbitrary mC++ program is determined.

- invoking an ordinary mC++ member function which is not delegated will lead to the calling of three C++ functions:

```

Obj::aPublicMemberFunction (3.2)
Obj::MCCaller_..._aPublicMemberFunction_... (Obj, Obj, Obj, "aPublicMemberFunction", 3.2)
Obj::M_aPublicMemberFunction (Obj, "aPublicMemberFunction", 3.2)

```

- invoking a mC++ member function which is delegated to another function (assume this is `Dte::theDelegateeMemberFunction`) will also lead to the calling of three C++ functions. Important to note here is that, if a member function A is delegated to another member function B which, in its turn, is delegated to another member function C, A will directly call the delegated member function C (instead of B). This means that a delegation will always result in three C++ function invocations regardless whether or not the delegatee is delegated itself.

```
Obj::aPublicMemberFunction (3.2)
Dte::MCCaller_..._theDelegateeMemberFunction_... (Dte, Obj, Obj, "aPublicMemberFunction", 3.2)
Dte::M_theDelegateeMemberFunction (Obj, "aPublicMemberFunction", 3.2)
```

- invoking a mC++ member function using the Dynamic Call Interface will always result in the invocation of `MCDynCaller...`. As this function will always call upon the function with the original signature, the number of C++ function invocations will always be four (one + the number of invocations when the function would be invoked by directly calling it).

```
Obj::MCDynCaller_..._aPublicMemberFunction_... (3.2)
Obj::aPublicMemberFunction (3.2)
...
```

- setting up a delegation will cost one C++ function invocation, namely the invocation of the function `MCSetDelegFlag_ClassName_MemberFunctionName_EncodedParameterList`; the overhead is thus one C++ function invocation.
- defining a mC++ class will always result in the invocation of the C++ function `MCIniter_ClassName_MemberFunctionName_EncodedParameterList` for each member function declared in the object type; the overhead of defining mC++ classes in terms of C++ function invocations thus equals the number of member functions defined for that class.

### 4.5.2 The Performance in a MADE Constraint Network

With respect to the performance of the MADE constraint system a distinction can be made between the several operations that can be performed on the constraint network:

- constraint classes are declared
- constraint relations are added to the constraint network
- a triggering or renewed-triggering action is performed on the constraint system
- the constraint network is satisfied

Especially the fourth operation is really interesting with respect to the performance. However, other aspects are discussed here as well to be complete.

When constraint classes are defined the normal overhead with respect to class initialization exists. Additionally, when a constraint class is declared also the (in)dependent shadow functions have to be declared and each shadow function will result in the invocation of an initializing C++ function (prefix 'MCIniter').

When a constraint relation is added to the constraint network, the independent objects, the dependent objects and the constraint object have to be registered with the constraint system. For each (in)dependent object one call is made to the CGO object, one to the ROUTER object and one dynamic call is made to the constraint object. Each of these functions perform some initialization instructions and the time needed to complete these should be minimal. Finally, the delegation has to be set up which means that for each independent object and each dependent object an additional C++ function call has to be made to set the delegation flag. If it is assumed that  $I$  is the number of independent objects and  $D$  the number of dependent objects involved in the constraint relation then the overhead of adding a constraint relation to the constraint network is  $(3+3+4+1)*I+(3+3+4+1)*D+3$  C++ function invocations.

Upon the triggering of the constraint system (see also §4.4.5), a delegated mC++ member function is invoked. The delegatee will invoke the original (delegated) mC++ member function and

then call upon a mC++ member function of the MCGO object which will invoke a mC++ member function of a CGO object. The CGO object will perform a number of direct calls and a number of dynamic calls: for each constraint object for which the triggering object is an independent object, three direct calls are performed and two dynamic calls. If it is assumed that T is the number of constraint objects for which the triggering object is an independent object, the total cost comes to  $4+3+3+3+(3*3+2*4)*T$  C++ function invocations. This means that the overhead due to the constraint system is 3 C++ invocations less (the original member function had to be called anyway):  $10+17*T$  C++ function invocations.

When a renewed-triggering action is performed (see also §4.4.6), a similar situation arises as when the constraint system is triggered (only the order in which certain things happen is different). The only difference between a triggering action and a renewed-triggering action (with respect to overhead) is that during a renewed-triggering action, the renewed-triggering object waits for the constraint network to be satisfied and that for each constraint object for which the renewed-triggering object is a dependent object four direct calls are performed and two dynamic calls. The performance of the constraint system with respect to the satisfaction of the constraint network is discussed next. For now, it is assumed that this will take S C++ function invocations. If it is assumed that R is the number of constraint objects for which the renewed-triggering object is a dependent object, the total cost comes to  $4+3+3+3+(4*3+2*4)*R+S$  C++ function invocations. This means that the overhead due to the constraint system is 3 C++ invocations less (the original member function had to be called anyway):  $10+20*R+S$  C++ function invocations.

Finally, this paragraph discusses the performance for the satisfaction of the constraint network. It is not always necessary to satisfy all the constraint objects in the constraint network. This depends on which (in)dependent object has performed a (renewed-)triggering action on the constraint network. This should be taken into account when discussing the performance of the constraint system. A lot of the time used by the constraint system to satisfy the constraint network is put in maintaining the different administration lists as discussed in §4.4.7.3. Therefore, the time spent doing this administration gets important with respect to the performance of the satisfaction process. Each non-cyclic constraint object is added to and removed from the *execution set* only once. The number of times such an object is added to and removed from the *block set* is equal to the number of independent objects that are defined for the constraint object. For cyclic constraint objects the number of times the constraint object can be added to and removed from the *execution set* can be more than once; it equals the number of times the cycle on which the constraint object lies is traversed. Also the times a cyclic constraint object is added to and removed from the *blocked set* is larger than in the case of the non-cyclic constraint object; this number is less than the number of independent objects of the constraint object multiplied by the number of times the cycle is traversed. Furthermore, every cyclic constraint object is added to and removed from the *cyclic set* once and it is added to and removed from the *current cycle set* as often as the cycle it traversed. In total, this brings the total overhead of administration to :

$$\sum_{(C|connected_O(C))} 2 * (cycleTraversal(C) * (1 + indep(C)) + cyclic(C) * (1 + cycleTraversal(C)))$$

where:

connected <sub>O</sub> (C)	= 0,	if there exists no path from O to C
	= 1,	otherwise
cyclic(C)	= 0,	if C is a non-cyclic constraint object
	= 1,	otherwise
cycleTraversal(C)	= 1,	if cyclic(C) = 0
	= CT,	if cyclic(C) = 1 and CT equals the number of traversals of the cycle



$\text{indep}(C) = \text{\#independent objects of } C$

During the constraint network satisfaction each constraint object is satisfied once. When a constraint is satisfied, a `CO_Wrap` object is created and this object will perform six direct mC++ member function calls (to update the administration of the constraint objects) and one dynamic mC++ member function call (to invoke the constraint function). As the `CO_Wrap` object has only one member function, the total overhead for the execution of one constraint function becomes  $1+6*3+1*4$  C++ function calls. Important to note here is that the complexity of the constraint function itself is unknown and could be important for total performance of the satisfaction of the constraint network. This factor will be described as  $CF_C$  for an arbitrary constraint object  $C$ . The total costs to satisfy a constraint network are thus:

$$S = \sum_{(C \text{ connected to } O)} 2 * (\text{cycleTraversal}(C) * (1 + \text{indep}(C)) + \text{cyclic}(C) * (1 + \text{cycleTraversal}(C))) + 23 + CF_C$$

When the performance analysis as described above is applied to the example of §4.4.7.3, the following results appear:

definition constraintObjectSD	5 C++ function calls
definition constraintObjectCFK	6 C++ function calls
definition constraintObjectDCC	5 C++ function calls
definition constraintObjectDCF	5 C++ function calls
definition constraintObjectDCK	5 C++ function calls
adding constraint relation for constraintObjectSD (I = 1, D = 1)	25 C++ function calls
adding constraint relation for constraintObjectCFK (I = 3, D = 3)	69 C++ function calls
adding constraint relation for constraintObjectDCC (I = 1, D = 1)	25 C++ function calls
adding constraint relation for constraintObjectDCF (I = 1, D = 1)	25 C++ function calls
adding constraint relation for constraintObjectDCK (I = 1, D = 1)	25 C++ function calls
triggering via sliderInstance (T = 1)	27 C++ function calls

satisfaction of constraint network (O = sliderInstance)

```

indep (constraintObjectSD)      1
cyclic (constraintObjectSD)     0
cycleTraversal (constraintObjectSD) 1
---
                                4 list operations

indep (constraintObjectCFK)     3
cyclic (constraintObjectCFK)    1
cycleTraversal (constraintObjectCFK) 1
---
                                12 list operations

indep (constraintObjectDCC)     1
cyclic (constraintObjectDCC)    0
cycleTraversal (constraintObjectDCC) 1
---
                                4 list operations

indep (constraintObjectDCF)     1
cyclic (constraintObjectDCF)    0
cycleTraversal (constraintObjectDCF) 1

```

	---
	4 list operations
indep (constraintObjectDCK)	1
cyclic (constraintObjectDCK)	0
cycleTraversal (constraintObjectDCK)	1
	---
	4 list operations
	28 list operations
	115 C++ function calls
CF_constraintObjectBD	6
CF_constraintObjectCPK	9
CF_constraintObjectDCC	6
CF_constraintObjectDCF	6
CF_constraintObjectDCK	6
	---
	33 C++ function calls
	28 list operations
	148 C++ function calls
	----
	28 list operations
	370 C++ function calls

A final remark has to be made with respect to these numbers. Because some constraint objects were satisfied in parallel, the constraint system did not cause an overhead of 370 sequential executed C++ function calls. The actual overhead in terms of time not only depends on the number of additional C++ function calls, the thread-package that runs underneath MADE and on the platform on which the program is executed, but also on other things which are very hard to formalize like workload of the machine. This makes it very hard to present results which are more concrete than the ones presented above.

Furthermore, efficiency was not a main design issue; the approach taken in the MADE constraint system may not be the most efficient one. The main purpose was to design a system which was elegant from the perspective of the programmer.

#### 4.6 Conclusions

In this chapter, the MADE constraint system was discussed. The concepts and terminology of the constraint system were described and the three important subsystems (triggering, propagation and satisfaction) of a constraint system were explained in relation to the MADE constraint system.

An important aspect of the triggering subsystem is the fact that the triggering of the constraint object is done by delegation. This allows for ordinary MADE objects to become independent or dependent objects without the need to change anything in the code for declaration and definition of their class. This not only means that the objects which are to be constrained do not have to be written specifically for that purpose, it also means that constraint triggering, propagation and satisfaction is done transparent to the application.

To keep the (negative) effect of the constraint system on the main application as small as possible, the satisfaction of the different constraints is done in parallel with the application as much as possible. For that reason active objects were introduced, which performed different constraint management tasks in their own thread of control. In this way, the MADE constraint system tries to perform those tasks, which are independent of each other, as separate of each other as much as

possible. However, a constraint system will also have some negative influences on the performance of the application; a constraint system is a general tool and its administration tasks will introduce overhead costs which may not be present if the solution to a problem is explicitly programmed in the application. However, in the latter case, a lot of flexibility is lost; the application can only be used in the situation it was designed for and adaptations to the code may turn out to be more complicated than the changing of a few constraint relations would have been.

In §4.5 the performance of the constraint system was discussed in more detail. The unit of measurement that was chosen here was the number of C++ function calls. It was argued that the number of C++ calls could be seen as an indication for the complexity of the algorithm and thus also for the time needed to execute the algorithm. In the same section, it was computed how many C++ functions calls were needed to create, trigger and satisfy the constraint network of the example program. Looking at the satisfaction process alone (which is the most interesting part of the complexity of the constraint system), it took 148 C++ function calls and 28 list operations. When the solution to the problem would have been programmed explicitly in the application, maintaining the relations between Celsius, Fahrenheit and Kelvin would only have taken 18 C++ calls (one mC++ call (and thus 3 C++ function calls) for each of the following tasks: set the value of Celsius, set the value of Fahrenheit, set the value of Kelvin, update the value of the Celsius clock, update the value of the Fahrenheit clock and update the value of the Kelvin clock). This means that the number of C++ function calls is increased by a factor 8. In return, more flexibility (other temperature systems can be added easily), a cleaner code (all aspects with regard to the relations between Celsius, Fahrenheit and Kelvin are concentrated in one place where the constraints are defined) and a higher level of abstraction on which the solution for the problem can be specified are obtained. With the ever increasing speed of computers today, the price to pay for these advantages seems to be acceptable.

Most of the design of the MADE constraint system was determined by the fact that the constraint system had to be embedded in the object-oriented language mC++ and that it was used in a multimedia environment. This meant that four different, sometimes even contradicting, paradigms had to be combined:

- **the Constraint Programming Paradigm**

This paradigm stands for two things; it assumes that constraint objects have access to all the information of other objects (otherwise it is possible that constraints cannot be satisfied due to lack of access to the information) and it assumes a declarative description of the solution for a constraint problem.

- **the Object-Oriented Paradigm**

This paradigm imposes that information is hidden in the different objects. In C++ (and thus also in mC++) special constructs exist which allow special objects to access private data of another object, but in general, the access to the data of a particular object is restricted.

- **the Imperative Programming Paradigm**

Because mC++ is an imperative programming language, in the end, instructions for the constraint system have to be imperative too.

- **the Multimedia Paradigm**

This paradigm is not really an existing paradigm, but it is used here to express the idea that multimedia applications are very diverse in what they do and equally diverse in the requirements they impose on a constraint system. One of the most important requirements is the ability to interactively change the way in which constraints are satisfied and which constraints are satisfied.

In the MADE constraint system, most of these paradigms are brought together in one way or another. This can be found in the following characteristics of the system:

- Constraints are only allowed on the execution of an object's member function. This rule preserves the information hiding (inherited from the Object-Oriented Paradigm), but still allows for constraint satisfaction. Constraints on the object's private data have to use a member function which sets and gets its value.
- The MADE constraint system allows the creation of constraint classes. These constraint classes can be seen as a declarative description of a constraint problem. The instances of the constraint class hide how (i.e. the imperative part) a constraint is implemented, but are an abstraction of what the constraint does (the declarative aspect).
- The ability to design and write new constraint classes allows programmers to tackle any constraint problem which might occur in multimedia environments. This, together with the special constructs, which are available to the constraint programmer so he can write his own constraint functions, gives the MADE constraint system its flexibility and allows the writing of any specialized constraint solution that is desired.
- It is possible to, dynamically, change the status of the individual constraint objects as well as the structure of the constraint network.



# CHAPTER 5

## The Quality Factory



In this chapter the concepts behind the Quality Factory will be described. This Quality Factory will be based upon the MADE constraint system. The concepts of this constraint system have been described and an example of its functioning has been given in Chapter 4.

Users of multimedia applications wanting to interact with the ongoing presentation, are often faced with the fact that they have to express their requirements in terms which are not familiar to them. Especially when they have quality requirements with respect to the presentation, they are often faced with the fact that there is a large discrepancy between the way in which the application manages its data and the way in which this data is perceived by the user. Examples of this were given in Chapter 2. Furthermore, it is difficult to expand existing applications with new versions of existing functionality or to add new functionality at all. This means that users of existing applications cannot always make use of existing new technology which means that they cannot always get the highest possible quality that can, in theory, be provided. For these kind of situations, the Quality Factory may provide a solution.

The Quality Factory is introduced to intercept quality requests of the user of a multimedia application and to translate the request in terms of this user's perception into terms of the underlying systems which can provide the required services. One of the functions of the Quality Factory is to provide negotiation facilities to allow negotiation over the usage of the available resources based upon the availability of these resources and on the quality requests made in the past. Note that the Quality Factory does not perform the negotiation itself; special negotiation objects are supported which then can negotiate over resource allocation based on predefined configurations. For this purpose, the Quality Factory will take over the original request, analyze it, transform it into more appropriate requests for the underlying systems, allow the negotiation objects to do their negotiation, issue this request on behalf of the user and return the results, on behalf of the underlying systems, to the user. Hence the name, Quality Factory, in analogy with the term 'Object Factory' which is used for systems which monitor the life-cycle of objects.

Important notions for the Quality Factory are services, filters, formats and resources. These notions guide the Quality Factory in translating requirements in terms of the user's perception to requirements which are detailed and specific enough for existing systems to understand. The most important notion for a Quality Factory is the *resource*. The sole purpose of existence of the Quality Factory is that resources often need to be shared between different applications and often are scarce. The Quality Factory acts as broker between the different requests for resources and the resources' managers. The thing that sets the Quality Factory apart from existing Quality of Service systems is the fact that the Quality Factory does not interact with the application (and thus the user<sup>1</sup>) in terms of availability or reservation of the resources but in terms of the user's perceived quality. Requests of perceived quality are translated by the Quality Factory to requests in terms of resources.

In a sense, the Quality Factory itself is a service provider; it provides services to the application. For this purpose, it uses other service providers and so-called filters; the Quality Factory acts as a shell around them to provide new and possibly other services. It can be said that the Quality Factory reuses the service providers and filters to generate more abstract services for the viewer. In order to do so, the Quality Factory makes use of a two-level structure:

- data level

the first level is the data level; this level describes which data has to be managed by the Quality Factory and how this data is used by services and filters.

---

<sup>1</sup> The user of a multimedia application is also the viewer of that same application. The terms 'viewer' and 'user' will be used interchangeably throughout this text.



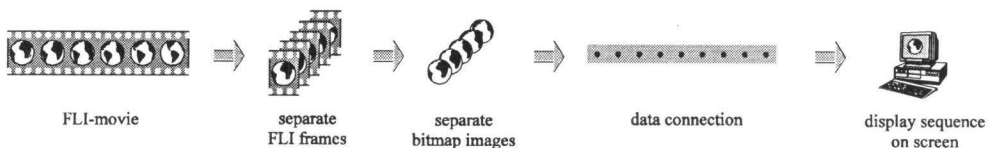
- context level

this level describes how the different filters work together and what kind of configurations can be specified in the Quality Factory using the underlying constraint system. The negotiation objects can be used to manage the different configurations.

Using the Quality Factory problems can be solved which are based on fixed relations; problems which need an extensive search process to find a relation cannot be tackled by the Quality Factory. An example which can make this difference clear is this: the Quality Factory cannot be used to find the optimal route by train (as for distance, costs or number of transfers) from station A to station B given a list of stations and a timetable (there is no fixed relation which describes how to get from A to B), but it can determine the time of arrival at station B when station A is left at time X given the route and the timetable (there are fixed relations between the times of departure and arrival for each pair of stations). The Quality Factory is even capable of updating the arrival time given some changes in timetables and/or followed route during the journey.

In the remainder of this chapter, the following topics will be discussed: the notions of services (§5.1), filters (§5.2), formats (§5.3) and resources and negotiation (§5.4). In §5.5 the similarities between the Quality Factory and software reuse will be discussed. In the §5.6 and §5.7 the implementation of the Quality Factory will be explained and the role of the MADE constraint system in this implementation. §5.8 discusses the performance of the Quality Factory and in §5.9 some conclusions will be drawn, based on what was discussed in the previous paragraphs.

Throughout this chapter, the text will be accompanied by examples of a demonstration program. This demonstration program is merely a toy-example which is chosen to be fairly simple in order to keep the explanation simple. In this example, at one location a movie in 'FLI' format is stored which contains a sequence of pictures which shows the earth turning around its own axis. This movie has to be sent over a certain data-connection to another physical location where it can be displayed. The problem to solve in this example is that the format 'FLI' cannot be displayed by the destined computer; it can only display 'XBM' formatted images. Therefore, a filter has to be inserted which transforms the 'FLI' images into sequences of 'XBM' formatted images. For this process, the individual frames of the 'FLI' movie have to be read, the image has to be grabbed from each frame (the actual image has a size of only 64x64 pixels whereas a normal 'FLI' movie has a size of 320x200 pixels) and it then has to be converted into the 'XBM' format. Furthermore, the user may specify the rounds per minute the globe is spinning and the level of detail at which the images are shown. All these requirements have to be translated into requirements with respect to the data-connection over which the data has to be sent and may contradict each other on this 'translated' level. Graphically, the situation will look as follows:



### 5.1 Services

A central notion in the Quality Factory is that of *service*. A service is something which can be requested by the application (indirectly this may be the user of the application) and which can be provided by different components. The software component which provides one or more services is

said to be a *service provider*. How these services are implemented and which resources are necessary to provide these services is only known to the service provider. It is even possible that one service provider is capable of supporting different versions of the same service (i.e. services with the same name which are controlled in a slightly different way or which support slightly different functionalities) at the same time and that different service providers provide the same service. Furthermore, it is possible that, during the execution of the application, a service or even complete service providers appear and disappear. This means that service providers can register services with the Quality Factory and that they can withdraw services. In other words: services need to be managed!

An application may make a *service request* to the Quality Factory. This request will be honored by the Quality Factory and a service provider will be requested (on behalf of the application) to provide a certain service. Which service provider is selected and which version of the *provided service* is needed from the service provider depends on the negotiation over the different resources which are available. This will be discussed in more detail in §5.4.

Within each service request information is present which specifies which service should be provided in what way. In general this information consists of three parts:

- the *name* of the service.
  - the name identifies which service is intended.
- the *arguments* of the service.
  - the arguments can be used to pass information to the service provider; it may contain information that is used for the service provider as data to work with or it can be information which controls the exact result of the service.
- the *result* of the service.
  - a service, as provided by a service provider, may return a result which may indicate in which way a service request was honored. This result can be either a success code, failure code or some data.

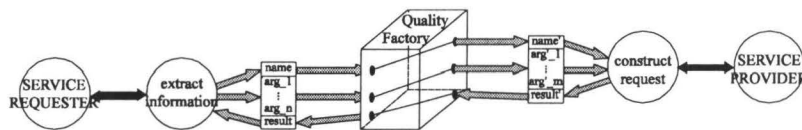
This subdivision of the service request into three parts is important. Different service providers can provide the same service which can differ on each of these points. This can cause problems when the Quality Factory wants to act as a shell around these service providers; the same information, contained in a single service request, must be mapped in different ways to different service providers. These differences may appear in one of the following aspects (or in a combination of these aspects):

- the number of arguments used by the service requester and the service provider may be different:
  - one argument of the service requester may be split into several arguments for the service provider.
  - several arguments of the service requester may be combined into one argument for the service provider.
  - an argument of the service request is not used by the service provider.
  - an argument which is demanded by the service provider is not provided by the service requester.
- the order in which arguments are specified by the service requester is different than the order used by the service provider.
- the types of some arguments used by the service requester are different from the types demanded by the service provider.
- the result of the service request may be reported back by the service provider differently than expected by the service requester:

- the result may be reported back as a return code by the service provider whereas the service requester expected the result to be stored in an argument.
- the result may be reported back in an argument by the service provider whereas the service requester expected the result as the return code.
- the name used for a particular service may be different from the one use by the service provider:
  - the service provider and the service requester may use the same name for two different services.
  - the service provider and the service requester may use two different names for the same service.
  - the service provider may expect a service which can only be supported by a combination of services provided by the service provider.
  - the service provider expects a service which can only be provided by the service provider as part of other provided services.

Because the way in which the various services may differ in name, number of arguments, the types of the arguments and the way in which a result is returned, the Quality Factory will take a service request, extract the different parts of information from this request and use the individual parts to redirect the request to a service provider and so honor the request. When a service provider is requested by the Quality Factory to deliver a certain service, the reverse process is applied; from the individual pieces of information a service request for the service provider is constructed.

To be able to do so, the Quality Factory will intercept the service requests from the requester before they have reached a service provider. Using the parts of information from the request, a translation will take place as part of the tasks performed by the Quality Factory. Once the translation of the request is finished, this modified request will be send to a service provider. In its turn, the service provider may return a result which is propagated to the original requester via the Quality Factory:



The approach, to use the different parts of information of a service request, is also useful for another reason; service requests may compete with each other for the same resources. After service requests, expressed in terms of the viewer's perceived quality, are translated to requests in terms of claims of resources, different service requests may (in the end) claim the same resource. As a result of that, some requests cannot be honored and some of them can only be partially honored. For these kinds of situations, negotiation may be in order to find a (semi-)optimal solution by which as much requests as possible are honored as best as possible. On this level, the information in the arguments of the different requests is essential; they may provide clues how to guide the negotiation process.

In the demonstration program, three services are provided. These services are described in Annex A.

## 5.2 Filters

For the translation from the viewer's perceived quality to claims of resources, the Quality Factory makes use of so-called *filters*. These filters usually take (part of) the user's requirement, modify some or all of the names and the arguments of the request and formulate a new request for either another filter or a service provider. The filters themselves can be seen as service providers too; they provide services to translate service requests. In the remainder of this text, the distinction between filters and service providers will be maintained to indicate clearly which part of the Quality Factory is referred to.

Important aspects for filters are the four characteristics of the viewers perception which were already introduced in Chapter 2:

- the viewer has a continuous perception.
- the viewer is unaware of underlying structures and formats.
- the viewer may not be able to perceive all of the information that is presented.
- the viewer may be unaware of the used resources.

These four aspects have a great impact on the nature of the filters. Filters will have to provide extra information to the Quality Factory to fill the gap between the user's perceived quality metrics and the metrics used on the lower, more structure and format oriented, level. The Quality Factory supports four kinds of filters:

- filters which are aware of the format of the data which is provided to them and can perform specific transformation on the data in that specific format.

Because these kind of filters know which format to expect, the data can be interpreted and specialized transformations on the data can be applied. Examples of these kind of filters are:

- a filter which can retrieve the I-frames from a MPEG-video.
- a filter which can change the brightness in a RGB-formatted image.
- a filter which can change the brightness in a YC<sub>b</sub>C<sub>b</sub>-formatted image<sup>2</sup>.
- a filter to compress a sound-file (containing speech) using the GSM 06.10 RPE-LTP specification.
- filters which are aware of the structure of the underlying network and the particularities of the communications in that network.

These filters are aware of the capacities of the connections between the different nodes in the network. They have information about baud-rate, buffers and reliability of the different nodes. Depending on this information, these filters may provide compression/decompression or extra buffering. Examples of these kinds of filters are:

---

<sup>2</sup> Note that the filter to change the brightness for a RGB-formatted image is different filter than the one which changes the brightness of an YC<sub>b</sub>C<sub>b</sub>-formatted image; the actions to change the brightness are very specific to the format in which the image is stored. In the latter case, simply the Y component has to be altered, whereas in the former case, the brightness has to be computed using the R, G and B components (e.g.  $Y = 0.2125 * R + 0.7145 * G + 0.0721 * B$ ).

- a filter to compress data using the *zip*-specifications<sup>3</sup>.
  - a filter to buffer data, either to memory or disk.
  - a filter to duplicate information to two different nodes in the network.
- filters which are able to transform data, structured in one format, into data, structured in another format, which still contains, in essence, the same information.

These filters may have to be inserted in the translation process when a certain operation is required but no format specific filters are available which are capable to do so for the data in the current format. These kinds of situations may occur frequently. In such a case, the operation may still be performed when a filter exists which can perform the desired operation for another format and one which can transform the data to that other format. An example of this is a situation where data is stored in one format and the, to the user, available tools cannot display that specific format (but only some other formats). Examples of these kinds of filters are:

- a filter which transforms a GIF-encoded image to the same image in PCX format.
  - a filter which transforms text to a sound file.
  - a filter which performs OCR (optical character recognition) on an image of a specific type.
  - a filter which translates mouse-actions to keystrokes.
- filters to negotiate over a particular resource

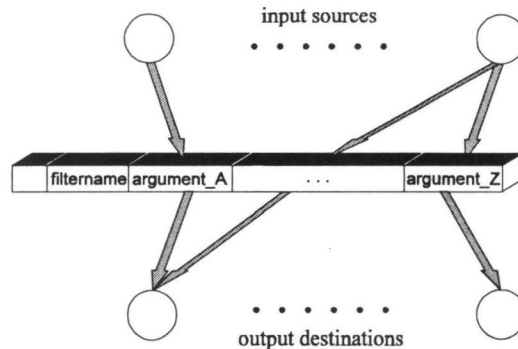
Service requests which may, in the eyes of the viewer, have nothing to do with each other, may be in conflict with each other when they are translated to a lower level and need the same resource. In that case negotiation filters may, given a certain claim on a certain resource, decide that the resource is or is not (yet) available for the new request. If it is not available or if the available capacity is not large enough to perform the desired task, the filters may decide that other users of that resource have to limit their usage in order to (partially) honor the current request. Examples of negotiation filters are:

- reservation protocols like RSVP, ST-II, etc.

A filter may receive its input from several sources and it may send (part of) its output to several destinations. It is possible that different sources are input to the same argument as well as that the same source is used to provide the input for several arguments. The output of a filter (in terms of the value of the arguments of the filter) can also be send to several destinations; one value may be send to several destinations at the same time as well as several values can be used to determine the output for one destination. These kinds of schemes allow to support situations where different service requesters, service providers and/or filters use different structures to pass information in their arguments (see also §5.1). The schematic representation of a filter is shown below:

---

<sup>3</sup> Note that 'zip' is a general compression technique. Although the GSM 06.10 RPE-LTP specification is also a compression technique, it is not considered to be a general compression technique. The reason is that GSM relies heavily on the fact that the data is sound and, more precise, that the sound is human speech. Applying GSM on arbitrary data may leave out information, which is essential for a lossless compression of the data.



When the application has made a service request, it may be necessary to activate several filters. Which filters have to be activated depends on:

- the service request that is made
- the service providers that can service the request
- the format of the data
- the resources used by the different filters themselves
- if appropriate, the structure of the network over which the data has to be sent.

Furthermore, the order in which the different filters are applied can have its effect on the efficiency in which the request is serviced, the quality provided and the usage of the different resources. In Annex A several filters are described. A clear understanding of the different filters may be necessary to understand the remaining part of this chapter and the logic why certain filters are (not) connected to other filters (it would not make any sense to connect a compression filter directly to a decompression filter).

### 5.3 Formats

The notion of format plays an important role in the Quality Factory. However, the Quality Factory itself has no control over the different formats; they are considered as a given entity. Formats are considered to be a *representation of information*. This means that different formats may contain the same information organized in a different way. However, it is possible that, when information is stored in a particular format, part of this information is lost. Examples of this phenomenon are numerous:

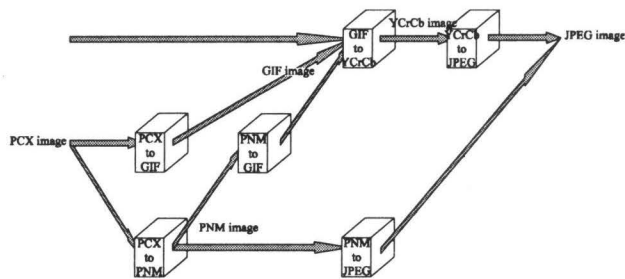
- digital samples of continuous data, meaning:
  - all digital sound files stored in wav, voc, snd or another format.
  - all digital images which contain a representation of a part of the real world.
- JPEG images which can be compressed more at the cost of some of the information.
- bitmaps versions of images which were specified using vectors.
- ...

For the Quality Factory, all information is stored in a certain kind of format. This implies that the types of the arguments are formats too. The role that the Quality Factory can play with respect to formats, is when one format has to be translated into another format. This translation is necessary when certain filters can only perform their operations on data in a specific format or when the

service request uses a different format for its arguments than the format expected by the filters or a service provider.

To make this conversion of formats possible, the Quality Factory has two different kinds of filters at its disposal; the format-aware filters, which can perform an operation on information in a particular format, and the filters that can transform information from one format into another one. By creating links between the filters of the first group and the filters from the second group, it is possible to create a functionality which is larger than just the sum of the functionality of all the individual filters.

By connecting all the conversion filters which transform a format "X" to another format "Y" to the filters which transform the format "Y" to another format "Z", the Quality Factory is able to convert the format "X" to the format "Z":



In the example above, the following conversions can be made:

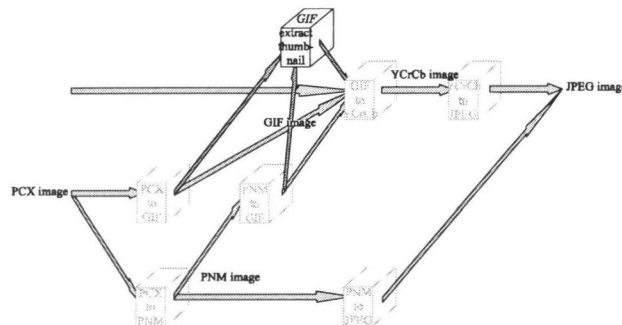
GIF	→	JPEG, YC,C <sub>b</sub>
JPEG	→	-
PCX	→	GIF, JPEG, PNM, YC,C <sub>b</sub>
PNM	→	GIF, JPEG, YC,C <sub>b</sub>
YC,C <sub>b</sub>	→	JPEG

If, in addition, all the format-aware filters which expect their information to be in format "X" are connected to the input arguments of the transformation filters "X->?" and to the output arguments of the transformation filters "?->X", it is possible to perform the operation of the format-aware filter on all the data which can be transformed to format "X" and the result of the operation can be converted to all the formats to which format "X" can be converted<sup>4</sup>:

---

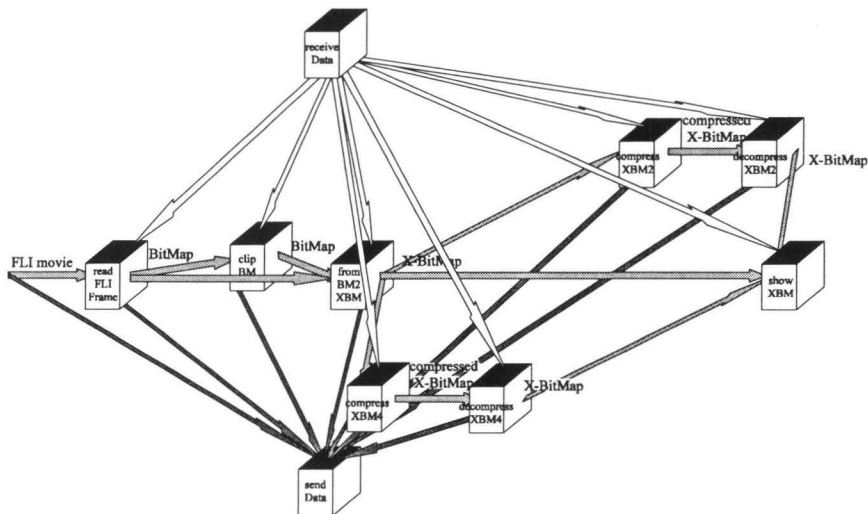
<sup>4</sup>

As a convention, which is used in the remainder of this chapter, all the light gray parts of a diagram are parts which have already been introduced in an other diagram; the dark parts are the parts which are new.



In the above structure, a format-aware filter is able to extract a thumbnail (i.e. a smaller, less detailed copy of the image, used for previewing purposes) from a GIF-encoded image. Because this filter is connected to the input argument of the transformation filter "GIF->YCrCb", the thumbnail of the GIF-encoded image can also be transformed to a YCrCb-encoded image or a JPEG-encoded image. Furthermore, it may be possible to extract a thumbnail from either a PCX-encoded or an PNM-encoded image; images stored in one of these two formats can be transformed to a GIF-encoded image. In the GIF format it is possible to store a pre-computed thumbnail in the image file. Depending on the way this transformation is done, a thumbnail may have been inserted in the new GIF-encoded image. If that is not the case, it depends on the filter which extract the thumbnail from the GIF-encoded image whether or not a thumbnail is created for the GIF-image after all (will the filter only look for a thumbnail block in the GIF-image and reproduce the information in that block, or will it create a thumbnail itself if such a block is not available).

If in the demonstration program, the different filters are connected in the way described above, the situation as shown below will be created. In this diagram, three different colors of arrows are used. This is done to make it easier to distinguish the different arrows in the diagram. The white arrows are all starting in *receiveData* and the dark gray arrows all end in *sendData*. The light gray arrows are used to show the connections between the remaining filters:





Thus the Quality Factory may be used to breakdown the barriers put up by the large amount of formats; by combining the right filters in the right order the Quality Factory can provide an environment which performs its tasks independent of the input-format. More generally speaking, a system like the Quality Factory can provide a system which is context independent in the sense that it can apply filters to its input to convert this input to the appropriate format.

### 5.4 Resources and Negotiation

As pointed out in Chapter 2, a viewer can have a totally different notion of what resources are available and how they are used than the notion used by an arbitrary application. In this section, it will be explained what the notion of resource means in relation to the Quality Factory.

For the Quality Factory, a resource is something which has a limited capacity and for which (part of) this capacity can be claimed by different filters. In the Quality Factory, the claims for these resources are typically managed by the negotiation filters. Because a resource has a limited capacity and different filters can claim part of this capacity at the same time, the filters may have to compete for the resource. The negotiation filter receives the different claims and, based on the available capacity and the priority with which the different claims are made, it will assign (some of its) capacity to the different claims. Note that it is possible that, due to a new claim, the negotiation filter reassigns a smaller portion of the resource's capacity to a filter which was formerly assigned a larger portion of the resource's capacity.

The claims for portions of the resource's capacity can be made with different priorities. The priorities scheme used by the Quality Factory differs from the scheme used in existing Quality of Service systems. The reason for this is that the Quality Factory may negotiate with the different filters, which claim a resource, on the portion of the capacity claimed. In order to negotiate, it is necessary for the Quality Factory to know how firm the request is and what the limits are which are still acceptable for a certain filter with respect to the portion of the resource's capacity which will be assigned to it. The following categorization with respect to the firmness of a claim within the Quality Factory can be made:

- **non-negotiable**

The claim cannot be negotiated on. The resource manager should either comply with the claim to the full extent or refuse the whole claim.

- **negotiable**

The claim can be negotiated on. The claim should be considered as a strong guide for the negotiation process. The final portion of resource capacity on which an agreement is reached should be close to the original claim.

- **best effort**

The claim is not a hard claim. The filter which makes an obliged claim will be satisfied with any portion of the resource's capacity it receives from the resource manager.

The three categories described above can be used by filters to claim (part of) the resource's capacity. Filters can claim this capacity in six different ways:

- **bound (lowerbound: *l\_firmness*, claim: *c\_firmness*)**

Claim a part of the capacity of the resource. The portion claimed is *claim* and the firmness with which this claim is made is *c\_firmness*. If the resource manager does not comply with this claim, a smaller portion of the capacity of the resource may be assigned. However, the smallest size that is accepted by the filter, which has made the claim, is

*lowerbound*. Again, this lowerbound has a certain firmness (*l\_firmness*). Note that it only makes sense that *l\_firmness* is firmer than *c\_firmness*.

- **bound&lock (lowerbound: *l\_firmness*, claim: *c\_firmness*)**

This claim is similar to the normal **bound** claim. However, once a portion of the resource's capacity is assigned to the claiming filter, this portion may not be changed anymore as a result of a negotiation process with other filters which claim part of the resource's capacity at a later moment in time. This claim assures that a certain filter can rely on its claimed and assigned resources and that the task, performed by that filter, produces a stable output.

- **circa (claim: firmness)**

Claim a part of the capacity of the resource. The portion claimed is *claim* and the firmness with which this claim is made is *c\_firmness*. However, no demands are made on the lowest acceptable portion of the resource's capacity. This means that the assigned capacity may vary from nothing to *claim*.

- **circa&lock (claim: firmness)**

This claim is similar to the normal **circa** claim. However, once a portion of the resource's capacity is assigned to the claiming filter, this portion may not be changed anymore as a result of a negotiation process with other filters which claim part of the resource's capacity at a later moment in time. This claim assures that a certain filter can rely on its claimed and assigned resources and that the task, performed by that filter, produces a stable output.

- **all: firmness**

This claim claims all of the available resource's capacity. This claim is made with firmness *firmness*. The aim of this claim is to get as much resources as possible, however the resource manager may decide to assign only a portion of the available capacity.

- **all&lock: firmness**

This claim is similar to the normal **all** claim. However, once (a portion of) the resource's capacity is assigned to the claiming filter, this portion may not be changed as a result of a negotiation process with other filters which claim part of the resource's capacity at a later moment in time. This claim assures that a certain filter can rely on its claimed and assigned resources and that the task, performed by that filter, produces a stable output.

Note that the scheme to allocate resources is much more flexible than the ones used in the QoS systems described in Chapter 2. Furthermore, the specification of the bounds allows for a more serious negotiation process where it is really possible to manage resources.

In addition to the above described claims to choose from, trade-offs can be made between different resources to provide to better allocation-scheme; when the assigned capacity of a certain resource is decreased, the need for another resource may become more or less urgent. This trade-off is another important aspect in the negotiation process; by using trade-offs, a negotiation filter may find a situation where the claims of the different resources are rearranged in such a way that the resources are used more efficiently due to which a service request can, overall, be better honored. Trade-offs will be discussed in more detail in §5.5.1.1.

In the demonstration program, the two negotiation filters both manage the bandwidth resource of the network connection between the two computers (the one where the movie is stored and the one where it is displayed). It can easily be seen that changing the turning speed of the globe affects the amount of data that has to be sent over the data-connection; when the turning speed is increased by a factor 2, the amount of data that has to be sent over the data-connection every second has to be doubled too. Likewise, when the turning speed is decreased by a factor 4, the amount of data that has to be sent over the data-connection every second is only 25% of the original amount. A similar argument can be used for the level of detail; when the level of detail may be reduced to 50%, data may be compressed using *compressXBM2* and thus data only half the size of the original data

can be sent over. When the level of detail may be reduced to 25%, *compressXBM4* may be used and data only quarter of the size of the original data can be sent.

Because both negotiation filters can affect the amount of data that has to be communicated, one may conclude that a trade-off relationship exists between the turning speed of the globe and the level of detail at which the globe is displayed.

In the Quality Factory, negotiation filters have to be based upon the *NegotiationObject*. This *NegotiationObject* is a constraint object which provides the foundation of negotiation in the Quality Factory. Details about this *NegotiationObject* are presented in §5.5.2 when the Quality Factory is explained in more detail.

### 5.5 Software Reuse

The task of the Quality Factory can be compared with the task of software reuse. In [Krueger 92] software reuse is described as the process of using existing software artifacts rather than building them from scratch. Typically, reuse involves abstraction, selection, specialization and integration of artifacts. The primary motivation for software reuse is to reduce the time and effort needed to build new applications.

The Quality Factory reuses the services provided by the service providers, and by doing so, the different resources available in the system. The introduction of filters and negotiation objects make it possible to combine different software components and improve their interoperability. By abstracting, selecting, specializing and integrating the services of different service providers, new services can be introduced in the Quality Factory and these new services can be used in combination with other services to create other, more complex services. In case services are combined, the output of one service may be used as input for another service and in this way the combination of the two services can be regarded as only one service which has combined the functionality of the two services together. By using the Quality Factory, time and effort, needed to build systems that honor particular service requests, are reduced.

In the demonstration program, different filters are combined in such a way that it is possible to view an FLI movie in an environment where before only XBM images could be displayed; the service *showNextFLIFrame* is a combination of reading a frame from a FLI movie (*readFLIFrame*), converting it to an image in XBM format (*clipBM*, *fromBM2XBM*) and finally displaying it on the screen (*showXBM*). The service *showNextFLIFrame* in its turn, could be used in combination with *initializeFLI* and *terminateFLI* to create a new service (for instance *showFLI*), which would combine the functions of the three services mentioned above; open the file which contains the FLI movie (*initializeFLI*), show all the frames of the FLI movie on the screen (repeatedly using *showNextFLIFrame*) and close the file again (*terminateFLI*).

In the literature, several systems are described which define an environment in which several (existing) software artifacts can be integrated. In [Verall 91] and [Berre 92] the Eureka Software Factory is described. This is a software bus to which several systems can be connected (as different hardware can be plugged into a hardware bus). A similar approach is taken in [Bergstra et al. 94]. They designed a 'ToolBus'. Other approaches are taken where the communication between different objects in an object-oriented environment is regulated ([Helm et al. 90], [Hölzle 93], [Mili 92], [Morley et al. 91]) or where design decisions are formalized and stored in special libraries ([Gamma et al. 93], [Pree 94]). However, in all cases, the key to successfully integrating different software artifacts lies in the quality and quantity of information which is provided by the different artifacts in order to make this integration possible. All these systems differ with the Quality Factory in that the Quality

Factory can dynamically add new software artifacts to reuse and that it can negotiate the resources that will be used.

In the remainder of this section, the different phases of software reuse (abstraction, selection, specialization and integration) are discussed as well as how each of these phases relates to the Quality Factory. However, before these phases are described, it is important to understand in which aspects software reuse and the tasks of the Quality Factory match and differ. There are a number of similarities between software reuse and the tasks of the Quality Factory:

- the set of provided services/artifacts is a given entity and no influences can be exercised on this set.
- the format of the data required by a service request/artifact, may be different from the format that is available; therefore format conversion may have to be applied before the artifact can be used.

However, the Quality Factory has some features that are not found in the area of software reuse:

- the Quality Factory is able to dynamically map service requests to provided services.
- the Quality Factory is able to deal with competitive and contradicting service requests.

### 5.5.1 Abstraction

Abstraction in software reuse is used to describe "what" the different artifacts do instead of "how" they do it. Abstraction is the most important part of software reuse. This part describes the functionality of the artifact in terms that are understandable by and which have meaning to the person or the program that has to realize the reuse. As a result of that, abstraction forms the basis for the other three parts. In the next sections it is shown what kind of information is needed to describe the "what" of the different filters in the Quality Factory and what kind of detail is needed to enable to Quality Factory to connect the different filters together. The abstraction in the case of the Quality factory is split into three parts: functional descriptions (§5.5.1.1), argument relations (§5.5.1.2) and (dynamic) contexts (§5.5.1.3).

#### 5.5.1.1 Functional Descriptions

For the Quality Factory, the different services provided, the available resources and the filters in the system should be described in such a way that the Quality Factory can connect the different parts so service requests can be translated into the activation and selection of the proper filters and provided services. This means that it is necessary to describe what each filter does, which formats are used by which filters and how much of which resources are needed by a certain filter or service provider. Below, for each type of filter, the necessary information with regard to their functional description is shown. These descriptions show the kind of information that is typically needed to make an inventory of what services and functionality is available in the Quality Factory and how these services use the different kinds of resources. The information described in this section will also be important for the other aspects of software reuse: selection, specialization and integration of the different filters.

- format aware filters:

Each of these filters only works on one specific format and performs a special operation. These filters can therefore be identified by the format and the operation they perform. Furthermore, they typically transform the data in such a way that either system resources, viewer's perceived resources or a combination of both are affected.

FORMAT	OPERATION	VIEWER'S QUALITY	SYSTEM QUALITY
GIF PCX	change brightness change number of colors	brightness contrast	bits/pixel (MEM) time (CPU)
	change size	sharpness contrast	size (MEM) time (CPU)
MPEG	retrieve I-frames	smoothness	size (MEM) time (CPU)
	compress video	smoothness sharpness	size (MEM) time (CPU)
GSM	change speech to text		size (MEM) time (CPU)

° transform filters:

Each of these filters transforms data from one specific format to another. Therefore, these filters can be identified by the source format and the destination format. The operation they perform is always a transformation. During this transformation both, the viewer's perceived resources and the system resources, can be affected.

FORMAT	VIEWER'S QUALITY	SYSTEM QUALITY
GIF->YC,C <sub>b</sub> GIF->PCX	brightness contrast number of colors	size (MEM) time (CPU)
PCX->ASCII	detail	time (CPU)

° network aware filters:

These filters perform operations like buffering and compression. They are typically identified by their operation.

OPERATION	VIEWER'S QUALITY	SYSTEM QUALITY
ZIP compress ARJ compress ZOO compress LZH compress	brightness contrast number of colors	size (BANDWIDTH) time (CPU) time (LATENCY)
buffer	response time smoothness	time (CPU) time (DELAY) time (JITTER)
merge stream duplicate stream		

° negotiation filters:

These filters perform the negotiation between service requests which claim a number of resources. They are typically identified by these resources itself. Negotiation filters can negotiate for one, two or more resources.

QUALITY 1	QUALITY 2	QUALITY 3
size (BANDWIDTH)	number of colors	
size (MEM)	size (BANDWIDTH)	
size (THROUGHPUT)	number of colors	frame rate
time (CPU)	size (MEM)	
time (LATENCY)		
time (DELAY)		
time (JITTER)	frame rate	
...		

As can be seen above, filters may affect several resources at the same time. The behavior of a filter can be influenced by assigning it different capacities of certain resources. The introduction of trade-offs between resources, such that less claims are made for resources which are more scarce, a better overall performance can be realized. Trade-offs are describes as a pair of resources and a description of the relative importance of these two resources. This description of relative importance is based on the notation which is used to describe the complexity of algorithms. In this notation, the order-symbol  $O$  is used to describe the relative growth of two functions:  $f(n) = O(x(n))$  means that  $f(n) \leq C * x(n)$  for some constant  $C$ .

In the description of these trade-offs, the constant  $C$  and the order of function  $x$  is used. The relative importance  $(m, n)$  expresses the relative growth of a *resourceB* with respect to *resourceA* as follows: if the claim on resource *resourceA* grows with a factor  $f$ , the need for capacity of resource *resourceB* will grow with  $m * f^n$ . The values for  $m$  and  $n$  may be negative. If one of these values is 0, no relation is assumed. Although the expressive power of this approach to specify the trade-offs is limited, it is sufficient for its purpose here; trade-offs will, in most cases, only be an estimate of the existing relations rather than a precise description:

TRADE-OFF (resourceA, resourceB)	meaning
(5, 1)	$\text{growth}(\text{resourceB}) \approx 5 * \text{growth}(\text{resourceA})$
(3, 0)	no relation
(1, 2)	$\text{growth}(\text{resourceB}) \approx \text{growth}(\text{resourceA})^2$
(6, -2)	$\text{growth}(\text{resourceB}) \approx 6 * \text{growth}(\text{resourceA})^{-2}$
(0, 0)	no relation

For each negotiation filter a trade-off can be specified for all the different pairs of resources it needs to manage.

For the filters defined in the example, the following information should be provided (in the table below the information for the four different filter types are mixed together):

	FORMAT	VIEWER'S QUALITY	SYSTEM QUALITY	TRADE-OFF
NegotiateTurningSpeed		turning speed	size (BANDWIDTH) time (JITTER)	(turning speed, BAND.) = (1, 1) (turning speed, JITTER) = (?, ?)
NegotiateDetailLevel		level of detail	size (MEM) size (BANDWIDTH)	(level of detail, MEM) = (1, 1) (level of detail, BAND.) = (1, 1)
readFLIFrame	FLI → BM	-	-	
fromBM2XBM	BM → XBM	-	size (MEM) size (BANDWIDTH)	
compressXBM2	XBM → XBM2	level of detail	size (MEM) size (BANDWIDTH)	
decompressXBM2	XBM2 → XBM	-	size (MEM) size (BANDWIDTH)	
compressXBM4	XBM → XBM4	level of detail	size (MEM) size (BANDWIDTH)	
decompressXBM4	XBM4 → XBM	-	size (MEM) size (BANDWIDTH)	
sendData (send data to a socket)	*	-	-	
receiveData (read data from a socket)	*	-	-	
initFLI (open file)	FLI	-	-	
closeFLI (close file)	FLI	-	-	
clipBM (extract rectangle)	BM	viewable portion	size (MEM) size (BANDWIDTH)	

From the information provided above, the following information, with respect to which filters can be connected to each other, can be derived (this information is exactly the same as was presented in the last diagram in §5.3):

- clipBM, in relation to fromBM2XBM, can be used as input source and, in relation to readFLIFrame, as output destination
- readFLIFrame, in relation to fromBM2XBM, can be used as input source
- initFLI, in relation to readFLIFrame and closeFLI, can be used as input source
- decompressXBM2, in relation to compressXBM2, can be used as output destination
- decompressXBM4, in relation to compressXBM4, can be used as output destination
- fromBM2XBM, in relation to compressXBM2 and compressXBM4, can be used as input source
- sendData, in relation to all conversion and format-aware filters which have at least one output destination, can be used as output destination
- receiveData, in relation to all conversion and format-aware filters which have at least one input source, can be used as input source

Note that, for instance, `closeFLI`, in relation to `readFLIFrame`, cannot be used as input source because `closeFLI` does not have the possibility to be used as input source in relation to any filter (it has no output parameters).

The following information can also be derived from the functional descriptions of the filters:

- the perceptual characteristic turning speed (`negotiateTurningSpeed`) can influence the system characteristic size(`BANDWIDTH`). Therefore, the turning speed can influence the result of `fromBM2XBM`, `compressXBM2`, `decompressXBM2`, `compressXBM4`, `decompressXBM4`, `clipBM` and `negotiateDetailLevel`.
- the perceptual characteristic level of detail (`negotiateDetailLevel`) can influence the system characteristics size(`MEM`) and size(`BANDWIDTH`). Therefore, the turning speed can influence the result of `fromBM2XBM`, `compressXBM2`, `decompressXBM2`, `compressXBM4`, `decompressXBM4`, `clipBM` and `negotiateTurningSpeed`.
- the perceptual characteristics turning speed and level of detail can be negotiated over by the Quality Factory. The relation between these two characteristics is linear:

$$\begin{aligned}
 &\text{growth (level of detail)} \\
 &= 1 * \text{growth (level of detail)} ^1 \\
 &= \text{growth (BANDWIDTH)} \quad ( \text{TRADE-OFF (level of detail, size(BANDWIDTH))} = (1, 1) ) \\
 &= 1 * \text{growth (turning speed)} ^1 \quad ( \text{TRADE-OFF (turning speed, size(BANDWIDTH))} = (1, 1) ) \\
 &\Rightarrow \text{TRADEOFF (turning speed, level of detail)} = (1, 1)
 \end{aligned}$$

This information will be used when the user states requirements in his own terms (turning speed and level of detail) and the Quality Factory has to translate this into control parameters with respect to throughput, bandwidth, delay etc. for the network communication software.

#### 5.5.1.2 Argument Relations

Beside the global description of the functionality of the various filters (described in the previous section), it is also necessary to describe in more detail how the different filters can be connected; the Quality Factory has to have some information how the data flows between filters such that results from one filter are correctly passed on to another filter. For this purpose, the Quality factory needs to know how the values of the individual arguments are computed using the arguments of other filters. The Quality Factory supports arguments of the service requester, the service provider and the filters which have the following types:

- `void` and `void*`, `void**`, etc.
- `char` and `char*`, `char**`, etc. (`char` is a C++ storage class of 1 byte; values: 0 .. 255)
- `short` and `short*`, `short**`, etc. (`short` is a C++ storage class; it can contain as least as much information as a `char`)
- `int` and `int*`, `int**`, etc. (`int` is a C++ storage class; it can contain as least as much information as a `short`)
- `long` and `long*`, `long**`, etc. (`long` is a C++ storage class; it can contain as least as much information as a `int`)
- `float` and `float*`, `float**`, etc. (`float` is a C++ storage class for floating point numbers)
- `double` and `double*`, `double**`, etc. (`double` is a C++ storage class; it can contain as least as much information as `float`)

Beside these standard types, the Quality Factory supports so-called *references*. A reference is specified by a symbolic name and a number of fields. Each of these fields has a name and a type. The type of each field has to be either a standard type, a pointer to a reference or a reference other than the one being specified. Furthermore, cyclic dependencies between references may be created. When a reference is referred to, it is placed between '<>'. Fields may be assigned a default value;



this is specified by the '='-sign and an immediate value. These references can be used to represent user-defined types.

A C++ structure declaration can be translated into a reference as follows (note that the default value for *integerField* cannot be represented in the C++ structure and that *anotherReference* is assumed to be a valid type which is already defined):

```
struct SymbName {  
    int          integerField;  
    anotherReference referenceField;  
    SymbName*    Next;  
};
```

```
REFERENCE SymbName (int integerField = 25; <anotherReference> referenceField; <SymbName*> Next);
```

Using this type scheme, the Quality Factory can support standard C++ types, structures and user-defined types. This type scheme is needed to describe all the possible service requests, which can be issued, and the provided services, which can be honored. A service request is identified by the keyword *SERVICEREQ*:

```
SERVICEREQ int ServiceRequestName {double doubleArgument, <SymbName*> referenceStarArgument} = 25
```

In this specification, it can be seen that the result argument of a service request (i.e. the argument which contains the return value of the service) can be assigned a default value. This default value is specified directly after the list of arguments by a '='-sign and a representation of the immediate default value. Arguments cannot be assigned default values as the values for the arguments are expected to be specified in the service request. The provided services are described as follows:

```
SERVICEPRV int ServiceProvidedName {short shortArgument = 25, <SymbName> referenceArgument}
```

In this specification, it can be seen that a non-result argument of a provided service can be assigned a default value (normally the value of an argument will be computed using the results of other filters., but when such a specification is omitted, a default value can be used). This default value is specified directly after the argument name by a '='-sign and a representation of the immediate default value. The result parameters cannot be assigned a default value as it is expected that it will be assigned a value by the service provider.

Within the Quality Factory, arguments of services which have one of the types describes above, may be connected to each other by means of an expression. That means that the Quality Factory not only simply copies the value of one argument to another argument, but that it is possible to compute the value of one argument using the values of several other arguments and some predefined operations. How this value is computed can be described in an *argument relation*. At the moment, the Quality Factory supports six operations in argument relations:

- the assignment operator '*destinationArgument*<-*expression*', which assigns the value of *expression* to *destinationArgument*.  
the assignment operator may only occur as the first operator in an argument relation. *destinationArgument* can be the result argument of a service request, a non-result argument of a filter or a non-result argument of a provided service. In *expression* an expression may be specified in terms of immediate values, result arguments of filters or provided services and the non-result arguments of either filter, provided service or service request.

- the synchronization operator '*destinationArgument*<#*expression*', which orders the activation of the different filters.  
the synchronization operator may only occur as the first operator in an argument relation. The filters, whose argument values are dependent on *destinationArgument*, are activated later than the filters which (indirectly) contribute to the evaluation of the values of the arguments used in *expression*. Or, in other words, the value of *destinationArgument* may only be used after all argument relations which contribute to the values of the arguments in *expression* are executed. *expression* may be an arbitrary expression; the expression will not be evaluated. The same restrictions hold with respect to the types of the arguments allowed in *destinationArgument* and *expression* as in the previous case.
- the add operator '*Argument1*+*Argument2*' will evaluate to the sum of the values of the two individual arguments.  
The add operator may not occur as the first operator in an argument relation. The values of the arguments *Argument1* and *Argument2* are added. *Argument1* and *Argument2* may not be the result argument of a service request.
- the subtract operator '*Argument1*-*Argument2*' will evaluate to the subtraction of the two individual arguments.  
The subtract operator may not occur as the first operator in an argument relation. The value of argument *Argument2* is subtracted from the value of argument *Argument1*. *Argument1* and *Argument2* may not be the result argument of a service request.
- the multiply operator '*Argument1*\**Argument2*' will evaluate to the product of the two individual arguments.  
The multiply operator may not occur as the first operator in an argument relation. The values of the arguments *Argument1* and *Argument2* are multiplied. *Argument1* and *Argument2* may not be the result argument of a service request.
- the divide operator '*Argument1*%*Argument2*' will evaluate to the quotient of the two individual arguments.  
The divide operator may not occur as the first operator in an argument relation. The value of argument *Argument1* is divided by the value of argument *Argument2*. *Argument1* and *Argument2* may not be the result argument of a service request.

The operators are left-associative and the priority of the operators is (in ascending order): <#, <-, +, -, \*, %. The Quality Factory does not support braces. This does not restrict the expressive power of the expressions but it may lead to expressions which are longer than the equivalent expression using braces. As the operators + and \* are associative, the result is independent of the fact whether they are computed left- or right-associative. For the operators - and %, a right associative sequence can be transformed to a left-associative sequence by using the rules  $a - (b - c) \equiv a + (c - b)$  respectively  $a \% (b \% c) \equiv a * (c \% b)$ .

The operations presented above provide only some basic functionality which can be used to connect arguments together. Other operations can be defined so more complex manipulations on the data can be performed by the Quality factory itself. However, a careful analysis should be made what kind of functionality should be build in and what kind of functionality should be realized by introducing new filters.

The arguments within the various expressions have a unique name. Their name is constructed from the name of the service/filter they belong to, a separator '::', the name of the argument within the service and, in case of a reference, a separator '\_' and the name of the field. The result of a

service/filter can be identified by the absence of the argument name (in that case, the separators are still present):

ServiceProvidedName::shortArgument<- ServiceRequestName::doubleArgument+1	assign 1 plus the value of argument <i>doubleArgument</i> of service request <i>ServiceRequestName</i> to the argument <i>shortArgument</i> of provided service <i>ServiceProvidedName</i> .
ServiceRequestName::<#ServiceProvidedName::	relations in which <i>ServiceRequestName::</i> is used in the right-hand side of the expression are only evaluated after all expressions which directly or indirectly contribute to the evaluation of <i>ServiceProvidedName::</i> are computed. An expression <i>y</i> contributes indirectly to the evaluation of <i>ServiceProvidedName::</i> in an expression <i>x</i> if the left-hand side of expression <i>y</i> is used in the right-hand side of <i>x</i> or if the left-hand side of expression <i>y</i> is used in another expression which indirectly contributes to <i>x</i> .
ServiceProvidedName::referenceArgument_integerField <-5	assigns the value 5 to field <i>integerField</i> of argument <i>referenceArgument</i> of filter <i>Service-ProvidedName</i> .

When necessary, types of values in the expressions are coerced. In the example above, *doubleArgument*, with type *double*, was used in the addition with the integer value 1. The result of that addition was assigned to *shortArgument* with type *short*. In these cases, several coercions have occurred. First, the immediate value 1 was coerced to the type *double*. This was added to *doubleArgument* and the result was coerced to the type *short*. This coerced value was finally assigned to *shortArgument*. In general, the Quality Factory uses the following rules for coercion:

- if one of the operations +, -, \* or % is encountered and both operands have the same type, no coercion takes place.
- if one of the operations +, -, \* or % is encountered and both operands have one of the types: *char*, *short*, *int* or *long*, both operands are coerced to the type *long*.
- if one of the operations +, -, \* or % is encountered and both operands have one of the types: *float* or *double*, both operands are coerced to the type *double*.
- if one of the operations +, -, \* or % is encountered and one operand has one of the types: *char*, *short*, *int* or *long* and the other operand has one of the types *float* or *double*, both operands are coerced to the type *double*.
- if the operation <# is encountered no coercions are necessary.
- if the operation <- is encountered a coercion is performed in which the right operand is coerced to the type of the left operand. In this case, information can be lost because (unlike the other cases when always coercions to a larger type were performed) values can be coerced to a smaller type which may, obviously, not be capable of storing all information.

Using the notation described above, it is possible to describe the different actions which have to be taken when a certain service request is issued. In Annex B the different argument relations for the demonstration program are showed. In this code, some constructs will appear which have not yet been discussed. This will be done in the next section on (dynamic) contexts.

### 5.5.1.3 (Dynamic) Contexts

The example in Annex B shows a problem; it is not possible to use the same filters for several different service requests without extra features. If all the argument relations, as specified in the annex, would be used, filters that are not meant to be activated would be activated; for instance, the value of *fromBM2XBM::* is used in three different places (line 56, 63 and 78). In these cases, the filters *showXBM*, *compressXBM2* and *compressXBM4* would all be activated by that same

argument. This is not what is desired; either no compression, a factor 2 compression or a factor 4 compression is desired. To circumvent these kinds of situations, argument relations can be specified in a so-called *context*. An application can *enter* and *leave* a context. In case a context is entered, the argument relations, which are specified in that specific context, are activated. If a context is left, those argument relations are inactivated.

Contexts are specified by the keyword "CONTEXT" and a symbolic name. An argument relation is valid only for the previously specified context. If no context was specified previous to the argument relation, the argument relation is valid in all contexts. To explicitly specify that argument relations are valid in all contexts, the special symbolic name "-" can be used. It is also possible to specify that an argument relation will be valid in several contexts or only when a group of contexts are entered at the same time:

CONTEXT C1: ...	All argument relations specified between this context specification and the next context specification are only valid when the context "C1" is entered.
CONTEXT C1,C2: ...	All argument relations specified between this context specification and the next context specification are only valid when both, context "C1" and context "C2", are entered. If context "C1" and "C2" are entered, all argument relations specified under this context specification ("C1, C2"), under the context specification "C1" and under the context specification "C2" are valid.
CONTEXT C1: <argument relation X> ...	If an argument relation is specified under two different contexts, the argument relation is valid if at least one of these contexts is entered. Thus in the example, the argument relation "X" is valid if only context "C1" or only context "C2" is entered or if both contexts are entered.
CONTEXT C2: <argument relation X> ...	
CONTEXT -: ...	The argument relations specified between this context specification and the next context specification are always valid regardless of which contexts are entered or left.

In the code in Annex B, three different configurations are described. The relations under the context "-" are always valid. Further, when the context "compress0" is valid, no compression is performed on the data of the XBM-formatted image when it is sent over the data-connection (the result of the filter fromBM2XBM is directly connected to sendData). When the context "compress2" is valid, the data is compressed by a factor 2 before it is sent over the data-connection (fromBM2XBM is connected to compressXBM2 which is connected to sendData). Similar, when context "compress4" is valid, the data is compressed by a factor 4 (fromBM2XBM is connected to compressXBM4 which is connected to sendData). The other contexts ("del1", "del2", "del3", "del4", "del5", "del6") are used for other purposes as will be explained below.

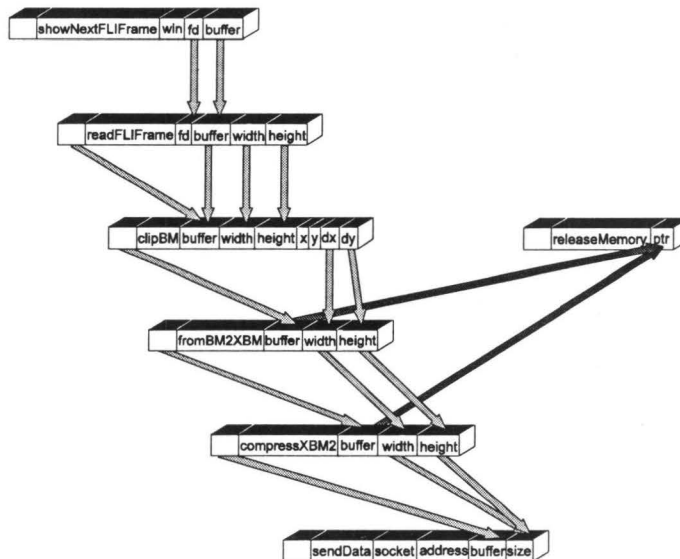
Beside the fact that an application can enter and leave a context, the Quality Factory supports so-called *dynamic contexts*. A dynamic context can be used when a context should be entered for the duration of the activation of a filter only. The context is entered just before the filter is activated and after the filter has finished its task, the dynamic context is left again. In that way, a filter can be used several times within the same service request, each time with different values for its input arguments. Any context can be used in a dynamic way.

The dynamic use of a context is indicated by appending the text 'using <contextA>, <contextB>, ...' after an argument relation which uses of the <- operation<sup>5</sup>. The effect of the dynamic context is

<sup>5</sup> In the case of a <# relation, the usage of a dynamic context makes no sense as no information is propagated due to the use of <#.

that the filters, which need to be activated to compute the value of the argument at the left side of the `<` operation, are activated as if the current entered contexts are entered as well as the contexts `<contextA>`, `<contextB>`, etc. For all other filters it holds true that if the contexts `<contextA>`, `<contextB>`, etc. were not already entered, they are not considered entered now. Thus, for this last group of filters, a dynamic context has no effect on the activeness of these filters.

This will be illustrated by the demonstration program. A special filter (not mentioned before) is introduced in the code: `releaseMemory`. This filter takes a pointer to a memory block and will release the memory, pointed to by this pointer, for other purposes (compare the C/C++ statement `free`). The use of this filter is clear; each conversion filter in the demonstration program allocates a new block of memory to return its result. However, these filters cannot de-allocate this memory as this memory block (read: their result) is used by others after the filter finishes. The dynamic context of the Quality Factory makes it possible to use the filter `releaseMemory` for all these filters. The procedure will be explained for context `"del1"` in line 66 only. The other dynamic contexts work in a similar way. In line 66, the argument relation `"releaseMemory::ptr<-fromBM2XBM::buffer"` is specified using the dynamic context `"del1"`. As a result of this, this argument relation is executed as if, beside the current active contexts (`"compress2"`), also context `"del1"` is active. In this latter context, the argument relation `"releaseMemory::ptr<#fromBM2XBM::"` is defined. This argument relation for context `"del1"` means that the value of `releaseMemory::ptr` may only be used after the argument relations which (indirectly) specify how to compute the value for `fromBM2XBM::` have been executed. Important in this approach is that all arguments of a filter implicitly contribute to the value of the return argument. Thus also `fromBM2XBM::buffer` contributes to the calculation of `fromBM2XBM::`. Therefore, the value of `fromBM2XBM::buffer`, which is assigned to `releaseMemory::ptr`, may only be used after the value of `fromBM2XBM::` is calculated (and thus after the execution of filter `fromBM2XBM` is finished). At that point in time, the information `fromBM2XBM::buffer` points to (the result of `clipBM`) is not used anymore and may be destroyed. Note that the relation under context `"del1"` is only relevant for the argument relations which use the construct `"using del1"`. For all other argument relations, the relation `"releaseMemory::ptr<#fromBM2XBM::"` does not have to be considered.



The diagram above shows the connections between part of the filters of the demonstration program when context "compress2" is entered. The dark gray arrows represent the argument relations which are specified using a dynamic context. In this case, the filters `fromBM2XBM` and `compressXBM2` both use the same filter `releaseMemory` to free some of the memory that has been allocated and will not be used anymore:

### 5.5.1.4 Summary

The previous sections described how the abstraction of the different filters in the Quality Factory have to be specified. Because the approach taken here uses the arguments of the filters as basis for describing the argument relations, a very flexible system results which is able to connect arbitrary filters to each other: data can be converted to the right format and different configurations can be specified (using contexts) to describe the behavior of the filters under different circumstances.

### 5.5.2 Selection

The selection process of software reuse deals with locating, comparing and selecting the artifact which can be best reused in the given circumstances. For this process, it is necessary that a good abstraction of the different artifacts is available. Without this abstraction, it is difficult to separate the artifacts that perform the desired task from the large collection of other artifacts. It is also difficult to compare the different artifacts which perform the desired task when no accurate description of these artifacts is available; when the functionality of the different artifacts is only described partially or not at all, the artifacts can be compared with only a small part of their functionality.

In the Quality Factory, the selection process is important but can also become very complex. Due to the fact that filters and provided services may be dynamically added to and removed from the Quality Factory, the selection process, which has to select which filter to activate next, changes with each of these alterations. Furthermore, the selection process has to deal with the fact that service requests may activate filters and provide services which compete with each other for the same resource.

The selection of which filter to use is guided by the negotiation process (some filters will not be able to perform their task using the limited capacities of the different resources and can therefore not be chosen), the way in which formats have to be translated and the original service request that was made. The description of the different filters with respect to the formats supported by them and the effects on possible claims for the different resources during the processing of the original service request is not only very important, but it is also the proper way to describe the filters as this information is exactly what is needed for the selection process.

The actual selection process in the Quality Factory is done by entering and/or leaving certain contexts. As a result of this, negotiation filters will have two tasks:

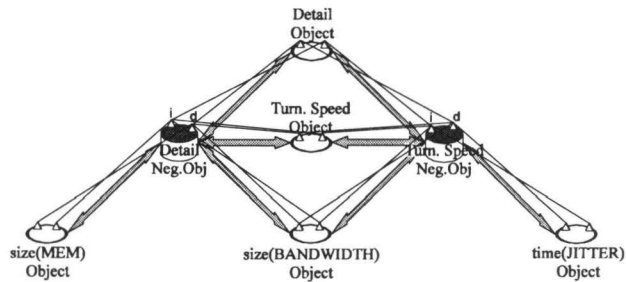
- 1) Negotiate with other negotiation filters over the claims made for the different resources.
- 2) Enter and/or leave contexts depending on the claims which are honored and the extent to which they are honored.

For the second task, the negotiation filter will need to have some information regarding to which context to enter/leave for which values of the perceptual characteristics. In the demonstration program, the following configuration could be used:

level of detail		context to enter
0%	- 25%	compress4
26%	- 50%	compress2
51%	- 100%	compress0

This information has to be given to the negotiation filter in advance (i.e. before it starts negotiating with other negotiation filters). In Annex C an impression is given of the interface of the 'NegotiationObject' object class which was mentioned before in §5.4. In this interface, member functions are provided which can be used to add trade-offs and information with regard to which context to set for which claim. As an illustration, part of the implementation of the 'NegotiationObject' is also presented to give the reader an idea of how negotiation takes place in the Quality Factory. The 'NegotiationObject' itself is a base class which is used to define the actual negotiation objects and which provides some basic functionality for negotiation objects. The way in which trade-offs are realized is using constraint relations; in this way the activation of a negotiation object can be done using the triggering mechanism of the constraint system which allows for transparent negotiation initiation.

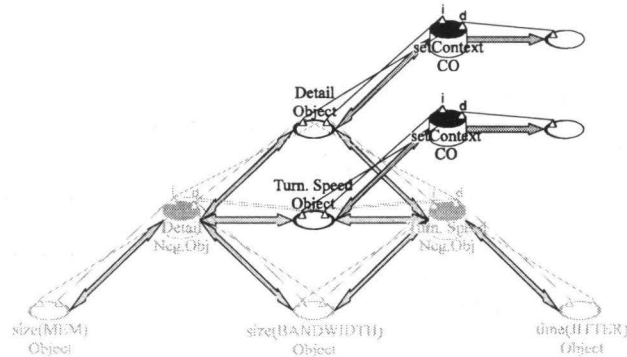
As soon as the filter negotiateDetailLevel is called, the current *claim*, *boundary* and *firmnesses* are stored for the negotiation process and finally the objects in the negotiation process are activated by only setting the new value for the level of detail (line 19). At this point, the constraint system is triggered because the DetailObj is an independent object for the negotiation object 'DetailNO'; if all relations are set for both the level of detail and the turning speed, the following constraint network is created:



Due to the constraint dependencies, the objects which may affect the same characteristics will start to negotiate over the different claims. Because the negotiation objects, which do affect some common characteristic, always form a cycle in the constraint network and constraint objects in cycles are always satisfied before constraint objects which do not lie on a cycle, the entering of a context can be realized via a constraint object which is connected to the cycle but does not lie on the cycle itself. For this reason, for every negotiation object, another constraint object is created which has the *negotiable* object as independent object. The situation thus becomes as shown below.

The new constraint objects in this diagram (setContext) deal with the actual entering and leaving of the contexts. The information which is needed for this, is obtained from the negotiation object. Note, that the new constraint objects both have a dependent object. These dependent objects are in fact dummy objects as they do not serve any purpose but they are required by the constraint system as every constraint object has to have a dependent object.





### 5.5.3 Specialization

Specialization in software reuse deals with the *specializing of generalized (or generic) artifacts*. In general, the reusable artifacts, which have a lot of similarities, are stored in a more generic form. Each of the original functionalities can be retrieved by specializing the generic version through parameters, transformations, constraints or some other form of refinement.

The filters in the Quality Factory can be regarded as examples of such generic building blocks. The exact behavior of a filter (i.e. the values which it produces for its output arguments) can be influenced by changing the values for its input arguments. Specialization in the Quality Factory means that the value of the input argument of the various filters is computed correctly using the output arguments of other filters. Furthermore, the (dynamic) contexts can be used to specify the exact behavior of general filters in different situations. This information is contained in the part of the abstraction which is called argument relations.

### 5.5.4 Integration

Integration deals with the process of connecting the different reused artifacts to each other. For this integration, it is important that it is clear which tasks can be performed by which filters and what kind of parameters are expected by a specific filter for a specific task.

Therefore, integration in the Quality Factory is the process of connecting the different input and output arguments of the various filters to each other and to the arguments of the service request and the provided services. Information about which connections are allowed is defined in the abstraction of the different filters; connections should be made in such a way that the formats of the arguments of both side of the connection match. However, this information is not enough. For the task of integration, more detailed information is necessary; a description of how the different arguments can be connected to each other. This information is, like the specialization information, also contained in the argument relations. The difference between the information for the specialization and the information for the integration lies in the fact that the information for the specialization is contained in how the value of an argument is computed (i.e. the expression on the right-hand side of the operation  $\leftarrow$ ) whereas the information for the integration is contained in which objects are used on the left-hand side of and which objects are used on the right-hand side of both operations  $\leftarrow$  and  $\leftarrow\#$ .



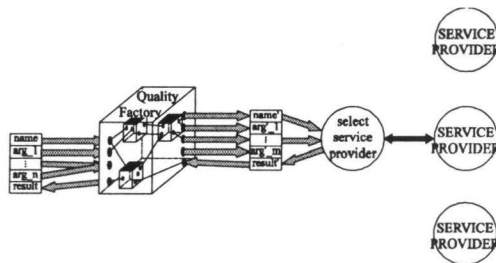
### 5.5.5 Summary

In the previous sections it was explained that the Quality Factory followed the rules of software reuse; abstraction, selection, specialization and integration of the different filters in the Quality Factory were discussed. It was shown that each of these aspects could be identified in the approach taken in the Quality Factory. The advantage of this 'software-reuse' approach is that it is formalized how different filters can be connected in a clear and clean way. Given that each different filter has been proven to work in a correct way it is relatively easy to show that the whole system (i.e. the whole network of connected filters) works correctly by proving that the dataflow between the different filters is correct. The fact that the Quality Factory uses a formal notation to describe these connections makes this process even easier. However, a detailed discussion of such a correctness proof is beyond the scope of this thesis.

### 5.6 Propagation of Information

From the previous sections, it may be clear that vital information is contained in the three subparts of a service request. Therefore, the Quality Factory will use the information in these subparts to propagate information from service requester, via the Quality Factory, to the service provider (as opposed to the situation where the complete service request as a whole is used to propagate information) and back.

During this information propagation, different filters will be activated to perform the necessary translation steps. To this purpose, the information propagation within the Quality Factory may be routed via several filters. The general view of a service request is presented in the figure below:



In this figure it is shown that the information of a service request may be used by several filters in the Quality Factory. Thus the information of the name of the requested service and the values of the different arguments are propagated to the different filters within the Quality Factory. The results of the filters are then propagated to other filters and so on. Once the information from the service request has passed the necessary filters, a service provider is selected by the Quality Factory which may honor the request. After that, the service provider may return a result. This result is propagated back, through the Quality Factory (and possibly again through a sequence of filters), to the service requester.

From the figure, it may also become clear why a service request has been split up into its name, its arguments and its return value; the direct link between service requester and service provider is broken and information in the service request is used to guide the Quality Factory in activating the appropriate filters. Input and output arguments of the various filters may be connected to each other, to the arguments of service requests and to the arguments of provided services to propagate information from one argument to another. Once all the input arguments are assigned a value, the

filter may be activated or a provided service may be executed. The reasons to make a connection between the different arguments are that:

- information to control the different filters is typically contained in the values of the arguments.  
The way a filter should behave is determined by the original service request and by the results of the other filters. The information in the arguments can be used as control information for the various filters; sometimes an argument is used directly as input for a filter, sometimes indirectly when the output argument of another filter, which was directly controlled by an argument of a service request, is used.
- information about the dependencies of the various filters is contained in the arguments of the request (and the output arguments of other filters).  
The order in which several filters have to be applied is already (partially) specified by the description of how to compute the value of an input argument. The dependency of the various filters on the values of the arguments of the service request and on the results of other filters already determines part of the order in which filters have to be activated. A filter which depends on the result of a second filter obviously has to be activated after the latter one is finished.
- information in one argument may be needed by several filters and information from several requests may be relevant for the same filter.  
Parts of the information contained in a service request may be relevant for several filters. In that case, it is more efficient to only pass the relevant information to the filter instead of all the information present in the service request. As this kind of a situation is likely to occur, it will often be necessary to split the information in the service request in different subparts.
- new filters can be easily added in the Quality Factory.  
For new filters to be incorporated in the Quality Factory, only new connections between the arguments of the new filter and the existing ones have to be created. The new connections ensure that the filter will be activated when necessary and in the proper sequence.

The task of the Quality Factory is to propagate the information through the right sequence of filters. More precise, the Quality Factory has to propagate information from argument to argument. The reader may have noticed that the way filters behave is similar to the process of independent objects, constraint objects and dependent objects in the MADE constraint system and that the connections between the different filters, between the service requester and the filters and between the service provider and the filters have similarities with the dependency relations in the MADE constraint system. In the table below, this analogy is elaborated somewhat further. Because of this analogy, the relations in the Quality Factory between the different filters can easily be turned into constraint relations. The process of how these relations are translated from abstract description to concrete constraint relations is the topic of the next section.

Quality Factory	MADE Constraint System
argument of a service request	independent object
issuing the service request (and at the same time setting the value of the different arguments)	triggering member function
input argument of a filter	dependent object
copying the value of the argument of the service request to the input argument of the filter	constraint function

output argument of a filter	independent object
activating the filter (which also sets the output arguments of the filter)	triggering member function
input argument of a filter	dependent object
copying the value of the output argument of the filter to the input argument of the filter	constraint function
output argument of a filter	independent object
activating the filter (which also sets the output arguments of the filter)	triggering member function
argument of a provided service	dependent object
copying the value of the output argument of the filter to the argument of the provided service	constraint function

### 5.7 Towards the Constraint Relations

In this section, it will be shown how the relations between the arguments of service requests, the arguments of the filters and the arguments of the provided services can be specified in terms of constraints. From the description of the argument relations, a mapping can be made to a MADE constraint network and the appropriate independent, dependent and constraint objects as well as the triggering and renewed-triggering member functions. This mapping can be done automatically.

In the remainder of this section, the following topics will be discussed: the structure of the constraint network which will be used (§5.7.1 and §5.7.2), the constraint classes and the constraint functions which are designed for the Quality Factory (§5.7.3). In all these sections it will be shown how constraints are used and what kind of constraint classes are needed to provide the functionality of the Quality Factory.

#### 5.7.1 Creating Objects for the Arguments

The arguments of the service request, the provided service and the different filters will act as the independent and dependent objects in the constraint network. Changes to the values of these arguments are the triggering mechanisms for information propagation. To map the argument relations of the previous section into constraint relations, for every argument (either of a service request, a filter or a provided service), several objects are created in the constraint network. Four different situations can be distinguished in this case; the network around every argument (§5.7.1.1), the additional objects in the network around the result argument (§5.7.1.2), the additional objects in the network around the fields of an argument which type is a *reference* (§5.7.1.3) and the additional objects in the network around arguments which are used during the execution of an argument relation (§5.7.1.4).

##### 5.7.1.1 The Non-Result Argument

For each argument, two objects, in addition to the constraint object, are created. One object, "*Name::argumentName*", is to hold the value of the argument "*argumentName*" of filter/service "*Name*". The other object, indicated by "*X.Name::argumentName*", is used to reset the status (and the value) of the argument<sup>6</sup>. The objects, which values are (indirectly) affected by the value of "*Name::argumentName*", will become an independent object of the *RstCO* constraint object. This ensures that the value of the argument is not destroyed before it has been propagated to all the

---

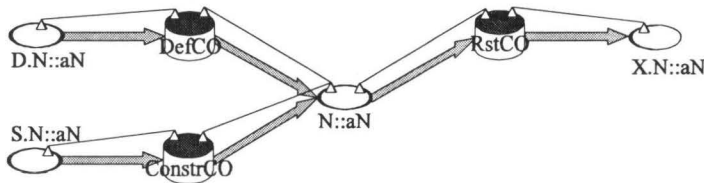
<sup>6</sup> An argument has to be reset after it has been used for one service request to allow it to be used properly in a second service request.

indicated places. The object "*X.Name::argumentName*" itself has no real purpose; it exists because, in the MADE constraint system, a constraint objects needs to have a dependent object.

A third object, "*D.Name::argumentName*" may be used in case an argument is assigned a default value. This object contains the default value and, if triggered, the associated constraint object (*DefCO*) will copy the default value of this object to "*Name::argumentName*". To make sure that a default value does not overwrite an assigned value, default values are not assigned when another value is already assigned to the argument. The object "*D.Name::argumentName*" is only created if a default value is specified.

Also a fourth object, "*S.Name::argumentName*", may be used. This object will be created if the type of *argumentName* is a reference. This object is used to synchronize propagation of information to and from "*Name::argumentName*" with the setting of the different fields of this argument:

- in case the argument is the result argument of a service request or a non-result argument of either a filter or a provided service, the value of the argument may be propagated only after all its fields are assigned a value; in this situation not the value of the different fields should be propagated but the value of the complete reference. "*S.Name::argumentName*" will act as a buffer for information propagation; the paths in the constraint network, which go through one of the fields of argument "*Name::argumentName*", are blocked at object "*S.Name::argumentName*" until propagation in these paths all have reached this object<sup>7</sup>. The constraint object *ConstrCO* will construct the value for "*Name::argumentName*" using the values of the different fields<sup>8</sup>:

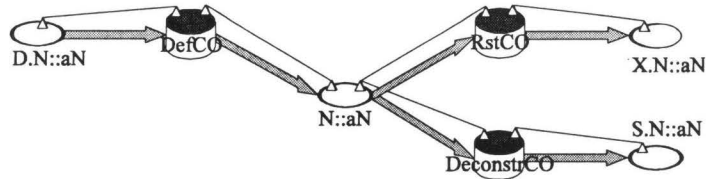


- in case the argument is a non-result argument of a service request or the result argument of either a filter or a provided service, the value of the different fields of the argument may be propagated only after the argument itself is assigned a value. "*S.Name::argumentName*" will act as a buffer for information propagation; the paths in the constraint network, which go through the argument "*Name::argumentName*", are blocked at object "*Name::argumentName*" until propagation in these paths all have reached this object<sup>9</sup>. At that point, the constraint object *DeconstrCO* will retrieve the values of the different fields of "*Name::argumentName*" using the value of the argument itself:

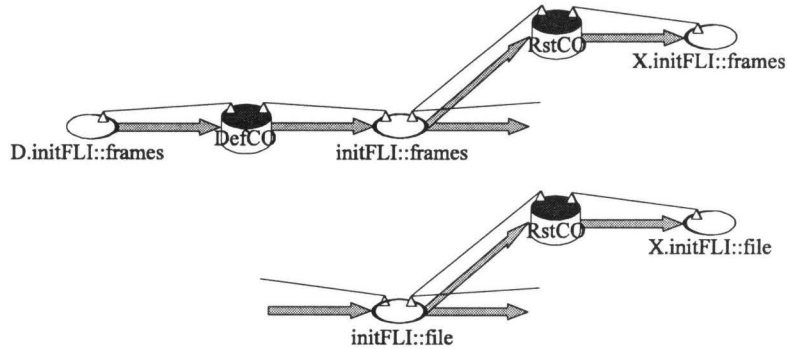
<sup>7</sup> Details about this buffering are described in §5.7.1.3.

<sup>8</sup> In the diagrams '*Name::argumentName*' will be abbreviated to '*N::aN*' and '*Name::argumentName\_fieldName*' to '*N::aN\_fn*'.

<sup>9</sup> Details about this buffering are described in §5.7.1.3.



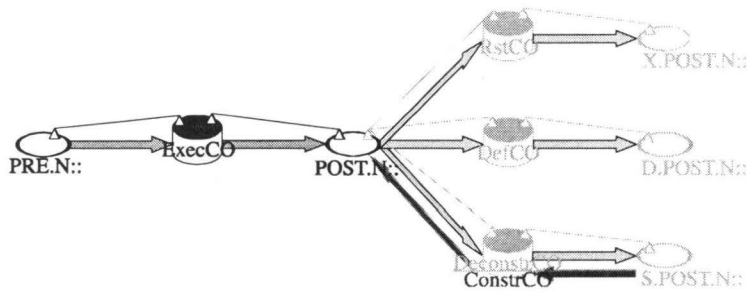
The diagram below shows part of the constraint network which is constructed for the service *initializeFLI*. The complete constraint networks which are created for *initializeFLI*, *showNextFLIFrame* and *terminateFLI* can be found in the Annex D. This diagram shows the different objects that are created for the argument of a filter. The arrows and lines which are not connected on one side lead to other objects in the constraint network which are not shown for reasons of readability:



#### 5.7.1.2 The Result Argument Object

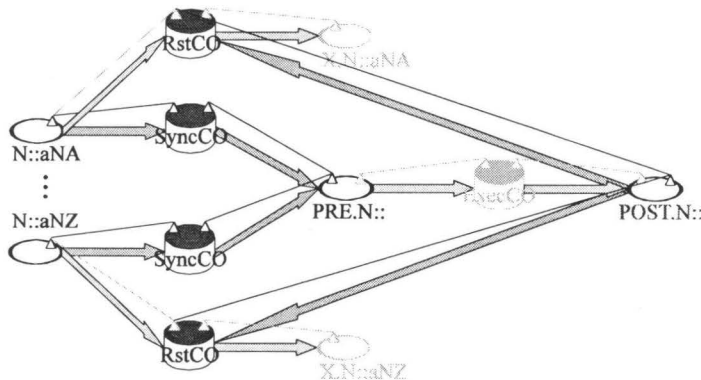
In case of a result argument, a special arrangement has to be made to make sure that the propagation of the information through the various arguments is properly synchronized with the activation of the filter, the execution of the provided service or the return of the result to the service request. For that purpose, two objects, apart from the constraint object, are created for every result argument. One object, which will be named "PRE.Name::", will be the independent object of the new constraint object whereas the other object, named "POST.Name::", will be its dependent object. The constraint object, *ExecCO*, connects these two objects. Its constraint function will,

- in case the result argument is an argument of a filter "Name", activate the filter and copy the result to the "POST.Name::" argument object.
- in case the result argument is an argument of a provided service "Name", request the service from the provider and copy the result to the "POST.Name::" argument object.
- in case the result argument is an argument of a service request "Name", wait until a result is stored in the "POST.Name::" argument object.



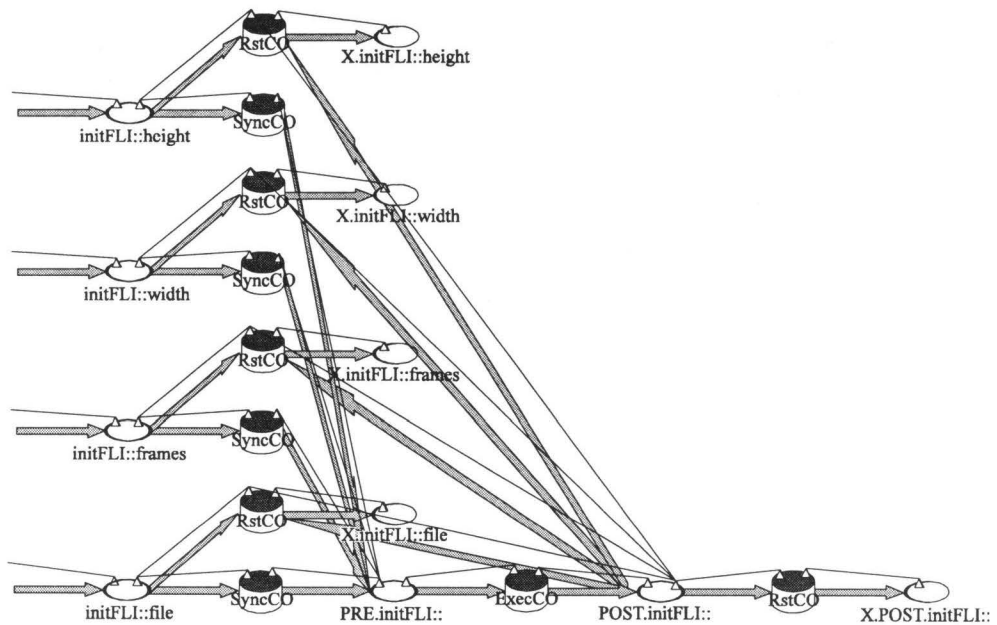
The object "POST.Name::" holds the value of the return argument of filter/service "Name". All references to "Name::" in the argument relations are replaced by references to "POST.Name::" during the translation to constraint relations. For this object, the same additional objects can be created as presented in the previous section; "X.POST.Name::", "D.POST.Name::" and "S.POST.Name::". The object "PRE.Name::" itself serves no special purpose; it is a dummy object to converge different paths in the constraint network. Only the object "PRE.Name::" exists (and not the objects "X.PRE.Name::", "D.PRE.Name::" and "S.PRE.Name::"). The independent object "PRE.Name::" acts as a kind of synchronizer:

- in case of a filter or a provided service, it will act as a buffer for information propagation; the paths in the constraint network, which go through one of the arguments of filter/provided service "Name", are blocked at object "PRE.Name::" until propagation in these paths have all reached this object. This behavior is accomplished by connecting all the arguments of a filter/provided service to the object "PRE.Name::" via a constraint object of the class *SyncCO*. The constraint functions of these objects are empty; no information has to be propagated from the arguments to "PRE.Name::". However, by creating this link, there is also an (indirect) link between every object that is used in setting the value of one of the arguments of "Name" and the object "PRE.Name::". Due to the procedures, followed by the MADE constraint system, to propagate information through the constraint network, this link makes sure that a filter will be activated and that a provided service will be executed only after all their arguments are set:



- in case of a service request, it will act as a buffer for information propagation; the paths in the constraint network, which go through "POST.Name::", are blocked at object "PRE.Name::" until propagation via these paths have all reached this object. Multiple paths exist primarily in the cases where "POST.Name" is used as the destination in synchronization argument relations (<#). The buffering behavior will be accomplished by introducing extra dependency relations in the constraint network; all the constraint objects which have "POST.Name::" as dependent object will also have "PRE.Name::" as dependent object. In this way, it is assured that the information in "POST.Name::" will not be propagated by the MADE constraint system before the constraint function of *ExecCO* is executed.

The diagram below shows another part of the constraint network for the service *initializeFLI*. This diagram shows the result argument of filter *initFLI*:

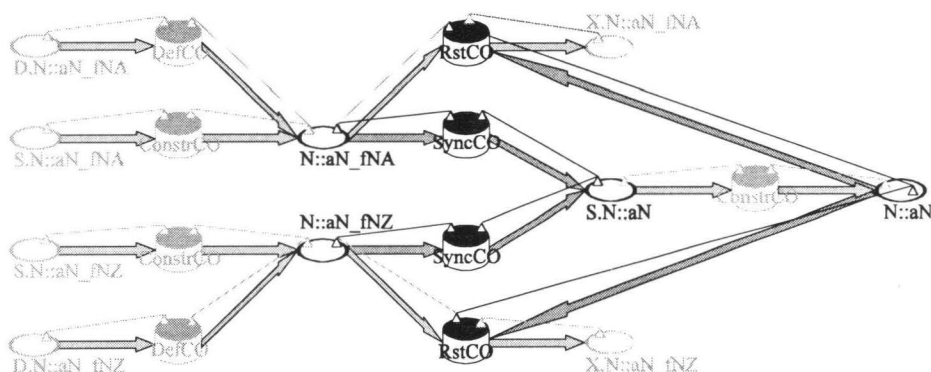


### 5.7.1.3 The Fields of a Reference

For every argument "Name::argumentName" which type is a reference, an object "S.Name::argumentName" was created. This object's task is to synchronize the setting of the value of the different fields of the reference and the propagation of the value of the argument.

There were two cases; a construction and a deconstruction case. In both cases, the different fields of the reference can be treated as ordinary arguments. Because of that, for each field, the

objects "Name::argumentName\_fieldName", "X.Name::argumentName\_fieldName", "D.Name::argumentName\_fieldName" and "S.Name::argumentName\_fieldName" can be created (the same way as explained in §5.7.1.1). When the value of a reference is constructed from the value of its fields, the following structure is created:



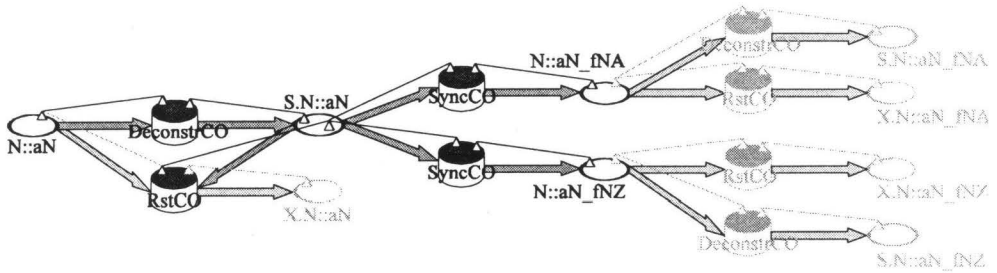
The fields are connected to the object "S.Name::argumentName" via *SyncCO* constraint objects; no information is propagated through these links but the structure of (in)dependent objects and constraint objects is important with respect to the buffering of the propagation via object "S.Name::argumentName". Due to the procedures, followed by the MADE constraint system, this link makes sure that the construction of the reference value will take place only after the values of all the fields are set.

The objects for the different fields themselves also have objects to reset their status. The value of the different fields may, of course, only be reset after the value of the reference is constructed. Therefore, the object "Name::argumentName" is made an independent object to the *RstCO* constraint objects: the MADE constraint system will not execute the constraint function of *RstCO* before the constraint function of *ConstrCO* is executed.

Furthermore, the different fields may also have default values and the fields may be of the reference type. In the latter case, the object "S.Name::argumentName\_fieldName" is always the independent object of the *ConstrCO* constraint object for which "Name::argumentName\_fieldName" is the dependent object. This is because the direction of the propagation of information will always be in the same direction as the propagation of information between "S.Name::argumentName" and "Name::argumentName".

When the values of the fields are retrieved from the value of the reference, a structure as presented below is created. The fields are connected to the object "S.Name::argumentName" via *SyncCO* constraint objects; no information is propagated through these links but the structure of (in)dependent objects and constraint objects is important with respect to the buffering of the propagation via object "S.Name::argumentName". Due to the procedures, followed by the MADE constraint system, this link makes sure that the retrieval of the fields takes place only after the value of the reference is set.



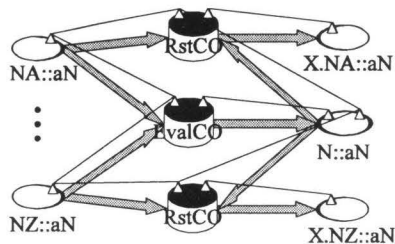


The objects for the different fields themselves also have objects to reset their status. The value of the reference may, of course, only be reset after the value of the fields is retrieved. Therefore, the object "S.Name::argumentName" is made an independent object to the *RstCO* constraint objects. The MADE constraint system will not execute the constraint function of *RstCO* before the constraint function of *DeconstrCO* is executed.

Furthermore, the different fields may not have default values; the value for each field is present in the value of the reference. The fields may be of the reference type. In the latter case, the object "S.Name::argumentName\_fieldName" is always the dependent object of the *DeconstrCO* constraint object for which "Name::argumentName\_fieldName" is the independent object. This is because the direction of the propagation of information will always be in the same direction as the propagation of information between "S.Name::argumentName" and "Name::argumentName".

#### 5.7.1.4 Execution of Argument Relations in Normal and Dynamic Contexts

Whenever the value of an argument must be computed using one of the argument relations specified in a non-dynamic context, a new constraint class is used. This new constraint class is the *EvalCO*. Instances of this class take this argument relation and compute and store the result of this computation in the argument object specified on the left-hand side of the relation. The independent objects of the *EvalCO* object are the argument objects specified on the right-hand side of the relation ("NameA::argumentName", ..., "NameZ::argumentName") and the dependent object is the argument object specified on the left-hand side of the relation ("Name::argumentName"). To prevent situations where the values of the independent objects of the *EvalCO* object are reset before they are used in the evaluation of the value of the dependent object, the dependent object has been made an independent object of the *RstCO* objects of those independent objects:



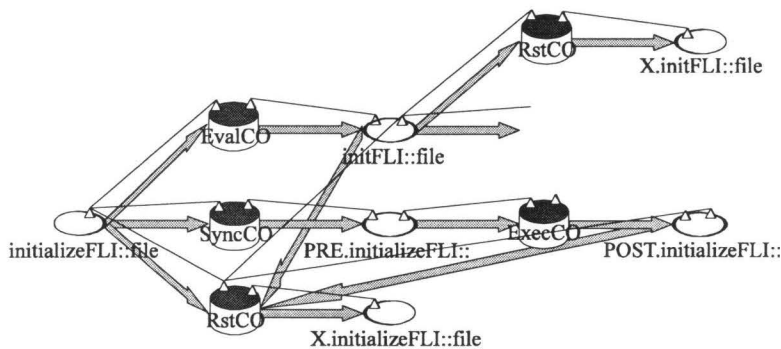
When a value for an argument is stored in an argument object, this value is stored together with a reference to all the contexts which were entered at the moment of storage. Furthermore, values for the same argument are not overwritten when they are computed using a different set of entered contexts. This is necessary for the Quality Factory to support dynamic contexts when using the MADE constraint system; in the MADE constraint system, values are propagated from a certain point in the constraint network only when propagation in all paths, which go through this point, have actually reached that point. This means that values for the same argument, computed in different (dynamic) contexts, need to be stored until they can be propagated. Afterwards, it is necessary to be able to retrieve the different values for the different sets of contexts. This retrieval is guided by the set of contexts, *the retrieve-set*, for which a value has to be retrieved. For the retrieval of a value it is not necessary to specify the complete set of contexts under which a value is stored; it is possible to specify a retrieve-set which is only a subset of this original set of contexts for which a value was stored. The rules regarding the retrieval of values under different contexts are:

- if there is a value stored under the same set of contexts as specified in the retrieve-set, this value is returned.
- if there exists at least one subset of the retrieve-set for which a value is stored, the value associated with the largest of these subsets is returned. If several of such largest subsets exist, the Quality Factory will raise an error as a result of the ambiguity.
- if no subset of the retrieve-set exists for which a value is stored, the default value for the argument will be returned. If no default value is specified, the Quality Factory will raise an error as a result of the lack of a proper value.

The introduction of a dynamic context has no effect on the constraint network as a whole. However, to support dynamic contexts, the constraint object *EvalCO* must make a distinction between whether an argument relation is executed in a normal or a dynamic context. In the latter case, extra care has to be taken: the values of the argument objects on the right-hand side of the relation must be retrieved as if the dynamic context was not entered (these values were also stored when the dynamic context was not entered) while the value for the argument object on the left-hand side should be stored when the dynamic context is entered. For this purpose, the *EvalCO* constraint object must keep some administration.

Furthermore, the constraint object *ExecCO* must also do some extra bookkeeping. This is necessary because the filter which is activated by the constraint object *ExecCO* must be activated as often as its arguments are used in a dynamic context.

In the diagram below part of the constraint network for the service *initializeFLI* is shown which demonstrates how an *EvalCO* object is embedded in the constraint network:



### 5.7.2 The Bookkeeping of the Argument Objects in the Quality Factory

For each argument, created as described in the previous section and maintained by the Quality Factory, some information is stored. This information is described here so the reader will be able to understand the specifications of the different constraint functions, described in the next section, somewhat better:

- the type of the argument
  - this can be either
    - result argument service request, non-reference type
    - non-result argument service request, non-reference type
    - result argument filter/provided service, non-reference type
    - non-result argument filter/provided service, non-reference type
    - result argument service request, reference type
    - non-result argument service request, reference type
    - result argument filter/provided service, reference type
    - non-result argument filter/provided service, reference type
- a status flag holding information about the current value (not assigned to, default value assigned to, value assigned to)
- the current value
- the size of the argument's type
- the alignment used of the argument's type
- a list to possible sub-arguments
  - this list is used differently by the each type of objects:
    - "D.Name::argumentName" objects use the list to point to the object "Name::argumentName".
    - "S.Name::argumentName" objects use the list to point to the object "Name::argumentName" and to all the fields of its type.
    - "X.Name::argumentName" objects do not use the list.
    - "PRE.Name::" objects do not use the list.
    - "POST.Name::" objects use the list to point to the function which provides the functionality for "Name" and to all the arguments of "Name".

Every argument object has a private variable *Bookkeeping* which has the following type:

```
struct ObjectBookkeepingStruct {
    int    typeOfObject;
    int    statusOfValue;
    void    *ptrToCurrentValue;
    int    sizeOfValue;
    int    alignmentForValue;
    void    *subList;
};
```

The field *subList* is used for many purposes. Often it points to the following structure:

```
struct chainedList {
    void    *item;
    chainedList* next;
};
```

### 5.7.3 Definitions of the Constraint Classes

In the previous sections, some of the used constraint classes have already been introduced. The Quality Factory makes use of seven constraint classes. These seven classes are:

- **ConstrCO: *Construction of a reference* Constraint Object**  
construct the value of a reference from its individual fields.
- **DeconstrCO: *Deconstruction of reference* Constraint Object**  
retrieve the values of the individual fields from the value of a reference.
- **EvalCO: *Evaluation of a argument relation* Constraint Object**  
compute the value of an argument based upon a certain expression.
- **SyncCO: *Synchronization of paths* Constraint Object**  
synchronize the propagation of information through the constraint network.
- **ExecCO: *Execution of services* Constraint Object**  
activate a filter, execute a provided service or return a result to a service request.
- **DefCO: *Default value* Constraint Object**  
copy a default value to an argument.
- **RstCO: *Reset argument* Constraint Object**  
reset the value of an argument so default and/or assigned values are removed to ensure that the behavior of this argument object is the same in successive calls.

In the next sections the constraint functions of the different constraint classes will be described to complete the description of the Quality Factory.

#### 5.7.3.1 The ConstrCO Constraint Object

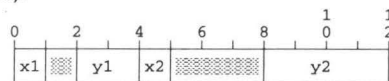
The constraint objects which are instances of *ConstrCO* always have a "*S.Name::argumentName*" as independent object and a "*Name::argumentName*" as dependent object. Given a pointer to the argument object "*S.Name::argumentName*", it is possible to construct the value for the argument. Therefore, *ConstrCO* constraint objects have a pointer to the "*S.Name::argumentName*" argument object stored in a private variable.

Structures are constructed in a simple way; fields are copied into an array of memory cells at the first proper alignment for that field and after the previous field. The fields are ordered in the way as specified in the description of the reference. The array of memory cells is aligned in such a way that the alignment of all types lies on the first cell.

An alignment of the reference {char x1, int y1, char x2, long y2} where:

- a char is aligned on a character boundary (i.e. on an offset divisible by 1) and has a size of 1 memory cell.
- an int is aligned on a word boundary (i.e. on an offset divisible by 2) and has a size of 2 memory cells.
- a long is aligned on a double word boundary (i.e. on an offset divisible by 4) and has a size of 4 memory cells.

would look like (■ means unused):



In pigeon-C, the constraint function looks like:

```
void cfConstrCO () {
    /* argumentPntr is the private variable which points to the "S.Name::argumentName" argument object */
    /* sizeOfValue is the variable in the argument object which holds the size of the space required to store the
```

```

        value of the argument */
/* alignmentForValue is the variable in the argument object which holds the alignment for the space required
   to store the value of the argument */
/* ptrToCurrentValue is the variable in the argument object which holds the current value of the argument */
/* subList is the variable in the argument object which holds the pointer to the list of arguments */

char* refMemory = (char*) malloc (argumentPntr->Bookkeeping->subList->item->sizeOfValue)
int refOffset = 0;
chainedList* fieldPntr = argumentPntr->Bookkeeping->subList->next;
while (fieldPntr->item != NULL) {
    refOffset = ((refOffset-1) & (~ (fieldPntr->item->alignmentForValue - 1))) + fieldPntr->item->alignmentForValue;
    copyValueFromTo (fieldPntr->item->ptrToCurrentValue, refMemory+refOffset, fieldPntr->item->sizeOfValue);
    refOffset += fieldPntr->item->sizeOfValue;
    fieldPntr = fieldPntr->next;
};

copyValueFromTo (refMemory, argumentPntr->Bookkeeping->subList->item->ptrToCurrentValue,
    argumentPntr->Bookkeeping->subList->item->sizeOfValue);
free (refMemory);
};

```

### 5.7.3.2 The DeconstrCO Constraint Object

The constraint objects which are instances of *DeconstrCO* always have a "Name::argumentName" as independent object and a "S.Name::argumentName" as dependent object. Given a pointer to the argument object "S.Name::argumentName", it is possible to construct the value for the argument. Therefore, *DeconstrCO* constraint objects have a pointer to the "S.Name::argumentName" argument object stored in a private variable.

The values of the fields are retrieved from the array of memory cells in the same way as they were stored in the array in the previous section. The difference between deconstruction and construction is that the values are copied in a different direction; deconstruction copies the values from the array to the argument objects.

In pigeon-C, the constraint function looks like:

```

void cfDeconstrCO () {
/* argumentPntr is the private variable which points to the "S.Name::argumentName" argument object */
/* sizeOfValue is the variable in the argument object which holds the size of the space required to store the
   value of the argument */
/* alignmentForValue is the variable in the argument object which holds the alignment for the space required
   to store the value of the argument */
/* ptrToCurrentValue is the variable in the argument object which holds the current value of the argument */
/* subList is the variable in the argument object which holds the pointer to the list of arguments */

char* refMemory = (char*) malloc (argumentPntr->Bookkeeping->subList->item->sizeOfValue)
int refOffset = 0;

copyValueFromTo (argumentPntr->Bookkeeping->subList->item->ptrToCurrentValue, refMemory,
    argumentPntr->Bookkeeping->subList->item->sizeOfValue);

chainedList* fieldPntr = argumentPntr->Bookkeeping->subList->next;
while (fieldPntr->item != NULL) {
    refOffset = ((refOffset-1) & (~ (fieldPntr->item->alignmentForValue - 1))) + fieldPntr->item->alignmentForValue;
    copyValueFromTo (refMemory+refOffset, fieldPntr->item->ptrToCurrentValue, fieldPntr->item->sizeOfValue);
    refOffset += fieldPntr->item->sizeOfValue;
    fieldPntr = fieldPntr->next;
};
}

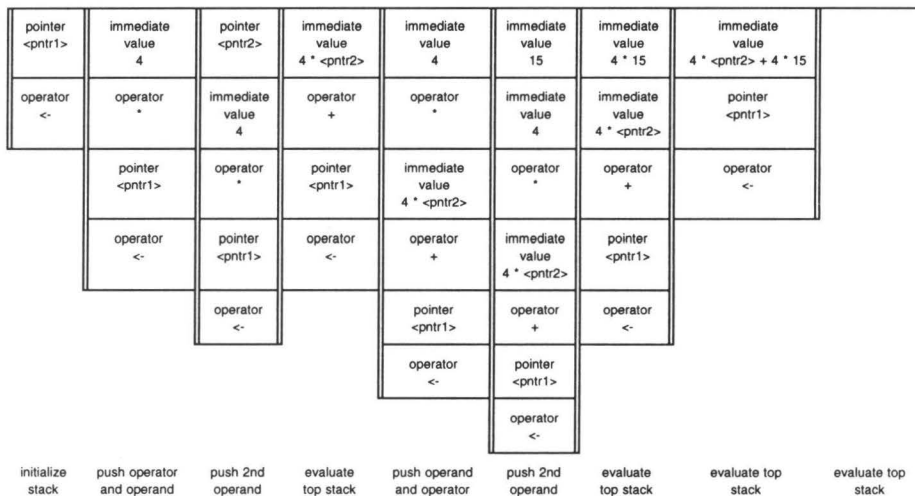
```

```
};
free (refMemory);
};
```

### 5.7.3.3 The EvalCO Constraint Object

The constraint objects which are instances of *EvalCO* always have "*Name::argumentName*" objects as independent and dependent objects. Given a pointer to an evaluation string, it is possible to compute the value for "*Name::argumentName*". This pointer to the evaluation string is stored in a private variable. This evaluation string is a modified version of the argument relation: each symbolic name of an argument is replaced by a pointer to the object. The argument relation

`ServiceRequestName::<-4*ServiceProvidedName::shortArgument+4*15` would thus become the evaluation string "`<pntr1>, <-, 4, *, <pntr2>, +, 4, *, 1510`". Elements of this string are pushed on a so-called *evaluation stack*. On this stack first the operator and then the first operand are pushed. If the next operator has a lower priority than the operator on the stack, the top of the stack is evaluated together with the second operand of the stack's operator (which is not yet pushed onto that stack). After that, the new operator and the result are pushed on the stack. If the new operator has an equal or higher priority, the operator is pushed onto the stack together with the operand which is the first operand of the newly pushed operator. For the example above this would lead to the following situations:



Instances of *EvalCO* also have to keep some extra administration in case they are used in dynamic contexts. This information is also stored in the private variable; the private variable *argumentPtr* points to a *chainedList* whose first element is a pointer to the list of contexts to enter and as remaining elements it stores the pointers to the evaluation string.

<sup>10</sup> Assumed is that pointer <pntr1> points to the argument object "ServiceRequestName::" and that pointer <pntr2> points to the argument object "ServiceProvidedname::shortArgument".



The pigeon-C code of the constraint function looks like:

```
void cfEvalCO () {
    /* argumentPtr is the private variable which points to a list of dynamic contexts and to the evaluation string */
    /* sizeOfValue is the variable in the argument object which holds the size of the space required
       to store the value of the argument */
    /* ptrToCurrentValue is the variable in the argument object which holds the current value of the argument */
    /* subList is the variable in the argument object which holds the pointer to the list of arguments */

    chainedList* = evalElement = argumentPtr->Bookkeeping->subList->next->next->next;

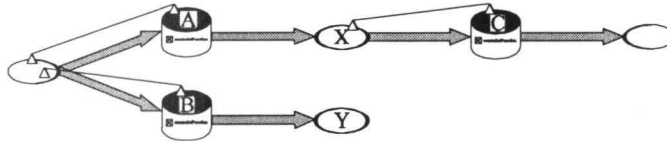
    stack* evalStack = empty;
    push (evalStack, argumentPtr->Bookkeeping->subList->next->next->item);
    push (evalStack, argumentPtr->Bookkeeping->subList->next->item);

    while (evalElement != NULL) {
        push (evalStack, evalElement->item);
        if (evalElement->next != NULL) {
            if (priorityLowerThan (evalElement->next->item, stackElem(evalStack, 2)) ) {
                while ( priorityLowerThan (evalElement->next->item, stackElem(evalStack, 2)) ) {
                    result = performOperation (stackElem(evalStack, 2), stackElem(evalStack, 3), stackElem(evalStack, 1));
                    pop (evalStack); // pop second operand from stack
                    pop (evalStack); // pop first operand from stack
                    pop (evalStack); // pop operator from stack
                    push (evalStack, result) // push result as second operand on stack
                };
                pop (evalStack); // pop the first operand of new operator from stack
                push (evalStack, evalElement->next->item); // push new operator on stack
                push (evalStack, result); // push first operand on stack
            }
            else {
                pop (evalStack); // pop first operand of new operator from stack
                push (evalStack, evalElement->next->item); // push new operator on stack
                push (evalStack, evalElement->item); // push first operand on stack
            };
        }
        else {
            while (! empty (evalStack)) {
                result = performOperation (stackElem(evalStack, 2), stackElem(evalStack, 3), stackElem(evalStack, 1));
                pop (evalStack); // pop second operand from stack
                pop (evalStack); // pop first operand from stack
                pop (evalStack); // pop operator from stack
                push (evalStack, result); // push result as second operand on stack
            };
            pop (evalStack);
            copyValueFromContextToContext (result, argumentPtr->Bookkeeping->subList->item->ptrToCurrentValue,
                argumentPtr->Bookkeeping->subList->item->sizeOfValue, currentContexts,
                argumentPtr->Bookkeeping->subList->item);
        };
        evalElement = evalElement->next->next;
    };
};
```

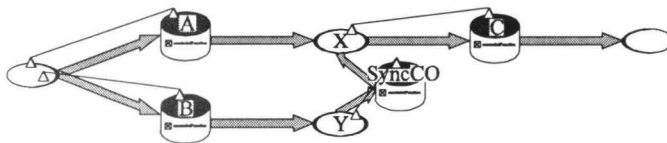
### 5.7.3.4 The SyncCO Constraint Object

The constraint functions of the instances of the *SyncCO* constraint class have an empty body; these constraint objects are automatically satisfied after they are triggered.

The use of the SyncCO constraint objects is based upon the fact that the MADE constraint system is satisfied in such a way that a constraint object is not invalidated due to the satisfaction process of another constraint object within the same execution plan. This can be illustrated by the following constraint networks. In the diagram below, the order in which the constraint objects *A* and *B* are satisfied is not determined; they can be satisfied in parallel, *A* can be satisfied before *B* or, the other way around. This means that no guarantees can be given as to whether information of object *X* is propagated before or after object *Y* is updated:



To define the relative order between object *X* and *Y*, for instance object *Y* should be updated before the information of object *X* is propagated, a new constraint object *SyncCO* can be introduced which has *Y* as independent object and *X* as dependent object:



In this case, the constraint function of *SyncCO* will be executed before the constraint function of *C*; the execution of the constraint function of constraint object *SyncCO* can invalidate constraint object *C* and therefore the MADE constraint system decides to satisfy the constraint object *SyncCO* before *C*. Because of this partial ordering, the information in object *X* will not be propagated through constraint object *C* before constraint object *SyncCO* is satisfied which, in its turn, will not occur before the value of object *Y* is set.

#### 5.7.3.5 The ExecCO Constraint Object

The constraint objects which are instances of *ExecCO* always have a "PRE.Name::" as independent object and a "POST.Name::" as dependent object. Given a pointer to the argument object "POST.Name::", it is possible to retrieve the values of the arguments of "Name". For this purpose, *ExecCO* constraint objects have a pointer to the "POST.Name::" argument object stored in a private variable.

The constraint function has to retrieve the values of the different arguments, put them into a special arguments list and activate the filter/execute the provided service. This process has to be repeated for each dynamic context in which the filter is used. For this purpose, the instance of the *ExecCO* constraint class has stored a pointer in its private variable to point to the list of dynamic contexts in which the filter was used. This list will be used to retrieve the values of the arguments as stored under each dynamic context and execute the filter for each of these dynamic contexts (thus the filter will be executed as often as there are elements in the list of dynamic contexts).

In pigeon-C, the constraint function looks like:



```
void cfExecCO () {
    /* argumentPntr is the private variable which points to a list of dynamic contexts and to the
       "POST.Name::" argument object */
    /* sizeOfValue is the variable in the argument object which holds the size of the space required
       to store the value of the argument */
    /* ptrToCurrentValue is the variable in the argument object which holds the current value of the argument */
    /* subList is the variable in the argument object which holds the pointer to the list of arguments */

    Context* cntxtPntr = argumentPntr->Bookkeeping->subList->item;

    while (cntxtPntr != NULL) {
        Args* argumentStruct = empty;

        chainedList* argPntr = argumentPntr->Bookkeeping->subList->next->next;
        while (argPntr->item != NULL) {
            addValueToArgStruct (argumentStruct, argPntr->item->ptrToCurrentValue, argPntr->item->sizeOfValue);
            argPntr = argPntr->next;
        };
        cntxtPntr = cntxtPntr->next;
    };

    result = callFunction (argumentPntr->Bookkeeping->subList->next->item, argumentStruct);
    copyValueFromTo (result, argumentPntr->Bookkeeping->ptrToCurrentValue, argumentPntr->Bookkeeping->sizeOfValue);
};
```

### 5.7.3.6 The DefCO Constraint Object

The constraint objects which are instances of *DefCO* always have a "*D.Name::argumentName*" as independent object and a "*Name::argumentName*" as dependent object. Given a pointer to the argument object "*D.Name::argumentName*", it is possible to assign the default value to the argument. For this purpose, *DefCO* constraint objects have a pointer to the "*D.Name::argumentName*" argument object stored in a private variable.

In pigeon-C, the constraint function looks like:

```
void cfDefCO () {
    /* argumentPntr is the private variable which points to the "D.Name::argumentName" argument object */
    /* sizeOfValue is the variable in the argument object which holds the size of the space required
       to store the value of the argument */
    /* ptrToCurrentValue is the variable in the argument object which holds the current value of the argument */
    /* statusOfValue is the variable in the argument object which holds the status of the value of the argument */
    /* subList is the variable in the argument object which holds the pointer to the list of arguments */

    if ( argumentPntr->Bookkeeping->statusOfValue != valueAssigned ) {
        copyValueFromTo (argumentPntr->Bookkeeping->ptrToCurrentValue,
            argumentPntr->Bookkeeping->subList->item->ptrToCurrentValue,
            argumentPntr->Bookkeeping->sizeOfValue);
    };
};
```

### 5.7.3.7 The RstCO Constraint Object

The constraint objects which are instances of *RstCO* always have a "*Name::argumentName*" as independent object and a "*X.Name::argumentName*" as a dependent object. The object "*X.Name::argumentName*" is merely a dummy object and is only required by the MADE constraint system to provide the *RstCO* constraints with a dependent object. The *RstCO* constraint objects have a pointer

to "*Name::argumentName*" stored in a private variable so they can reset the value of that argument object when necessary (i.e. when the value of "*Name::argumentName*" will not be used anymore). The Quality Factory uses constraints to reset the value of argument objects as constraints can easily model the dependency relations, with respect to which argument needs the value of which other argument, between the different argument objects.

In pigeon-C, the constraint function looks like:

```
void cfRstCO () {
    /* argumentPtr is the private variable which points to the "X.Name::argumentName" argument object */
    argumentPtr->resetValue ();
};
```

## 5.8 Performance of the Quality Factory

The performance of the Quality Factory in this section will be expressed in the number of constraint relations that have to be satisfied to service a viewer's request. Based on the information described in §4.5, the final performance of the Quality Factory can be derived. The choice of which provided service(s) to invoke in order to service a request does not have to be made by the Quality Factory itself; all relations described in the contexts that are entered are enforced whereas the relations which are described in the contexts that are not entered will not be considered (the actual selection is made by the negotiation objects: see §5.5.2).

Whenever a provided service is invoked, its execution will be guided by the satisfaction of a number of constraint relations. First of all, every argument  $N::a_N$  has one constraint relation (*SyncCO*) with  $PRE.N::$  and one constraint relation (*RstCO*) with  $X.N::a_N$ . Furthermore,  $PRE.N::$  has one constraint relation (*ExecCO*) with  $POST.N::$  and  $POST.N::$  has one constraint relation (*RstCO*) with  $X.POST.N::$ . If it is assumed that the number of arguments of service  $N$  is  $A$ , then the total number of constraints to satisfy is  $2*A+2$ .

Additional constraint relations have to be satisfied when the type of either an argument or the result is a reference. In that case,  $2*R+1$  constraint relations (where  $R$  is assumed to be the number of fields present in the reference) have to be satisfied additionally for every argument which type is a reference; every field  $N::a_N\_fN$  has one constraint relation (*SyncCO*) with  $S.N::a_N$  and one constraint relation (*RstCO*) with  $X.N::a_N\_fN$  and  $S.N::a_N$  has one constraint relation (either *ConstrCO* or *DeconstrCO*) with  $N::a_N$ .

Furthermore, the value of every (field of an) argument of a provided service has to be computed via either a *DefCO* constraint relation or via an *EvalCO* constraint relation. This means that for every (field of an) argument an additional constraint relation has to be satisfied. Also the specification of a '<#' relation requires an additional *SyncCO* constraint to be satisfied for every argument specified on the left-hand side of the argument relation. The total costs (in terms of satisfaction of constraint relations) to execute one particular provided service will thus become:

$$(\sum_{a_N} REF(N::a_N) + SYN(N::a_N)) + 2$$

where:  $REF(A) = 3$ , if  $A$  is not a reference  
 $= 2*R+4$ , if  $A$  is a reference and  $R$  is the number of fields that are defined for that reference  
 $SYN(A) = 0$ , if  $A$  is not used in a left-hand side of a '<#' argument relation  
 $= SR$  if  $A$  is used  $SR$  times in the left-hand side of a '<#' argument relation

The number of constraint relations that has to be satisfied to honor a request from the viewer depends on the number of provided services that are needed to realize the viewer's desired request; the costs of the execution of each of these provided services should be added up to compute the total costs. Besides that, the requested service itself requires the satisfaction of several constraint relations:  $2 \cdot A + 2$  where  $A$  stands for the number of arguments. In case of the demonstration program where context *compress2* is entered (and contexts *compress0* and *compress4* not) the total number of constraints that need to be satisfied when the viewer issues a *showNextFLIFrame* request is:

execution of <i>showNextFLIFrame</i>	8
execution of <i>readFLIFrame</i>	14
execution of <i>clipBM</i>	24
execution of <i>fromBM2XBM</i>	11
execution of <i>compressXBM2</i>	11
execution of <i>sendData</i>	14
execution of <i>receiveData</i>	2
execution of <i>decompressXBM2</i>	11
execution of <i>showXBM</i>	20
execution of <i>releaseMemory</i> ( $4 \cdot 6$ )	24
----	
	139 constraint satisfactions

For this number of constraint satisfactions, the data of a video is read from a file, frames are extracted from the video, converted into another format, compressed, send over the network, decompressed again and displayed on the screen. In this example quite a large number of constraint satisfactions are required. This is due to the fact that the primary entities, used by the Quality Factory, are the arguments of every provided service (as opposed to the service itself). However, as it was argued before, the use of the arguments as the primary entity by the Quality Factory allows for a very flexible system. On average, the example program uses eleven (rather simple) constraint satisfactions per provided service that is executed to honor a viewer's request.

To compute the total costs (in terms of C++ function invocations), it is necessary to know that all constraint objects are non-cyclic. Furthermore, most constraint objects (except some *EvalCO* and *RstCO* constraint objects) have only one independent and one dependent object. To be precise, there are eighty constraint objects with one independent object, twentyeight with two independent objects, nine with three independent objects, 2 with four independent objects and 2 with five independent objects. In total, the following result are obtained (see also §4.5.2):

setting up the constraint network (only the part for <i>compress2</i> ):	3685 C++ function invocations
triggering the constraint network	27 C++ function invocations
satisfying the constraint network (assuming $CF_c = 0$ for all $c$ ):	3197 C++ function invocations
	604 list operations

The same argument that was used in §4.5, that computers are developed which become faster and faster, can be used here again; the overhead of a system like the Quality Factory becomes less and less important due to the increasing power of computers while the benefits of the Quality Factory with respect to the flexible applicability, integration, maintainability and reuse of software modules are undeniable.

### 5.9 Conclusions

In this chapter, the Quality Factory was discussed. Important notions in the Quality Factory are services, filters, formats and resources. By connection different filters to each other, the Quality Factory is able to provide more abstract services using a collection of other services, provided by a given set of service providers. These filters are external components which can transform formats

into other formats, perform operations on data in a certain format or manage a resource and provide negotiation on the usage of (a part of) the capacity of that resource.

The Quality Factory is based upon the MADE constraint system. This constraint system provides the Quality Factory with a powerful propagation mechanism which enables it to easily forward information from one filter to another one and from a filter's argument to the argument of another filter. The Quality Factory has defined seven constraint classes to build the constraint network. These constraint classes enable the Quality Factory to support simple expressions between arguments, default values, contexts and dynamic contexts. These features ensure that the most basic relations and also some more sophisticated relations can be expressed in the Quality Factory.

The Quality Factory itself merely forms only a shell around other services. Filters are the foundations for the functionality of and the basis for the services that can be provided by the Quality Factory. The added value of the Quality Factory is the fact that it can combine the different filters into one big system which results in a greater functionality than just the sum of the functionality of the different individual filters:

- the Quality Factory can make connections between the different filters and transform data from one format into another one so that a certain service, which is not available in the required format, can still be provided for that data, by first translating the format, performing the operation and translating the format back again. This process is done transparent to the requester of the service.
- the Quality Factory can provide hooks for negotiation on different resources which allows the different requests to influence each other; different requests can be compared against each other to allow for a utilization of the resource's capacity which is optimal for the given situation. Because the negotiation is not performed by the Quality Factory itself but by third-party negotiation objects, the Quality Factory has no influence on whether or not the optimal solution will be chosen. In fact, the Quality Factory has almost no influence on the negotiation process itself; it only provides different contexts between which the negotiation objects may choose. Given certain conditions (like the history of previous issued requests), negotiation objects may indicate to the Quality Factory that certain contexts should be entered or left. Also the argument relations defined in a certain context cannot be controlled by the Quality Factory; they are a given entity. However, the Quality Factory gives the possibility to perform negotiation on the allocation of resources; the quality of that negotiation is determined by the variety of the available contexts and the ability of the negotiation objects to reach an agreement over the resource allocation.
- the Quality Factory can be adjusted to the situation dynamically; filters can be involved in the process or not depending on the context which is desired. Alternate solutions may be chosen when the original (straightforward) solution to service a request may not be practicable.
- the way the Quality Factory formalizes the description how filters work together may be used to provide an environment in which it is easy to maintain a complex system build from different components and where the correctness of this large system may be proven more easily.

Looking back at what was required of the Quality Factory in Chapter 2, it can be concluded that most of these requirements stated in §2.3 have been realized.

- commands issued by a viewer using perceptual quality metrics can be serviced by the Quality Factory; a viewer's request is converted into (a number of) provided service invocations while the values for the provided service's arguments are extracted from the

- information as provided by that viewer when issuing the request. This translation from service request to provided service invocation is totally transparent to the viewer.
- the argument relations allow the Quality Factory to support arbitrary filters; the way in which a filter is controlled by means of its arguments is described in the arguments relations. As the Quality Factory allows for the manipulation of the dataflow, information can be adapted, via the Quality Factory by using built-in operators and other provided services, to the needs of the particular provided service. This means that third-party filters can be supported by the Quality Factory without having to be prepared for this usage.
  - based on the description of the various argument relations, clustered per context, it is possible to describe different situations in which different scenarios can be followed. Because a context is implemented as a set of constraint relations and because constraint relations can be added dynamically to the system, new contexts can be added dynamically to the Quality Factory. This also means that new filters can be introduced at runtime and be supported by the Quality Factory. And as a new filter can also be connected to other filters in the Quality Factory, more conversions are possible and more complex services can be honored by the Quality Factory.
  - some built-in operations are implemented in the Quality Factory. The operations that were shown in this chapter were relatively simple and straight forward. Important to note is, however, that they showed that it is possible for the Quality Factory to support built-in operations. With relatively small effort other built-in operations can be added to the Quality Factory to support other manipulations of the dataflow. The complexity of these operations can range from simple (like the arithmetical operations shown in this chapter) to really complex (like the conversion of a JPEG-encoded image into a bitmap). Although it is not really the purpose of the built-in operations of the Quality Factory to perform these kinds of intricate manipulations, the last example shows how complex these built-in operations may become and the potential that they can have.
  - contexts also provide a way to negotiate over resource allocation. The Quality Factory supports the entering and leaving of contexts (which is, in fact, the activation and inactivation of constraint relations) and therefore the ability to change the way in which resources are used. Using different filters or using filters in a different way may lead to different resource allocations. By negotiating over the resource allocations, a set of contexts may be identified which provides, when entered, an (semi-)optimal result for the servicing of the required service (in terms of time used, resources used, provided quality, ...) due to a particular allocation of the resources. This negotiation is not done by the Quality Factory itself, but by so-called negotiation objects. However, these negotiation objects can notify the Quality Factory which context to enter to obtain the (in the current situation relatively) best results.

With respect to the response time, needed by the Quality Factory to perform the different manipulations on the dataflow, some questions may be raised. The problem of the Quality Factory is that it is based on a constraint system which is written in a language which is not directly compiled. Both the constraint system as well as the translation step from mC++ to C++ cause inefficiencies in the final executable version of the Quality Factory. Due to this, problems may arise when time-critical services are required by the viewer. §5.8 shows clearly that in the current situation, the Quality Factory introduces quite a lot of overhead. Therefore, it can be concluded that, although the Quality Factory can be used very well for the conversion of perceptual quality metrics to application-oriented metrics, it will probably perform poorly with respect to time-critical demands like latency or the synchronization of different media.

In relation to the previous chapter, this chapter proves that the Quality Factory is a good environment to experiment with the constraints of the MADE constraint system. It shows that the MADE constraint system can deal with complex network relations and that it can perform diverse

actions. Constraint functions can be designed according to the demands of the situation. Furthermore, some of the features of the MADE constraint system are automatically inherited by the systems which are build on top of this system. In the Quality Factory this can be seen in:

- the ability to add new filters dynamically which is inherited by the possibility to dynamically add or remove constraint objects from the constraint network.
- the ability to use contexts which is inherited by the possibility to activate and inactivate constraint objects.
- the ability to synchronize the evaluation of the various argument of different filters which is inherited by the propagation mechanism of the MADE constraint system.
- the ability to make the working of the Quality Factory transparent to the requesters of the factory's services which is inherited by the use of delegation in the MADE constraint system such that constraints can be defined transparent to the independent and dependent objects.

In the end, it can be concluded that the MADE constraint system is very well suited for specifying the different dependency relations in such a complex system as the Quality Factory. Furthermore, the characteristics flexibility, transparency and dynamics, found in the MADE constraint system, have all been inherited by the Quality Factory. All these aspects were very useful in this system. The parallelism in constraint satisfaction was also inherited by the Quality Factory. This feature was not always desired, but the MADE constraint system also provided means to overcome these problems (the implementation of the *SynCO* constraint class). However, it can also be concluded that the current implementation of the Quality Factory is not able the effectively support time-critical requirements.



# CHAPTER 6

Discussions  
and  
Conclusions





In the first chapter of this thesis, four notions were introduced which played a central role in this thesis. One of these notions was human perception. In Chapter 2, examples were given which showed that the human perception was imperfect and that, as a result of that, information could be left out from (among other things) images and sound samples without (noticeably) degrading the perceived quality. The following four characteristics of human perception were mentioned:

- the viewer has a continuous perception.
- the viewer may not perceive all the information that is present in theory.
- the viewer is unaware of the format in which the digital information is stored.
- the viewer is unaware of the resources needed to present the information.

The first aspect addresses the problem of the discrepancy between the sampled (and thus discrete) representation of digital data and the way a viewer interprets this information. The second aspect states that a viewer may not perceive all the information that is presented and thus that less effort can be made (by presenting less information) to realize the same presentation without loss in perceived quality. The last two aspects indicate that a viewer has, in general, no notion of how a computer goes about presenting information to the viewer. This led to three conclusions:

- data can be compressed using a lossy compression technique without a viewer necessarily noticing that information is lost.
- the viewer and the presentation system use different 'languages' where (certain aspects of) presentations are concerned and these languages should be translated into each other so the viewer can use his own (perceptual) notions instead of the system's (technical) terms.
- as the viewer does not know how the computer is going to present information anyway, it is irrelevant to that same viewer if the presentation system would use a different technique (once in a while).

Based on these conclusions, a new system, the Quality Factory, was proposed in Chapter 2. This system, designed for an object-oriented environment, uses the fact that data can be manipulated and/or compressed without perceived quality loss by trying to find a margin that could be used to translate the different 'languages' of viewer and system into each other. For this purpose, the techniques, used by the presentation system to present information, had to be changed. Although the viewer may not care about the fact that the techniques are changed, still some care had to be taken when alterations to the techniques were introduced. One of the most important aspects in this respect was that existing software should still be usable; this meant that the new alterations to the techniques should be transparent to the presentation system. The alterations were, of course, not transparent to the viewer; he would be able to use his own terminology and would not have to worry about formats, structures and network layout.

To provide a transparent way to make the desired alterations, constraint technology was proposed. Chapter 3 described the general notion of constraint programming and a number of constraint systems for object-oriented environments. In most of these examples, objects that had to have the ability to be constrained had to modify their interface. This was in contradiction to the requirements for the Quality Factory. Therefore, a new constraint system was designed in the multimedia environment MADE. This constraint system, described in Chapter 4, allowed arbitrary objects to be constrained without adapting their interface. Transparency was realized by using delegation; objects that had to be constrained could delegate their behavior to so-called constraint objects. These constraint objects would then take the appropriate actions (i.e. adapt the behavior of the constrained object dynamically) to ensure that the constraint relation was maintained.

Together with the required transparency, other features were added to the MADE constraint system. Among these were the dynamics with which constraints could be added and/or deleted from

the constraint network, the flexibility which allowed for constraint objects to be (temporarily) inactivated, dynamic changes to the constraining function of the constraint object, adaptation of the constraint's policy with respect to the propagation strategy (change between the eager and lazy state) and the fact that constraints were satisfied in parallel whenever possible.

The reason for developing a constraint system in a multimedia environment was inspired by the fact that a multimedia environment provides a broad area where a number of fields in computer science are active. In most of these fields constraints can be used to solve sometimes complex problems. The unique character of a multimedia environment is that each of these fields need different kinds of constraints. In that way, the MADE multimedia environment provided an excellent test environment to see whether the delegation mechanism was generally applicable.

The MADE constraint system was used as the foundation of the Quality Factory, described in Chapter 5. The Quality Factory can provide the viewer with a uniform view of the presented information in terms of his own perceptual terminology. To do so, the Quality Factory breaks down the underlying presentation system into small filters; each filter performs some kind of action. Within the Quality Factory, it is possible to have different filters which provide the same functionality. Requests of the viewer are intercepted by the Quality Factory and analyzed. Based on this analysis, a selection of the different filters available can be made so that the request of the viewer will be honored. Filters were divided into four groups:

- filters that transform data from one format into another.
- filters that perform a certain action on data in a certain format.
- filters that perform actions which relied on the underlying network structure.
- filters that negotiate over a certain (perceptual) resource.

By combining the appropriate set of filters, the Quality Factory is able to make up for the fact that the viewer has a continuous perception, does not perceive all the presented information and is unaware of the used format or required resources.

In the Quality Factory the MADE constraint system was used in two ways. In the first place, the process of selecting different filters was guided by negotiation over the different perceptual and/or system resources. This negotiation process, where specialized objects select the appropriate filters which are to be used by the Quality Factory, was implemented by using the constraint system. Negotiation is typically a process where information flows between the different parties of the negotiation process and where parties have to react when another party makes a new proposal. This kind of behavior is precisely what can easily be modeled in the MADE constraint system; which objects have to react when another object makes changes to information which concerns them all. Thus when a negotiation object altered information which was negotiated over by other objects as well, the constraint system made sure that each object concerned was informed about this and allowed them to react. Finally the constraint system was able to tell when an agreement between the different objects was reached and thus when and which choice between the available filters could be made.

The second use of the MADE constraint system in the Quality Factory was to specify the (possible) relations between the different filters. It was argued that the constraint relations should not be specified between the filters themselves, but between the arguments of the filters. The reasons for this decision were that:

- information to control the behavior of a filter is typically contained in the values of (some of) its arguments.
- information about the dependencies between the different filters is contained in the dependencies of the values of the arguments of a certain filter on the values of arguments of other filters.
- information needed by a certain filter - passed in (some of) its arguments - may be stored differently (different number and type of arguments) by other filters on which it depends.

Therefore, the smallest unit of information was not the filter but its argument. The advantages of using constraints in order to specify the relations between the arguments of a filter (in favor of other methods) were threefold:

- because constraints can temporarily be inactivated, relations between filters can temporarily be (dis)regarded.
- because the constraint system implicitly specifies a certain order in which the constraints are satisfied, the execution of the different filters is also ordered by the way in which constraints are placed between these filters.
- because constraints can be added and removed from the constraint network dynamically, it is possible to dynamically change the set of available filters. This means that it should be possible to use the Quality Factory in environments which comply with COMM or CORBA where new software components can be introduced dynamically.

The disadvantage of using the MADE constraint system had to do with the parallel satisfaction of the different constraints; filters needed to be executed one after another. Therefore, a way had to be found to synchronize the execution of the different filters. Because of its general nature, this solution could be implemented by using the MADE constraint system itself; so-called synchronization constraint objects were designed which were able to synchronize the satisfaction of individual constraints.

### 6.1 Discussions on the MADE Constraint System

For the MADE constraint system, an implementation of the system is realized. This constraint system is written in the language mC++, the programming language of the MADE environment, and provides the building blocks for specifying constraint relations between arbitrary mC++ objects. The delegation mechanism used in the MADE constraint system, provides hooks to transparently connect these mC++ objects to the constraint system. Furthermore, the encapsulation of the data of the mC++ objects is ensured as the delegation (and thus the hooks) are placed on the member functions of the objects and not on the private data of the object. An important notion in this respect is that delegation in mC++ does not require the object to have a special interface, but that it can be applied dynamically to arbitrary member functions of the object. In the MADE constraint system, no predefined constraint relations exist. This means that every constraint relation has to be programmed. The lack of predefined constraint relations stems from the fact that the MADE constraint system was designed to be a general constraint system; it must be possible to write constraints in the MADE constraint system for all kinds of different application areas. The advantage of this is that a constraint programmer can define his own constraint relations, tailored to his own needs. Special constructs, provided by the constraint system, allow the programmer to write his own powerful constraint functions. Furthermore, as constraints in the MADE constraint system are realized as objects, libraries of constraint objects can be created that make it possible for a programmer to build up a collection of his own 'basis' constraint relations.

The MADE constraint system is not an integral part of mC++, but programmed as a toolkit using mC++. This means that the current realization of the MADE constraint system is not optimal. As a

result of this, the specification of constraints has to comply with the syntax of mC++; no specialized keywords like 'constraint function', 'independent object' or 'shadow function' can be used in the mC++ program to identify the different components necessary to create a constraint relation. This meant that other tricks, like macros, had to be used to provide an interface which could hide at least most of the necessary details to create a constraint object in mC++. However, this solution is not the most elegant one. The dependence on the mC++ language can also be seen in the separate specification of the declaration and the definition of shadow functions; at the time of the implementation of the MADE constraint system, the mC++ language did not support inline member functions definitions in class declarations. Ideally, the declaration of a constraint class (as an example the constraint class declaration of `DegreeToClock` from §4.4.1.3 is taken) would look like:

```
Constraint DegreeToClock {
  independent:
    void setTemp (int);

  dependent:
    int getClockValue ();

  relation:
    constraintFunction = { int value; value = ... };

  public:
    DegreeToClock (Degree* Clock);
    ~DegreeToClock ();

  private:
    Degree *degreeObject;
    Clock *clockObject;

};
```

In this case, the keyword 'Constraint' would be used like 'Active', 'Mutex' or 'Unprotected', to indicate the type of the object; the requirement to explicitly put the superclass CO in the header of the class declaration and to put an initiator at the header of the constructor could be done automatically by the compiler when the 'Constraint' classtype is recognized. The keywords 'independent' and 'dependent' could be used, just like 'private' and 'public', to identify different sections in which the independent and dependent shadow functions can be specified. The shadow functions would then be declared and defined at the same time. Finally, the 'relation' keyword would create a section where all the constraint relations, which can be supported by the constraint object, are declared and defined; the programmer does not have to worry about the signature of the constraint function; it will automatically be of type "void (\*)()". This improved syntax would probably not only make things easier for the programmer, it will also improve the readability of a constraint class declaration. In the same way, the definition of a constraint relation could be improved. Instead of all the 'register..' function calls, new keywords should be introduced in the language such that, for instance, the following piece of code would create a constraint relation (compare with the code in §4.4.3):

```
CONSTRAIN "int clockInstanceC::getClock()" AS "int getClockValue()"
  BY "void degreeInstanceC::setTemp(int)"
  USING "constraintFunction"
  OF "constraintObjectDCC"
```

With respect to the special constructs which can be used in the constraint functions (see also §4.4.1.2), an extension to the current set may be necessary. Using the current set it is possible to identify the triggering object and member function of the constraint object very well, but it does not provide possibilities to retrieve information about the context of the constraint object in the global constraint network. This information may also be important when priorities need to be implemented; based on the dependent and independent objects of the constraint object and of the priorities of these objects with respect to triggering and constraint satisfaction, different actions may be needed. In the current implementation the MADE constraint system does not support priorities. Moreover, no special constructs are available to support a constraint programmer in defining priorities. For the implementation of these priorities, the constraint function will need to have more information about the structure of the constraint network and thus about which objects are independent objects and which objects are dependent objects of the current constraint object. This information is, in principle, already available in the ROUTER object and it should take little effort to make this information available to the constraint function.

A point which has a large influence on the performance of the constraint system is the fact that a mC++ program is not compiled directly to executable code, but that it is translated to C++ first before a C++ compiler creates an executable. This means that programs which make use of the constraint system are faced with a double introduction of performance overhead. The first part of the overhead is caused by the translation step of a source code from mC++ to C++; mC++ member functions are replaced by a set of C++ functions and each mC++ member function invocation leads, in the C++ code, to the invocation of at least 3 C++ functions. The second part of the overhead is caused by the constraint system itself; constraint management, inevitable, introduces overhead due to the necessary administration of the different relations that need to be maintained. However, the overhead introduced by the constraint system in terms of mC++ actions is minimal; delegation makes sure that no special checks have to be made by the constraint system to check whether constraint relations are invalidated and the constrained objects do not have to provide special hooks or have to be prepared to be subject to constraint relations. Moreover, the use of delegation in the MADE constraint system realizes the most efficient way to trigger the constraint objects. First of all, a constraint object is only (renewed) triggered when the (renewed-)triggering member function of an (in)dependent object is invoked. This is also the only possibility to perform a (renewed-)triggering action of the constraint object; so no time is spent by any part of the constraint system to do some polling or checking of the (in)validity of the constraint relations when no (in)dependent (renewed-)triggering member function is invoked. Furthermore, delegation introduces (as far as the mC++ runtime is concerned) practically no overhead when compared to a normal mC++ member function call. Once the delegated call is executed, the constraint object is triggered automatically. This means that the delegatee (i.e. the constraint object) only has to call the original (renewed-)triggering member function and then perform the maintenance tasks of the constraint system.

With respect to the satisfaction of the constraint network, the MADE constraint system tries to be as efficient as possible. This means that only that part of a constraint network will be satisfied which contains constraint relations which really are invalidated. The constraint system also makes sure that every constraint relation is satisfied once during a traversal of the constraint network; the order in which constraint relations are satisfied is manipulated in such a way that this is possible. Constraint relations which are cyclic, naturally, may have to be satisfied more than once during a single traversal of the constraint network. However, the number of times the cycle is traversed is reduced as much as possible. Because the contents of the constraint functions determine, to a large extent, how often a cyclic needs to be traversed, the constraint system cannot totally control this aspect. Therefore, the total complexity of the satisfaction process of the constraint network is partially determined by the complexity of the constraint functions (and thus by the constraint programmer) and, as a result of that, the MADE constraint system has no means to control the overall complexity. However, this may be desirable in cases where constraint relations are used to control the



complexity of an application (as is the case in the Quality Factory). In those situations special functions should be available either to let the constraint programmer indicate the complexity to the constraint system or to let the constraint system analyze the constraint function to find out the complexity of it itself. This area, obviously, need some more research.

Compared to other constraint systems, the MADE constraint system is very flexible and does not require the objects that are to be constrained to provide any special hooks. In this sense, the MADE constraint system is really a general system (both in terms of constraints it can support and in terms of the objects that it can constrain).

Due to the inefficiencies, primarily introduced by the MADE runtime code, the performance of the constraint system in actual life cannot really be compared with other constraint systems which are an integral part of a programming language. Performance was not the primary goal of this thesis (flexibility, elegance and maintainability were). Therefore, this thesis did not contain any time-measurements on the performance of the MADE constraint system; constraint systems which are an integral part of a programming language will clearly outperform the MADE constraint system. As a result of the reduced performance of the MADE constraint system, it will probably not be possible to model time-critical constraint relations; the overhead introduced by the MADE constraint system might be too high to guarantee the correct maintenance of any time-relations specified by means of constraint relations. In theory however it is possible to define such time-critical relations in the MADE constraint system. The need for these kinds of relations in multimedia environments is obvious; synchronization actions and scheduling operations clearly make use of time-critical relations to define aspects like duration, latency, speed etc. Beside the fact that a constraint system, integrated in a programming language, would provide a better readability of the constraint relation specifications, the need for time-critical constraint relations is another (important) point to consider the incorporation of the MADE constraint system in the mC++ language. Note that this incorporation of the constraint system into the language is only one of the requirements for the constraint system to be able to support time-critical constraint relations; a list of requirements would, however, be beyond the scope of this thesis. Nevertheless, it provides one of the possible directions in which the MADE constraint system can be developed. In this context, delegation as a way of triggering the constraint system is a very promising approach because its overhead (when implemented efficiently) is very small and it reduces the overhead needed by the constraint system to detect any triggering actions. Nevertheless, it remains an open issue if the overhead of constraint management in general ever becomes small enough to be usable for time-critical relations.

### 6.2 Discussions on the Quality Factory

For the Quality Factory a prototype implementation is written. This prototype implements the functionality as described in Chapter 5. Characteristics of the Quality Factory have clearly been inherited from the MADE constraint system which is used as its foundation: the transparency of the Quality Factory with respect to the (re)use of existing filters, the flexibility with which relations between arguments of the filters can be (dis)regarded and the dynamics with which new filters can be added to and old filters can be removed from the Quality Factory. With respect to this last aspect, the dynamic addition and removal of filters, the prototype is not yet complete. When a filter is added (removed), constraint relations have to be added (removed). It is only possible to do this manually, since this cannot be done automatically by the prototype.

With respect to the requirements put on the Quality Factory (as described in Chapter 2) it can be said that most of them are realized. The prototype of the Quality Factory shows that the MADE constraint system makes it relatively easy to specify the relations between the (arguments of the) different filters, that these relations are transparent to the viewer and the filters, that they can

accomplish the desired translation from the 'language' of the viewer to the 'language' of the filters and that they can describe different situations. However, using a constraint system as a foundation to build the Quality Factory on inevitably introduces overhead; beside the advantages of the MADE constraint system, the Quality Factory also has inherited some of its disadvantages. As the MADE constraint system had, in practice, some problems to maintain time-critical relations, the Quality Factory introduces another level of overhead and will not be suited to support perceptual quality requirements with respect to time. This implies that requests with respect to duration, response time, delay, delay variation etc. cannot be supported by the Quality Factory. However, these kinds of metrics can be translated into terms of resource capacity (compare the QoS systems in network communication in §2.1) and thus, theoretically, they can be modelled in the Quality Factory. A general concept for translating the time-critical request to the technical parameters for the various filters is one of the possible research topics which can be looked at in the future when the prototype implementation of the Quality Factory is developed towards a real implementation. This research may then indicate that time-related issues should be dealt with in a special way.

Another point with respect to the performance of the Quality Factory, which shows a flaw of the current mC++ implementation, has to do with the creation of the constraint network. The MADE constraint system creates a number of active CGO objects. Each of these CGO objects manages a constraint network which contains objects which have no connections with objects in the constraint networks of other CGO objects. When such a link is created after all, the two CGO objects are merged. A problem with this approach arises on the SGI platform (which has only 40 threads available for 40 active objects) when the constraint network of the Quality Factory is constructed; a lot of constraint networks may be created which have no connections with each other during the first phase of the construction. And although only a few constraint networks will finally result when all argument relations are processed and several CGO objects are merged, the number of temporal CGO objects during the construction of the constraint network may exceed the maximum of 40. In general it should also be true that the number of argument relations that are specified determines the time it takes to create the constraint network. To a certain extent this is obvious; the more argument relations are specified, the more constraint relations need to be specified. However, due to the frequent creating and merging of CGO objects the amount of time needed to create the constraint network in the Quality Factory may increase dramatically. To overcome these kinds of problems with the creation of the constraint network, maybe some kind of preprocessing of the argument relations is needed which orders them in such a way that the number of merges is minimized.

With respect to the overhead introduced by the Quality Factory itself a few remarks can be made. Because the MADE constraint system is used as the foundation, the overhead to maintain the argument relations equals the overhead to satisfy the different constraint relations needed to describe these argument relations. As the constraint functions which implement these constraint relations have a simple complexity themselves, the total complexity of the satisfaction process (from the viewpoint of the mC++ programming language) does not have to be that high. However, when a small system is created (such as the example in Chapter 5), already a large set of argument objects are created. Due to this large amount of argument objects, a large number of constraint relations is created. And although not all the constraint relations will be considered at the same time (not all contexts are entered at the same time), still a large amount of constraint relations remain active. The problem that remains is that this large amount of argument objects is needed when the flexibility of the Quality Factory has to be realized. Nevertheless, the number of constraint objects in the constraint network can be reduced. Especially when no explicit manipulation is specified on the dataflow, an argument relation does not require the presence of a constraint relation (consider, for instance, the argument relations which specify that the value of an argument equals the value of one other argument). Also in other parts of the constraint network, created by the Quality Factory, constraint objects can be merged (a set of constraint objects can be replaced by one constraint object where the (in)dependent objects of the original set become (in)dependent objects of the new



constraint object) or even removed. However, some care has to be taken when constraint objects are merged/removed; the presence of different contexts makes this process somewhat more complicated. Nevertheless, a substantial gain in performance can be achieved here and a next version of the Quality Factory should certainly be improved on this point.

Also the influence of the Quality Factory on the negotiation process is minimal in the current prototype implementation. As a result of this, the Quality Factory cannot influence the outcome of the negotiation process. Some research may be done in this area to look how the interaction between Quality Factory and negotiation objects can be improved.

Comparing the Quality Factory with other systems, it can be concluded that the functionality of the Quality Factory is rather unique; there is not really another system which provides the same functionality. Most other Quality-of-Service systems provide only a fixed number of limited quality levels. Due to this restriction, the processes to perform negotiation and to enforce the required quality level become much simpler and thus easier to implement efficiently. In this sense, the Quality Factory obviously performs worse than these 'discrete' quality systems, but it provides more flexibility with respect to the supported quality levels. The fact that some of the quality levels are, at the moment, technically not sensible or realistic may soon be falsified considering the great pace with which hardware technology is developing. Systems like CORBA and COM also cannot be compared with the Quality Factory as they provide a totally different functionality which supports the distribution of applications over several computers. Also a system like OLE does not compare with the Quality Factory; OLE only exchanges data from different applications using a standard format while it cannot support interoperability between software components.

### 6.3 Conclusions

Although a number of elements in the MADE constraint system and the Quality Factory have been pointed out that can be improved, both systems prove to be valuable. When time-critical relations are not considered, the current implementations show that the translation from perceptual metrics to technical metrics can be done very well in the way that is described in this thesis. The approach taken here provides a very flexible system. Furthermore, time-critical relations may not be supported in practice, they can be specified in the Quality Factory. In this sense, the Quality Factory provides also a good modelling environment (even for time-critical relations).

Furthermore, the Quality Factory proves to be a system which cannot only be used for perceived Quality of Services systems, but also in a much broader context; it is a system which provides an environment in which the integration of really different software components can actually be realized while, at the same moment, this integration is formalized such that the use of the Quality Factory is elegant for the end-user and its maintainability is elegant for the programmer. Problems like the introduction of the new currency in Europe (the 'euro') or the turn of the century (dates changing from 1999 to 1900 instead of 2000) might not have been so huge when environments like the Quality Factory would have been used and filters could be plugged in transparently to convert currencies and dates to and from the old and new system.

As a final conclusion it should be noted that the MADE constraint system shows that complex problems such as perceived quality requirements and negotiation can be solved in an elegant and rather straightforward way by using constraint relations and that the Quality Factory shows that the MADE constraint system provides a powerful constraint system which is capable of supporting a diverse number of constraint problems.

# ANNEX



## A Description of Provided and Requested Services

In this annex, the different filters and services as needed in the demonstration program of Chapter 5 are presented. These functions will return in the remainder of the annex in different forms. Therefore, this section forms the basis for understanding the following sections.

The demonstration program used in Chapter 5 provides a number of services which can be requested by the user:

### FILE **\*initializeFLI** (char \*file)

This service opens the file named *file*. The service returns a pointer to the opened file structure.

### void **showNextFLIFrame** (MyWin \*win, FILE \*fd, void \*buffer)

This service shows the next frame from the 'FLI' file pointed to by *fd*. This image is stored in *buffer* and also displayed in an X-windows environment in window *win* on the screen. The pointer *buffer* should point to a memory area which is large enough to contain the data for one frame.

### void **terminateFLI** (FILE \*fd)

This service closes the file pointed to by *fd*.

The demonstration program also needs a number of filters. First of all, there are three filters which provide the actual functionality of the three services (in practice, the services are created depending on the filters available; it should be realized that the services mentioned above are derived from these three filters):

#### ◦ FILE **\*initFLI** (char \*file, int \*frames, int \*width, int \*height)

This filter opens the file named *file* and reads some control information from the header of the 'FLI' file. This information can be returned in the parameters *frames* (contains the number of frames in the 'FLI' movie), *width* (the width of each individual frame in number of pixels) and *height* (contains the height of each individual frame in number of pixels). If NULL pointers are supplied for these last three parameters, no information is returned for these attributes. The filter returns a pointer to the opened file structure.

#### ◦ void **readFLIFrame** (FILE \*fd, void \*buffer, int width, int height)

This filter reads the contents of the next frame of the file pointed to by *fd* into the memory block pointed to by *buffer*.

#### ◦ void **closeFLI** (FILE \*fd)

This filter closes the file pointed to by *fd*.

There are eleven other filters. These can be divided into two filters for negotiation, one filter that is able to transform data from one format into another one, two filters which are aware of the underlying network and six filters that work on data in a specific format:

#### ◦ void **negotiateTurningSpeed** (int rounds, int claim, int firmness)

This filter will calculate the effects of the increase (decrease) in rounds per minute the globe has to spin on the bandwidth needed for the network connection. The new requirement with respect to the turning speed is expressed in *rounds*. The parameter *claim* describes what kind of requirement this is and with which *firmness* this requirement has to be issued. As the throughput of a network connection has a maximum limit above

which it cannot go, specifying a new turning speed may also effect the detail at which the globe is displayed.

◦ void **negotiateDetailLevel** (int detail, int claim, int firmness)

This filter will calculate the effects of the new level of detail which is displayed on the bandwidth needed for the network connection. The new requirement with respect to the level of detail is expressed in *detail*. The parameter *claim* describes what kind of requirement this is and with which *firmness* this requirement has to be issued. As the throughput of a network connection has a maximum limit above which it cannot go, specifying a new level of detail may also effect the turning speed of the globe.

◦ void **\*fromBM2XBM** (void \*buffer, int width, int height)

This filter can convert an image (size *widthxheight* pixels), stored in a memory block pointed at by *buffer* in a normal bitmap format (i.e. each pixel in the image is represented by one byte of information) into an image in XBM format (i.e. each eight horizontal successive pixels are stored in one byte). A pointer to the memory block in which this image in XBM format is stored is returned by the filter. Notice that a normal bitmap may use 256 different colors for one pixel whereas 'XBM' images are in black and white.

◦ void **sendData** (int socket, char \*host, void \*buffer, int size)

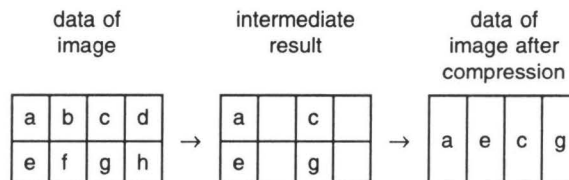
This filter sends data, *size* bytes stored in a memory block pointed to by *buffer*, to the socket identified by *socket* of the computer named *host*.

◦ void **\*receiveData** ()

This filter is capable of reading the data sent to a specific socket and store this information somewhere in memory. If no data was sent, a NULL pointer is returned by this filter. Otherwise a pointer is returned which points to the memory block in which the received data is stored.

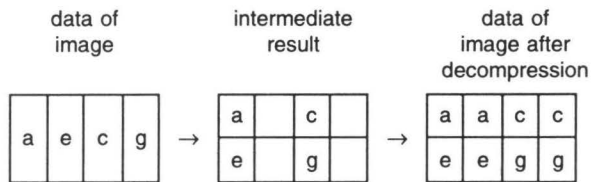
◦ void **\*compressXBM2** (void \*buffer, int width, int height)

This filter will compress the data of an XBM image (size *widthxheight*, pointed to by *buffer*) by a factor two. However, this also means that the level of detail is reduced by a factor two. Compression is done by leaving out every second pixel in a horizontal scan line. Then the holes in the data are compressed by leaving out every second pixel and merging every second scanline with the first scanline. The data which is obtained in this way is stored in a memory block and a pointer to this block is returned by the filter.



◦ void **\*decompressXBM2** (void \*buffer, int width, int height)

This filter will decompress the data, obtained from the filter *compressXBM2*. The parameter *buffer* contains a pointer to this memory block. From each scanline in the compressed image data, every first and second scanline are restored and then within each restored scanline, every second pixel is restored as a copy of the first pixel:



◦ void **\*compressXBM4** (void \*buffer, int width, int height)

The filter compressXBM4 does the same as filter *compressXBM2* except that it will compress the data by a factor 4; the information of every second pixel of every scanline is removed and every second scanline is removed. The transitions are similar to those of *compressXBM2*.

◦ void **\*decompressXBM4** (void \*buffer, int width, int height)

This filter will undo the compression of filter *compressXBM4* analogous to the working of *decompressXBM2*.

◦ void **\*clipBM** (void \*buffer, int width, int height, int x, int y, int dx, int dy)

This filter extracts from a bitmap image, size *widthxheight*, pointed to by *buffer*, a second bitmap image which has size *dxxy* and which is situated at location *(x,y)* in the original image. A pointer to the memory block in which this new image is stored is returned by the filter.

◦ void **showXBM** (MyWin \*win, void \*buffer, int width, int height, int xoffset, int yoffset)

This filter can display the 'XBM' image, size *widthxheight* and pointed to by *buffer*, on the screen identified by *win* at location *(xoffset, yoffset)* in a X-windows environment.



## B Specification of the Argument Relations

The services and filters from Annex A are used in this section to show an example of the specification of argument relations. Below, a network of filters is created where data (in this case a movie in 'FLI'-format) can be read from a file, send over a network connection and displayed on a screen. The data can be compressed before it is send over the network connection if necessary; it may be compressed by a factor 2 (when *compressXBM2* is used) or by a factor 4 (when *compressXBM4* is used).

```

SERVICEREQ FILE *initializeFLI (char* file);
SERVICEREQ void showNextFLIFrame (MyWin *win, FILE *fd, void *buffer);
SERVICEREQ void terminateFLI (FILE* fd);

5  SERVICEPRV FILE *initFLI (char *file, int *frames =0, int *width =0, int *height =0);
SERVICEPRV void readFLIFrame (FILE *fd, void *buffer, int width =320, int height =200);
SERVICEPRV void closeFLI (FILE *fd);

SERVICEPRV void *clipBM (void *buffer, int width, int height, int x, int y, int dx, int dy);
10 SERVICEPRV void *fromBM2XBM (void *buffer, int width, int height);
SERVICEPRV void *compressXBM2 (void *buffer, int width, int height);
SERVICEPRV void *compressXBM4 (void *buffer, int width, int height);
SERVICEPRV void *decompressXBM2 (void *buffer, int width, int height);
SERVICEPRV void *decompressXBM4 (void *buffer, int width, int height);
15
SERVICEPRV void sendData (int socket = 80, char *address = "bosgors", void *buffer, int size);
SERVICEPRV void *receiveData ();

SERVICEPRV void showXBM (MyWin *win, void *buffer, int width, int height, int xoffset, int yoffset);
20
SERVICEPRV void releaseMemory (void *ptr);

CONTEXT -:
initFLI::file<-initializeFLI::file
25 initializeFLI::fd<-initFLI::fd
closeFLI::fd<-terminateFLI::fd
terminateFLI::fd<-closeFLI::fd

30 readFLIFrame::fd<-showNextFLIFrame::fd
readFLIFrame::buffer<-showNextFLIFrame::buffer

clipBM::buffer<-readFLIFrame::buffer
clipBM::buffer<-#readFLIFrame::buffer
35 clipBM::width<-readFLIFrame::width
clipBM::height<-readFLIFrame::height
clipBM::x<-128
clipBM::y<-74
clipBM::dx<-64
40 clipBM::dy<-64

fromBM2XBM::buffer<-clipBM::buffer
fromBM2XBM::width<-clipBM::dx
fromBM2XBM::height<-clipBM::dy
45
showXBM::win<-showNextFLIFrame::win
showXBM::width<-fromBM2XBM::width
showXBM::height<-fromBM2XBM::height
showXBM::xoffset<-300%2-fromBM2XBM::width%2
50 showXBM::yoffset<-300%2-fromBM2XBM::height%2

showNextFLIFrame::fd<-#showXBM::fd
releaseMemory::ptr<-showXBM::buffer
using del2;

```



```

55 CONTEXT compress0:
    sendData::buffer<-fromBM2XBM::
    sendData::size<-clipBM::dx%8*clipBM::dy
    releaseMemory::ptr<-fromBM2XBM::buffer                using del1;

60 showXBM::buffer<-receiveData::

CONTEXT compress2:
    compressXBM2::buffer<-fromBM2XBM::
    compressXBM2::width<-fromBM2XBM::width
65 compressXBM2::height<-fromBM2XBM::height
    releaseMemory::ptr<-fromBM2XBM::buffer                using del1;
    sendData::buffer<-compressXBM2::
    sendData::size<-clipBM::dx%8*clipBM::dy%2
    releaseMemory::ptr<-compressXBM2::buffer                using del3;

70 decompressXBM2::buffer<-receiveData::
    decompressXBM2::width<-compressXBM2::width
    decompressXBM2::height<-compressXBM2::height
    showXBM::buffer<-decompressXBM2::
75 releaseMemory::ptr<-decompressXBM2::buffer                using del4;

CONTEXT compress4:
    compressXBM4::buffer<-fromBM2XBM::
    compressXBM4::width<-fromBM2XBM::width
80 compressXBM4::height<-fromBM2XBM::height
    releaseMemory::ptr<-fromBM2XBM::buffer                using del1;
    sendData::buffer<-compressXBM4::
    sendData::size<-clipBM::dx%8*clipBM::dy%4
    releaseMemory::ptr<-compressXBM4::buffer                using del5;

85 decompressXBM4::buffer<-receiveData::
    decompressXBM4::width<-compressXBM4::width
    decompressXBM4::height<-compressXBM4::height
    showXBM::buffer<-decompressXBM4::
90 releaseMemory::ptr<-decompressXBM4::buffer                using del6;

CONTEXT del1:
    releaseMemory::ptr<#fromBM2XBM::

95 CONTEXT del2:
    releaseMemory::ptr<#showXBM::

CONTEXT del3:
    releaseMemory::ptr<#compressXBM2::

100 CONTEXT del4:
    releaseMemory::ptr<#decompressXBM2::

CONTEXT del5:
105 releaseMemory::ptr<#compressXBM4::

CONTEXT del6:
    releaseMemory::ptr<#decompressXBM4::

```

## C The Negotiation Objects

In Chapter 5 so-called negotiation objects were introduced. The interface of these objects (the 'NegotiationObject', the base class for all the negotiation objects) looks as follows:

```

Mutex NegotiationObject : public CO {
public:

    NegotiationObject ();
    ~NegotiationObject ();

    void addTradeOff (MObject* negObj, int m, int n);
    void addContext (int low, int high, char* context);

    void cfBound ();                // constraint function to manage a bound claim
    void cfBound_Lck ();            // constraint function to manage a bound&lock claim
    void cfCrca ();                // constraint function to manage a circa claim
    void cfCrca_Lck ();            // constraint function to manage a circa&lock claim
    void cfAll ();                 // constraint function to manage a all claim
    void cfAll_Lck ();             // constraint function to manage a all&lock claim

    void unlock();                 // function to unlock the negotiation object after a ...&lock claim

    declareIndepShadowFunc (void setValue, (int));

protected:

    MObject* negotiable;           // the characteristic that is negotiated over by this negotiation object

    int currentClaim;
    int currentClaimFirmness;
    int currentBoundary;
    int currentBoundaryFirmness;

    int lastTradeOff;              // auxiliary variable to help with negotiation process

    CnbtLst Cnbt [];               // list with information about which context to enter for which claim
    TradeOffLst TradeOff [];       // list with information about the trade-offs
    ...

};

...

void NegotiationObject::cfBound () {
    int realizedClaim = negotiable->getValue ();
    case (currentClaimFirmness) {
        NonNegotiable : {                // NonNegotiable Claim:
            if (realizedClaim != currentClaim) { ... };
        };
        Negotiable: {                    // Negotiable Claim:
            if (realizedClaim != currentClaim) { ... };
        };
        BestEffort: {                    // BestEffort Claim:
            // do nothing
        };
    };
};

...

defineIndepShadowFunc (void, NegotiationObject, setValue, (int i), (i));

```

Using the base class 'NegotiationObject' new, specialized negotiation objects can be created. To set up a negotiation filter (for instance, 'negotiateDetailLevel'), a new objects is created which inherits from the base class 'NegotiationObject'. The following code is an example of the specification of a negotiation object (it is assumed that only *bounded* resource claims will be made for the detail-level where the boundaries are equal to the claim itself):

```
Mutex DetailNO : NegotiationObject {
    public:
        DetailNO (MObject* object);

5    void negotiateDetailLevel (int detail, int claim, int firmness);
};

DetailNO::DetailNO (MObject object) {
    negotiable = object;
10 ...
};

void DetailNO::negotiateDetailLevel (int detail, int claim, int firmness) {
    currentClaim = claim;
15    currentClaimFirmness = firmness;
    currentBoundary = claim;
    currentBoundaryFirmness = NonNegotiable;
    lastTradeOff = 0;
    negotiable->setValue (detail);
20 };

MObject *TurningSpeedObj, *DetailObj, *sizeMemObj, *sizeBandwidthObj;

DetailNO* no = new DetailNO (DetailObj);
25 registerIndependentObject (DetailObj, DetailNO, "void setValue(int)"); // register the constraint relations
    registerDependentObject (DetailObj, DetailNO);

    no->addTradeOff (sizeMemObj, 1, 1) // enter the different trade-offs with DetailObj
    registerIndependentObject (sizeMemObj, DetailNO, "void setValue(int)");
30 registerDependentObject (sizeMemObj, DetailNO);

    no->addTradeOff (sizeBandwidthObj, 1, 1)
    registerIndependentObject (sizeBandwidthObj, DetailNO, "void setValue(int)");
    registerDependentObject (sizeBandwidthObj, DetailNO);
35


    no->addTradeOff (TurningSpeedObj, 1, 1)
    registerIndependentObject (TurningSpeedObj, DetailNO, "void setValue(int)");
    registerDependentObject (TurningSpeedObj, DetailNO);

40 no->addContext ( 0, 25, "compress4"); // enter the different ranges for context activation
    no->addContext ( 26, 50, "compress2");
    no->addContext ( 51, 100, "compress0");

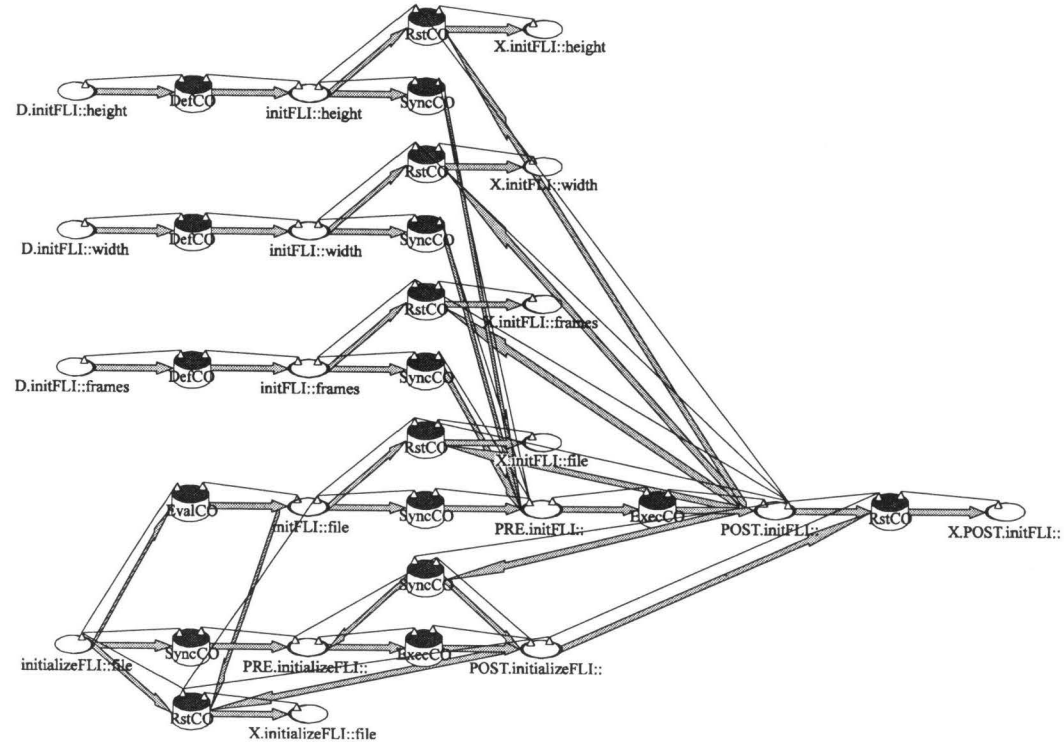
    registerConstraintObject (DetailNO, "void cfBound()");
```

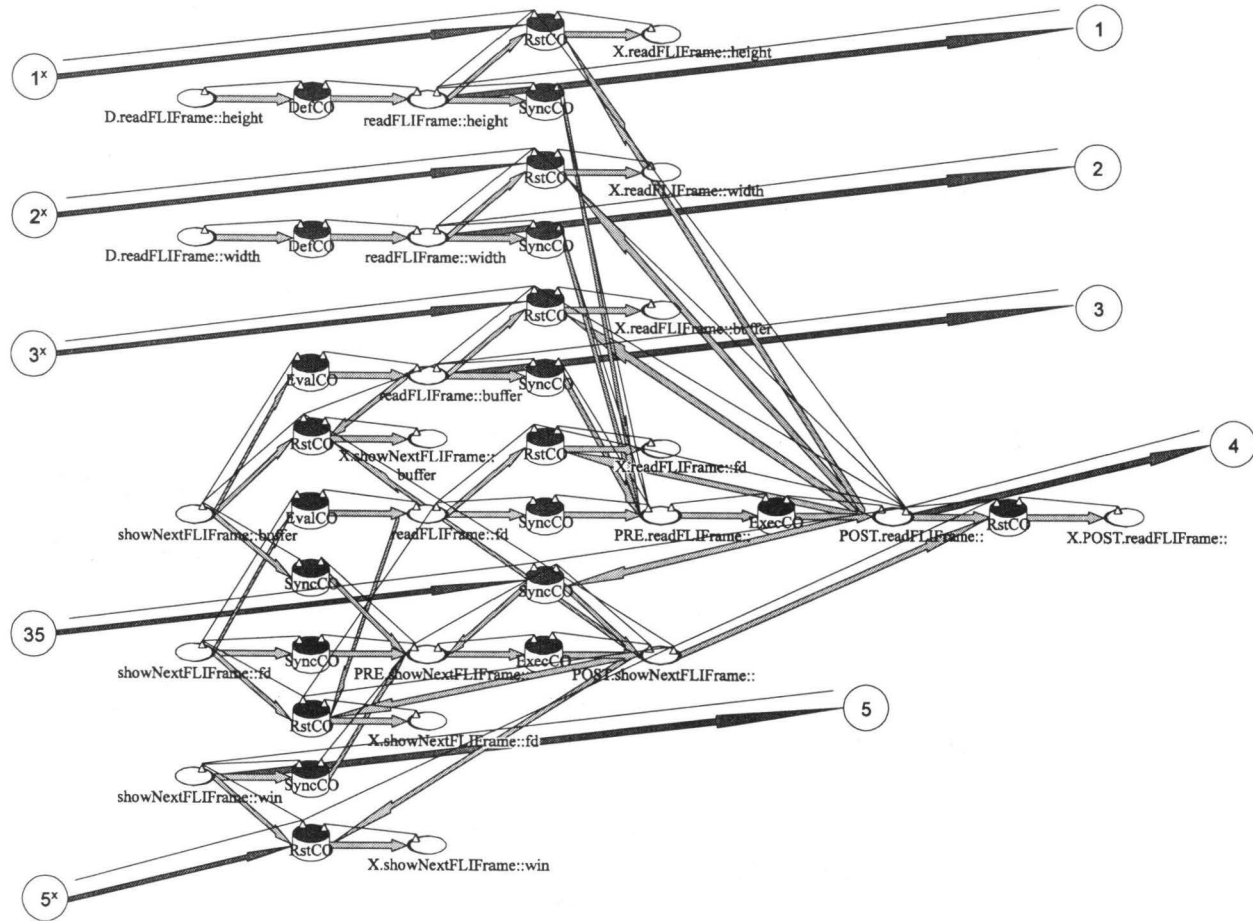
This code makes sure that the allocation of resources for detail-level, turning-speed, memory and bandwidth are all coupled together (trade-offs are defined between the different resources; lines 28-38) and that the adaptation of the detail-level will trigger the negotiation object (line 25). Furthermore, in lines 40-42, it is specified which context should be entered for the various values of the detail-level.

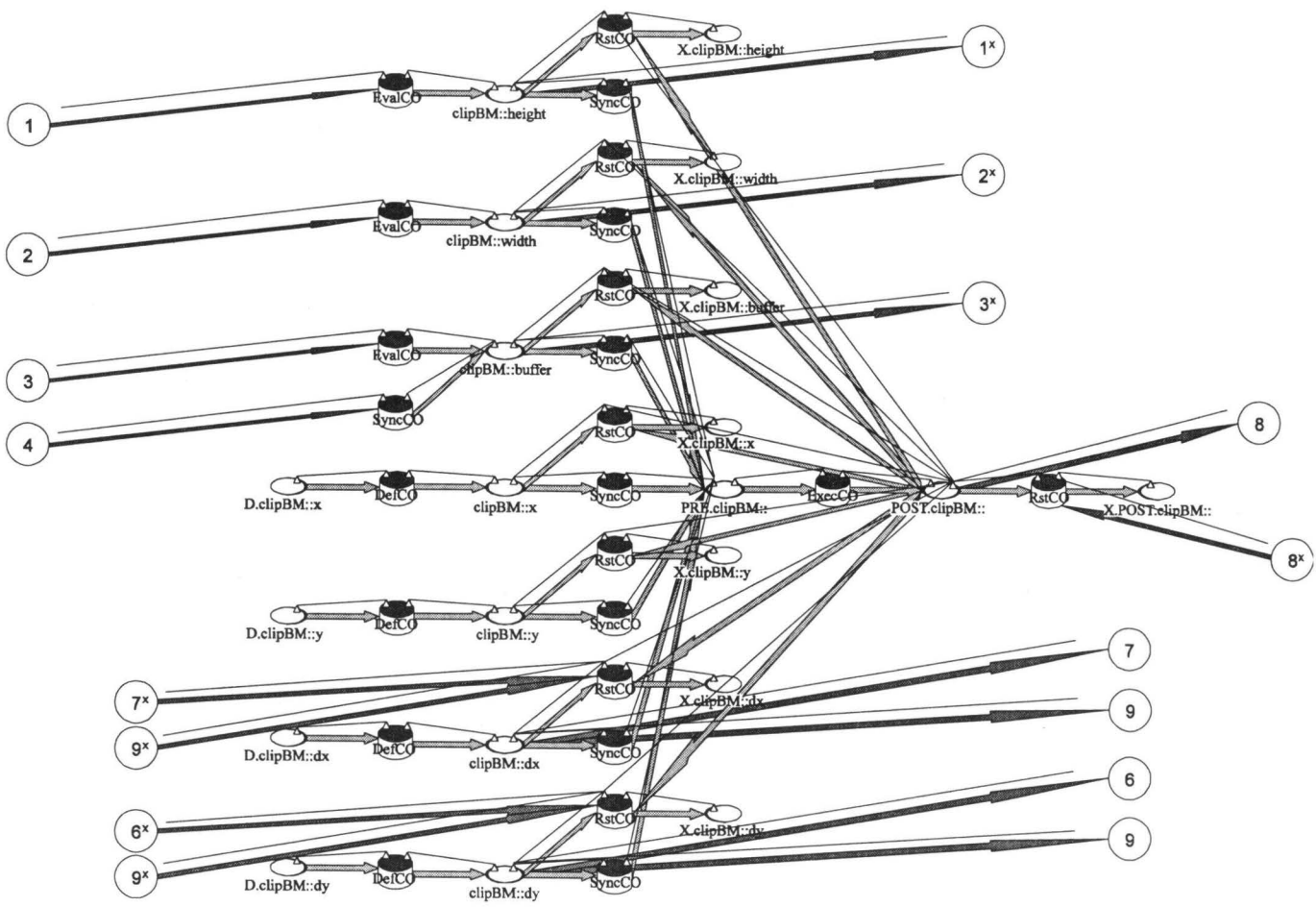
## D Overview of the Constraint Networks

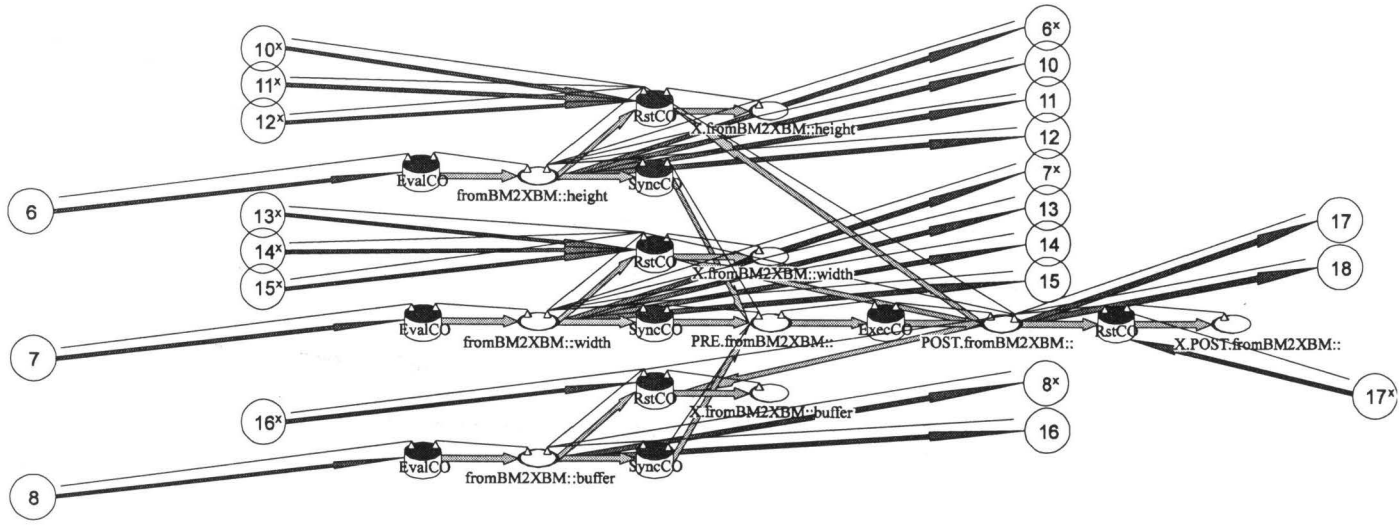
In this annex several diagrams are presented, each showing part of the constraint network for the example services and filters which were introduced in Chapter 5. It is assumed that the filters *initFLI*, *readFLIFrame*, *closeFLI*, *clipBM*, *fromBM2XBM*, *compressXBM2*, *compressXBM4* and *sendData* are running on one computer and *receiveData*, *decompressXBM2*, *decompressXBM4* and *showXBM* are running on another computer. On each of the following pages, the constraint networks for the separate filters are shown. If a dependency relation exists which is defined between the filter described on a particular page and another filter described on another page, then this is represented by an arrow pointing to a symbol similar to: . In the diagram of the other filter the same symbol appears and an arrow will indicate where the dependency relation should actually be connected.

Note that the constraint relations described for the filters *compressXBM2* and *decompressXBM2* are only active when context "compress2" is entered. The same holds for the filters *compressXBM4* and *decompressXBM4* with respect to context "compress4". In this respect it is important to remember that dependencies on one page do not necessarily all have to be active as these dependencies may cross page boundaries and can be connected to filters which are active for different contexts.

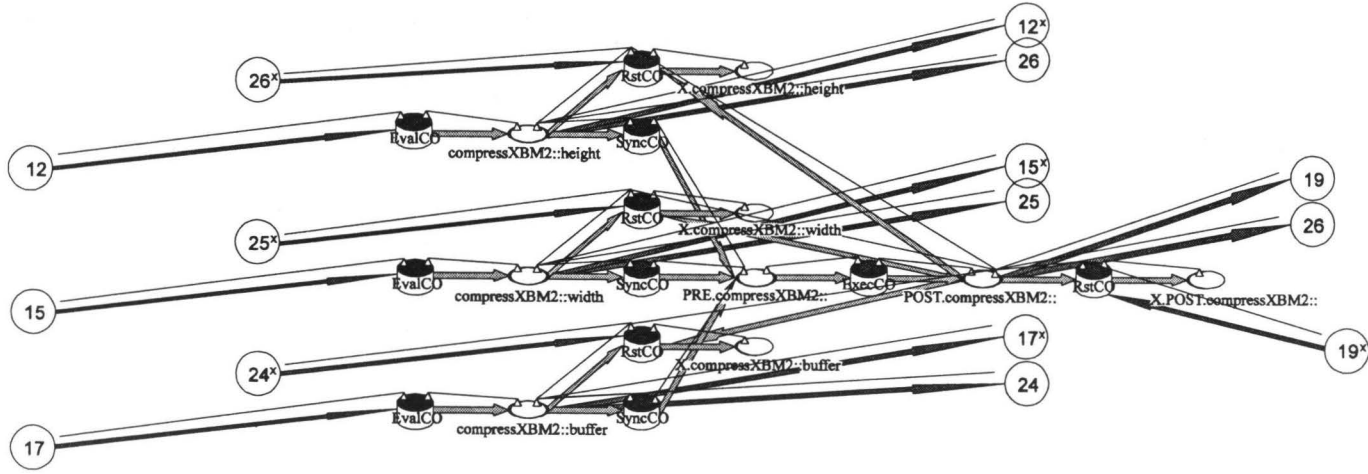


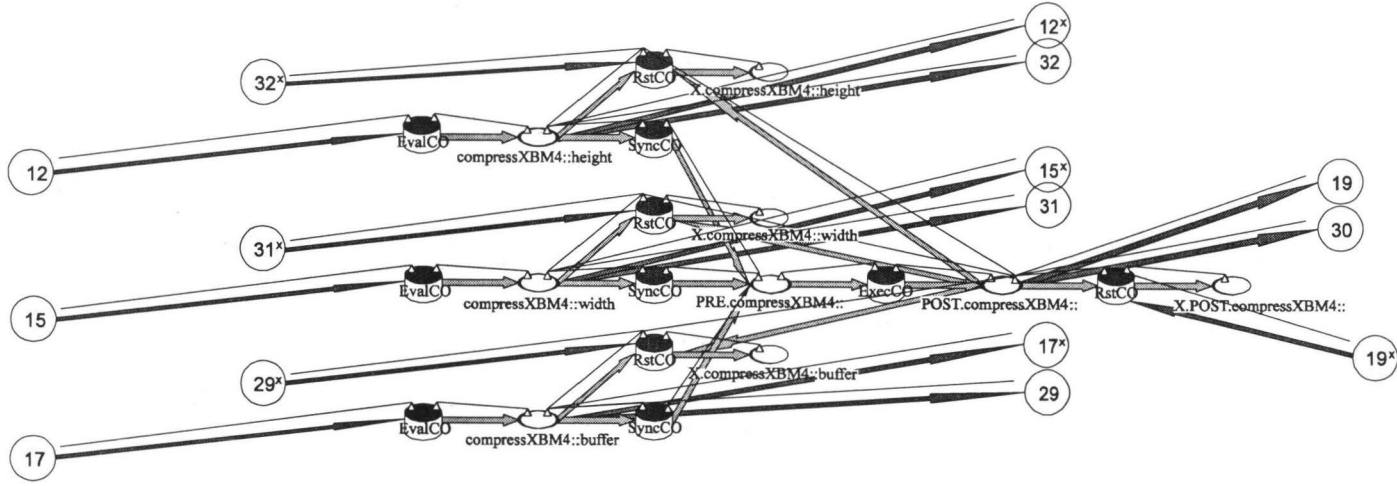


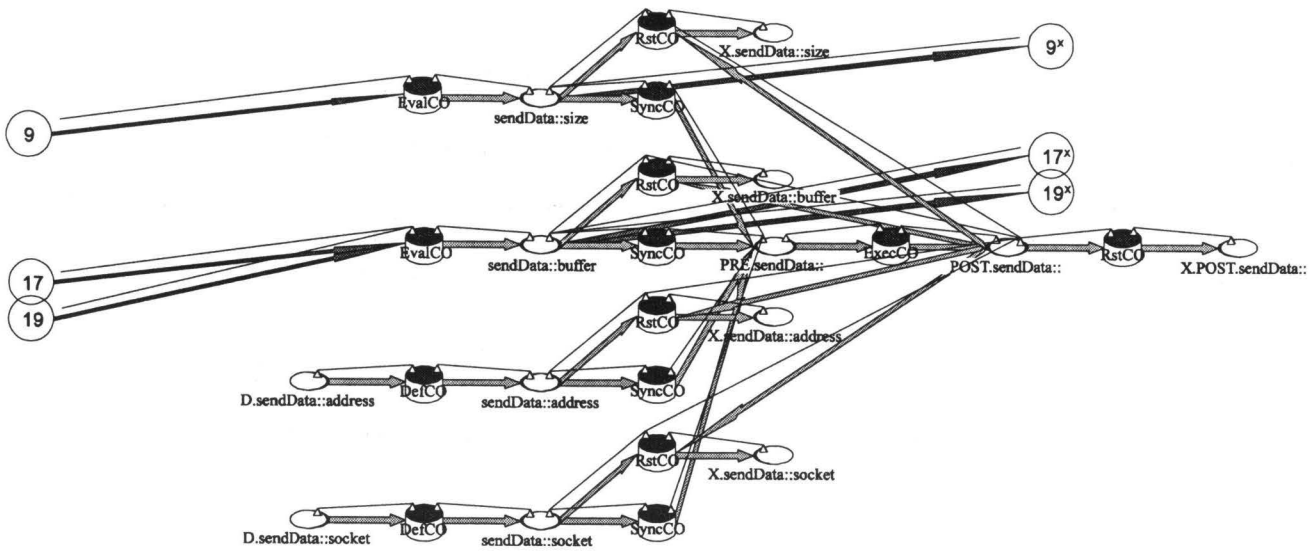


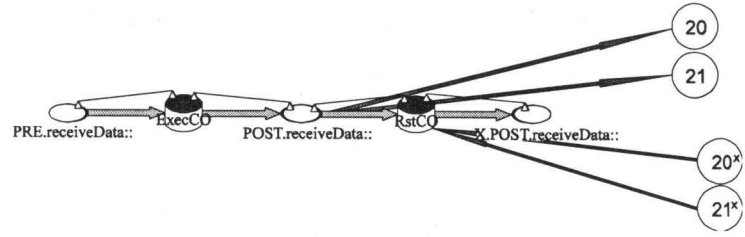


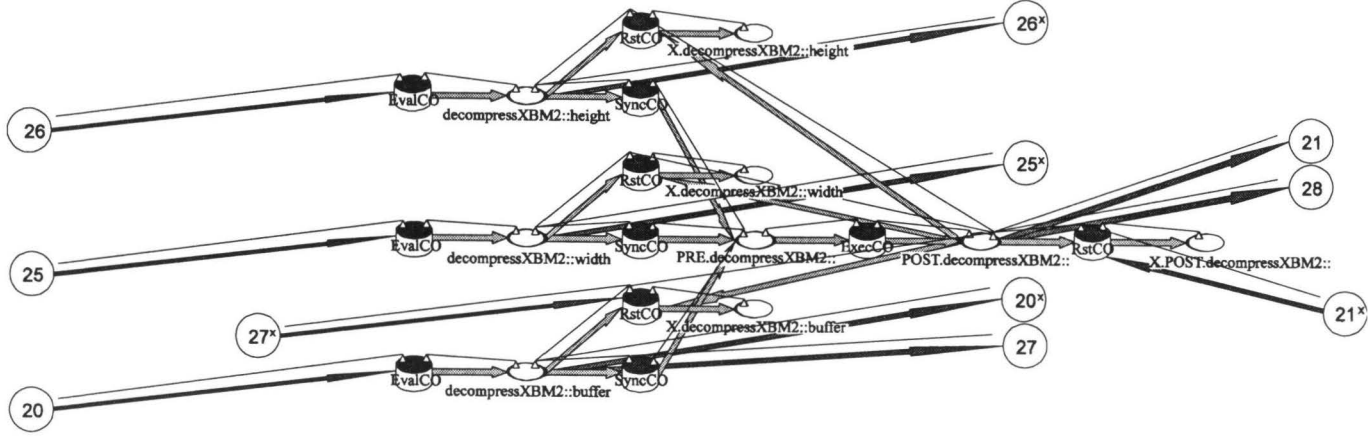


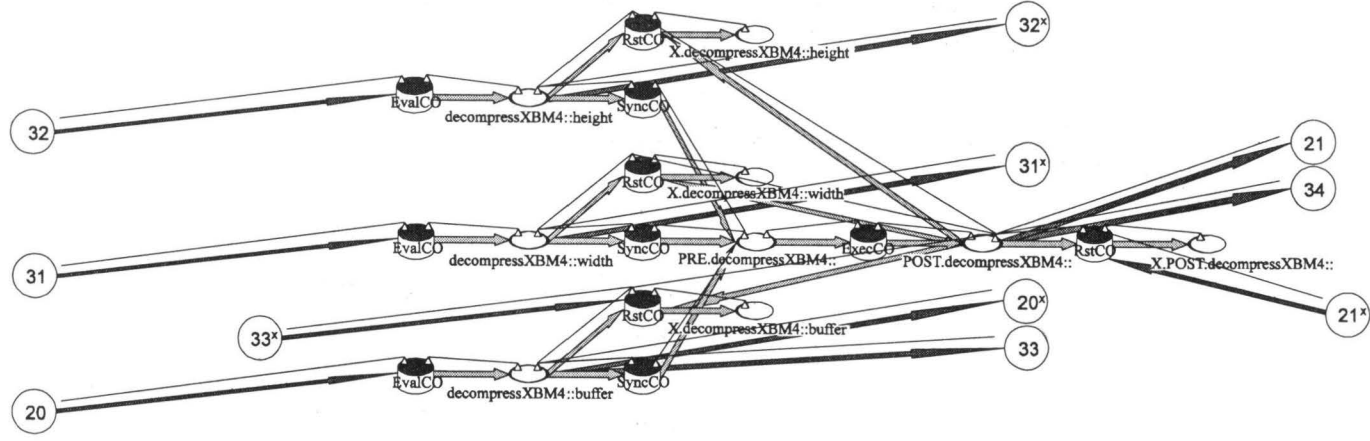


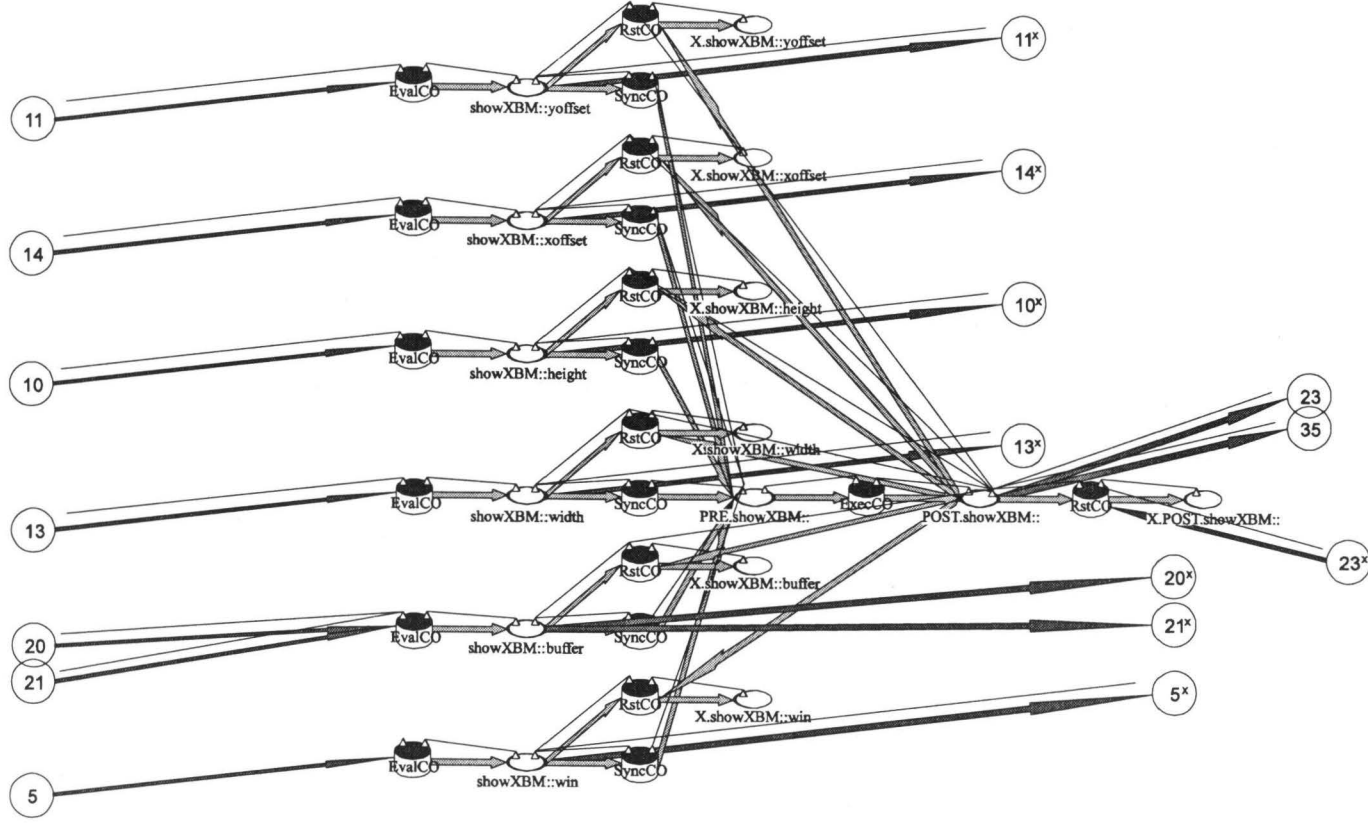


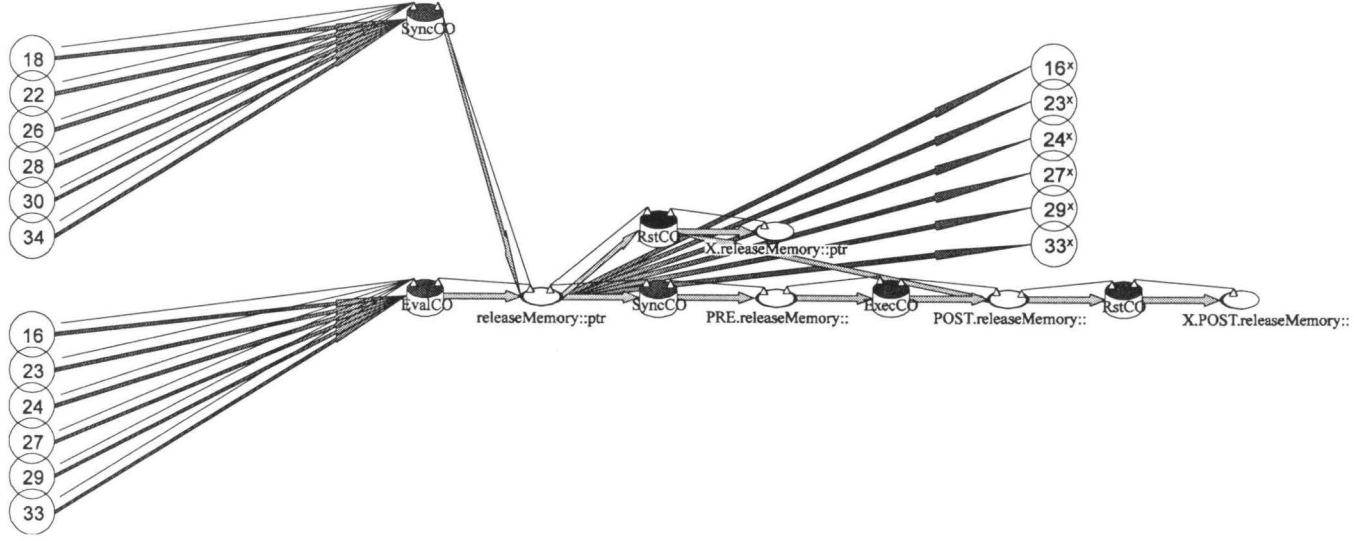




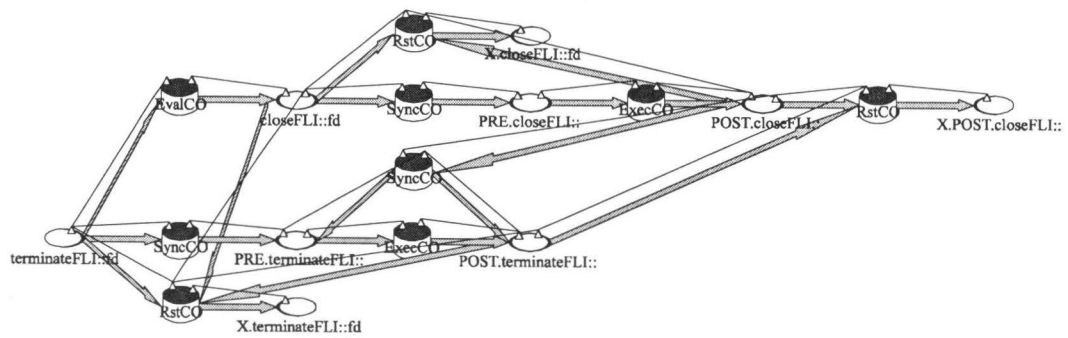












# REFERENCES



- 
- [Amenyo et al. 93] Amenyo J.T., Lazar A.A., Pacifi G., Proactive Cooperative Scheduling and Buffer Management for Multimedia Networks, in *Multimedia Systems*, **1:1**, pp. 37-49, Springer Verlag, 1993.
- [Arbab et al. 93a] Arbab F., Hagen P.J.W. ten, Haindl M., Heeman F.C., Herman I., Reynolds G.J., Siebes A., *Specifications of the MADE Object Model*, tech.rep. T/OM, CWI, 1995.
- [Arbab et al. 93b] Arbab F., Herman I., Reynolds G.J., An Object Model for Multimedia Programming, proceedings EuroGraphics'93, in *Computer Graphics Forum*, **12:3**, pp. 101-113, The Eurographics Association, 1993.
- [Atkinson et al. 95] Atkinson S., Blair G.S., Correia N., Duce D.A., Duke D.J., Hagen P.J.W. ten, Herman I., Hintum J.E.A. van, Paterno F., Palangue P., Reynolds G.J., Scholefield D., Stenzel H., *PREMO Types and Objects for Time Management*, ERCIM, 1995.
- [Bergmans 94] Bergmans L., *Composing Concurrent Objects*, Ph.D. thesis, University of Twente, 1994.
- [Bergmans et al. 96] Bergmans L., Akşit M., Composing Synchronization and Real-Time Constraints, *Journal of Parallel and Distributed Computing*, **36**, pp. 32-52, Academic Press, 1996.
- [Bergstra et al. 94] Bergstra J.A., Klint P., *The TOOLBUS - a Component Interconnection Architecture*, tech.rep. P9408, University of Amsterdam, 1994.
- [Berre 92] Berre A.J., Experiences from Systems Integration through an Object-Oriented Software Bus, proceedings TOOLS EUROPE'92, in *TOOLS 7*, ISBN 0-13-917436-2, pp. 33-45, Prentice Hall, 1992.
- [Biezunski 92] Biezunski M., Space and Time in HyTime, in <TAG> *The SGML Newsletter*, **5:10**, pp. 1-5, SGML Associats Inc. & Graphic Communications Association, 1992.
- [Blake 91] Blake E., *(Anti-)Declarative Propaganda*, proceedings of the Second Eurographics Workshop on Object-Oriented Graphics, pp. 215-216, The Eurographics Association, 1991.
- [Blake et al. 95] Blake E., Codognet P., Duce D., Duke D., Gorgan D., Hagen P.J.W. ten, Herman I., Hintum J.E.A. van, Hopgood F.R.A., Kansy K., Reynolds G.J., Ruttkay Z., Veltkamp R., Vieira A.S., *ERCIM Constraint Management Workshop Report*, ERCIM, 1995.
- [Bordegoni 92] Bordegoni M., *Multimedia in Views*, tech.rep. CS-9263, CWI, 1992.
- [Borning 81] Borning A., The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory, in *ACM Transactions on Programming Languages and Systems*, **3:4**, pp. 353-387, ACM Press, 1981.

- [Borning 86] Borning A., Classes versus Prototypes in Object-Oriented Languages, in *Proceedings of the IEEE/ACM Fall Joint Computer Conference*, pp. 36-40, ACM Press, 1986.
- [Borning et al. 86] Borning A., Duisberg R., Constraint-Based Tools for Building User Interfaces, in *ACM Transactions on Graphics*, **5:4**, pp. 345-374, ACM Press, 1986.
- [Borning et al. 87] Borning A., Duisberg R., Freeman-Benson B., Kramer A., Woolf M., Constraint Hierarchies, proceedings OOPSLA'87, in *SIGPLAN notices*, **22:12**, pp. 48-60, ACM Press, 1987.
- [Braden et al. 96] Braden R., Zhang L., Berson S., Herzog S., Jamin S., *Resource ReSerVation Protocol (RSVP) -- Version 1 Functional Specification*, internet draft "draft-ietf-rsvp-spec-13.txt", 1996.
- [Bulterman 94] Bulterman D.C.A., Managing the Adaptive Processing of Distributed Multimedia Information, in *CWI Quarterly*, **7:1**, pp. 3-25, CWI, 1994.
- [CIE 17.4] Central Bureau of the Commission Internationale de L'Éclairage, Publication CIE N° 17.4.
- [Correia et al. 95] Correia N., Duce D.A., Duke D., Hagen P.J.W. ten, Herman I., Hintum J.E.A. van, Reynolds G.J., *Proposal: New Synchronization Object Types for PREMO Part2*, tech.rep. ISO/IEC-JTC-1/SC24/WG6 OME-115, ISO, 1995.
- [Cournarie et al. 91] Cournarie E., Beaudouin-Lafon M., *ALIEN: a Prototype-Based Constraint System*, in proceedings of the Second Eurographics Workshop on Object-Oriented Graphics, pp. 93-114, The Eurographics Association, 1991.
- [DAVIC-1 95] Digital Audio-Visual Council, Part 1: *Description of DAVIC Functionalities*, revision 3.1, draft version, DAVIC, 1995.
- [DAVIC-2 95] Digital Audio-Visual Council, Part 2: *System Reference Models and Scenarios*, revision 3.0, draft version, DAVIC, 1995.
- [DAVIC-3 95] Digital Audio-Visual Council, Part 3: *Server Architecture and APIs*, revision 3.0, draft version, DAVIC, 1995.
- [DAVIC-4 95] Digital Audio-Visual Council, Part 4: *Delivery System Architectures and APIs*, revision 3.1, draft version, DAVIC, 1995.
- [DAVIC-6 95] Digital Audio-Visual Council, Part 6: *High Layer Protocol*, revision 2.1, draft version, DAVIC, 1995.
- [DAVIC-7 95] Digital Audio-Visual Council, Part 7: *Mid-Layer Protocols*, revision 3.1, draft version, DAVIC, 1995.
- [DAVIC-8 95] Digital Audio-Visual Council, Part 8: *Lower Layer Protocols and Physical Interfaces*, revision 3.1, draft version, DAVIC, 1995.

- 
- [DAVIC-9 95] Digital Audio-Visual Council, Part 9: *Information Representation*, revision 3.1, draft version, DAVIC, 1995.
- [DAVIC-11 95] Digital Audio-Visual Council, Part 11: *Usage Information Protocols*, revision 3.1, draft version, DAVIC, 1995.
- [Davis 87] Davis E., Constraint Propagation with Interval Labels, in *Artificial Intelligence*, **32**, pp. 281-31, Elsevier Science Publishers B.V., 1987.
- [Degener 94] Degener J., Digital Speech Compression, in *Dr. Dobb's Journal*, **19:15**, pp. 30-34, M&T Publishing, 1994.
- [Delgrossi et al. 94] Delgrossi L., Halstrick C., Hehmann D., Herrtwich R.G., Krone O., Sandvoss J. Vogt C., Media Scaling in a Multimedia Communication System, in *Multimedia Systems*, **2:4**, pp. 172-180, Springer Verlag, 1994.
- [Delgrossi et al. 95] Delgrossi L., Berger L., *Internet Stream Protocol Version 2 (ST2), Protocol Specification - Version ST2+*, rfc1819, 1995.
- [Demeure et al. 96] Demeure I., Farhat-Gissler J., Gasperoni F., *A Scheduling Framework for the Automatic Support of Temporal QoS Constraints*, in proceedings of the Fourth International Workshop on QoS'96, pp. 127-138 , GMD, 1996.
- [DeRose et al. 94] DeRose S., Durand D.G., *Making Hypermedia work: a user's guide to HyTime*, ISBN 0-7923-9432-1, Kluwer, 1994.
- [Dincbas et al. 88] Dincbas M., Hentenryck P. van, Simonis H., Aggoun A., Graf T., Berthier F., The Constraint Logic Programming Language CHIP, proceedings of the International Conference on Fifth Generation Computer Systems, in *FGCS'88*, **3**, pp. 693-702, Springer Verlag, 1988.
- [Fox 92] Fox E., Multimedia: Application and Practice, Tech.rep ISSN 1017-4656, EG92 TN6, Eurographics Technical Report Series, 1992.
- [Freeman-Benson 90] Freeman-Benson B.N., Kaleidoscope: Mixing Objects, Constraints and Imperative Programming, proceedings ECOOP/OOPSLA'90, in *SIGPLAN notices*, **25:10**, pp. 77-88, ACM Press, 1990.
- [Freeman-Benson et al. 90] Freeman-Benson B.N., Maloney J., Borning A., An Incremental Constraint Solver, in *Communications of the ACM*, **33:1**, pp. 54-63, ACM Press, 1990.
- [Freuder 78] Freuder E.C., Synthesizing Constraint Expressions, in *Communications of the ACM*, **21:11**, pp. 958-966, ACM Press, 1978.
- [Frølund 92] Frølund S., Inheritance of Synchronization Constraints in Concurrent Object-Oriented Programming Languages, proceedings ECOOP'92, in *Lecture Notes in Computer Science* **615**, ISBN 3-540-55668-0/0-387-55668-0, pp. 185-196, Springer Verlag, 1992.

- [Gamma et al. 93] Gamma E., Helm R., Johnson R., Vlissides J., Design Patterns: Abstraction and Reuse of Object-Oriented Design, proceedings ECOOP'93, in *Lecture Notes in Computer Science 707*, ISBN 3-540-57120-5/0-387-57120-5, pp. 406-42 1, Springer Verlag, 1993.
- [Gehani et al. 90] Gehani, N., Roome W.D., *The Concurrent C Programming Language*, ISBN 0-13-170820-1, Silicon Press, 1990.
- [Gibbs et al. 92] Gibbs S., Dami L., Tsichritzis D., An Object-Oriented Framework for Multimedia Composition and Synchronization, in *Multimedia - Systems, Interaction and Applications*, EurographicsSeminar Series, pp. 100-111, Springer Verlag, 1992.
- [Gibs 91] Gibs S., Composite Multimedia and Active Objects, proceedings OOPSLA'91, in *SIGPLAN notes*, **26:11**, pp. 97-112, ACM Press, 1991.
- [Goldberg 84] Goldberg A., *Smalltalk-80: The Interactive Programming Environment*, Addison-Wesley, 1984.
- [Gosling 83] Gosling J., *Algebraic Constraints*, Ph.D. thesis, Carnegie-Mellon University, 1983.
- [Graf et al. 89] Graf T., Hentenryck P. van, Pradelles C., Simulation of Hybrid Circuits in Constraint Logic Programming, in *International Joint Conference on Artificial Intelligence*, ISBN 1-558860-094-9, pp. 72-77, Morgan-Kaufmann Publishers Inc., 1989.
- [Guimarães et al. 92a] Guimarães N., Correia N., Carmo T., Programming Time in Multimedia User Interfaces, in *proceedings of the UIST'92 Conference*, 1992.
- [Guimarães et al. 92b] Guimarães N., Correia N., Carmo T., Xt based Support for Continuous Media, in *proceedings of the Xhibition'92 Conference*, 1992.
- [Hardman et al. 92] Hardman L., Bulterman D.C.A., Rossum G. van, *The Amsterdam Hypermedia Model, extending Hypertext to support Real Multimedia*, tech.rep. CS-9306, CWI, 1992.
- [Heeman et al. 93] Heeman F.C, Herman I., Reynolds G.J., Ruiter M.M de, *Implementation Specification of the MADE mC++ language*, tech.rep. T/OM-S.1, CWI, 1993.
- [Helm et al. 90] Helm R., Holland I., Gangopadhyay D., Contracts: Specifying Behavioral Compositions in Object-Oriented Systems, proceedings ECOOP/OOPSLA'90, in *SIGPLAN notices*, **25:10**, pp. 169-180, ACM Press, 1990.
- [Hentenryck 89] Hentenryck P. van, A Logic Language for Combinatorial Optimization, in *Annals of Operations Research*, **21**, pp. 247-274, J.C. Baltzer A.G. Scientific Publishing Company, 1989.

- 
- [Hentenryck 91] Hentenryck P. van, Constraint Logic Programming, in *The Knowledge Engineering Review*, **6:3**, pp. 151-194, Cambridge Press, 1991.
- [Hentenryck et al. 91] Hentenryck P. van, Simonis H., Dincbas M., *Constraint Satisfaction using Constraint Logic Programming*, tech.rep. CS-91-62, Brown University, 1991.
- [Herman et al. 94] Herman I., Reynolds G.J., Davy J., MADE: A Multimedia Application Development Environment, in *proceedings of the IEEE International Conference on Multimedia Computing and Systems*, IEEE CS Press, 1994.
- [Hilton et al. 94] Hilton M.L., Jawerth B.D., Sengupta A., Compressing Still and Moving Images with Wavelets, in *Multimedia Systems*, **2:5**, pp. 218-227, Springer Verlag, 1994.
- [Hintum et al. 95] Hintum J.E.A. van, Reynolds G.J., A Multimedia Constraint System (or: do we have it MADE), proceedings EuroGraphics'95, in *Computer Graphics Forum*, pp. 135-148, The Eurographics Association, 1995.
- [Hintum 95] Hintum J.E.A. van, Unconstrained Constraint Programming, in *CWI Quarterly*, **8:4**, pp. 329-354, CWI, 1995.
- [Hodges et al. 89] Hodges M., Sasnett R., Ackerman M., A Construction Set for Multimedia Applications, *IEEE Computer*, **22**, pp. 37-43, IEEE CS Press, 1989.
- [Hoepner 92] Hoepner P., Synchronizing the presentation of multimedia objects - ODA extensions, in *Multimedia - Systems, Interaction and Applications*, EurographicsSeminar Series, pp. 87-100, Springer Verlag, 1992.
- [Hölzle 93] Hölzle U., Integrating Independently-Developed Components in Object-Oriented Languages, proceedings ECOOP'93, in *Lecture Notes in Computer Science 707*, ISBN 3-540-57120-5/0-387-57120-5, pp. 36-56, Springer Verlag, 1993.
- [Horn 92] Horn B., Constraint Patterns As a Basis For Object Oriented Programming, proceedings OOPSLA'92, in *SIGPLAN notices*, **27:10**, pp. 218-234, ACM Press, 1992.
- [Hutschenreuther et al. 96] Hutschenreuther T., Kümmel S., *INA-QoS T-Integrated Network Architecture for QoS-based Transmission in Heterogeneous Environments*, in proceedings of the Fourth International Workshop on QoS'96, pp. 55-66, GMD, 1996.
- [IMA 92] Interactive Multimedia Association, *Request for Technology: Multimedia Systems Services*, draft for comments, IMA, 1992.
- [ISO-N9309 95] ISO/IEC-JTC-1/SC 21, *Open Systems Interconnection, Data Management and Open Distributed Processing: QoS - BASIC FRAMEWORK*, committed draft paper N9309, ISO, 1995.



- [ISO-N9681 95] ISO/IEC-JTC-1/SC 21, *Open Systems Interconnection, Data Management and Open Distributed Processing: Quality of Service - METHODS AND MECHANISMS* working draft paper N9681, ISO, 1995.
- [ISO-N1190 95] ISO/IEC-JTC-1/SC 24, *Information Processing Systems - Computer Graphics and Image Processing - Presentation Environments for Multimedia Objects (PREMO)*, committed draft paper N1190, ISO, 1995.
- [ITU-R BT.709 90] ITU, *Basic Parameter Values for the HDTV Standard for the Studio and for International Programme Exchange*, ITU-R Recommendation BT.709, ITU, 1990.
- [JPEG 93] ISO/IEC-JTC-1, *Digital Compression and Coding of Continuous-Tone Still Images*, International Standard ISO/IEC IS 10918, ISO, 1993.
- [Kayargadde et al. 94] Kayargadde V., Martens J.B., Estimation of Edge Parameters and Image Blur Using Polynomial Transforms, in *Graphical Models and Image Processing*, **56:6**, pp. 442-461, Academic Press, 1994.
- [Kayargadde et al. 96a] Kayargadde V., Martens J.B., An Objective Measure for Perceived Noise, in *Signal Processing*, **49**, pp. 187-206, Elsevier Science B.V., 1996.
- [Kayargadde et al. 96b] Kayargadde V., Martens J.B., Perceptual Characterization of Images Degraded by Blur and Noise: Experiments, in *Journal of the Optical Society of America*, **13:6**, pp. 1166-1177, Optical Society of America, 1996.
- [Kayargadde et al. 96c] Kayargadde V., Martens J.B., Perceptual Characterization of Images Degraded by Blur and Noise: Model, in *Journal of the Optical Society of America*, **13:6**, pp. 1178-1188, Optical Society of America, 1996.
- [Knightly et al. 96] Knightly E.W., Rossaro P., *Improving QoS through Traffic Smoothing*, in proceedings of the Fourth International Workshop on QoS'96, pp. 219-231, GMD, 1996.
- [Koegel 94] Koegel Buford J.F., Architectures and Issues for Distributed Multimedia Systems, in *Multimedia Systems*, ISBN 0-201-53258-1, pp. 45-64, Addison Wesley Publishing Company, 1994.
- [Krueger 92] Krueger C.W., Software Reuse, in *ACM Computing Surveys*, **24:2**, pp. 131-183, ACM Press, 1992.
- [Laffra et al. 91] Laffra C., Bos J. van den, Propagators and Concurrent Constraints, proceedings of the Object Based Concurrent Systems Workshop at ECOOP/OOPSLA '90, in *OOPS Messenger*, **2:2**, pp. 68-72, 1991.
- [Laffra 92] Laffra C., *PROCOL: A Concurrent Object Language with Protocols, Delegation, Persistence and Constraints*, Ph.D. thesis, University of Rotterdam, 1992.

- 
- [Lassez 87] Lassez C., Constraint Logic Programming, in *Byte*, **12:9**, pp. 171-176, McGraw-Hill Inc., 1987.
- [Leger et al. 91] Leger A., Omachi T., Wallace G.K., JPEG still picture compression algorithm, in *Optical Engineering*, **30**, pp. 947-954, SPIE, 1991.
- [Leler 88] Leler W., *Constraint Programming Languages: their Specification and Generation*, ISBN 0-201-06243-7, Addison-Wesley, 1988.
- [Leydekkers et al. 96] Leydekkers P., Gay V., *ODP View and Quality of Service for Open Distributed Multimedia Environments*, in proceedings of the Fourth International Workshop on QoS'96, pp. 31-43, GMD, 1996.
- [Lieberman 86] Lieberman H., Using Prototypical Objects to implement Shared Behaviour in Object-Oriented Systems, in *proceedings OOPSLA'86*, pp. 214-223, ACM Press, 1986.
- [Lima et al. 96] Lima, F.H.S., Madeira, E.R.M., *ODP-based QoS Specification for the Multiware Platform*, in proceedings of the Fourth International Workshop on QoS'96, pp. 45-54, GMD, 1996.
- [Little 94] Little T.D.C., Time-Based Media Representation and Delivery, in *Multimedia Systems*, ISBN 0-201-53258-1, pp. 175-200, Addison Wesley Publishing Company, 1994.
- [Mackworth 77] Mackworth A.K., Consistency in Networks of Relations, in *Artificial Intelligence*, **8**, pp. 99-118, North-Holland Publishing Company, 1977.
- [Maloney et al. 89] Maloney J.H., Borning A., Freeman-Benson B., Constraint Technology for User-Interface Construction in ThingLab II, proceedings OOPSLA'89, in *SIGPLAN notices*, **24:10**, pp. 381-388, ACM Press, 1989.
- [May et al. 92] May V. de, Breiteneder C., Dami L., Gibbs S., Tschritzis D., Visual Composition and Multimedia, proceedings EuroGraphics'92, in *Computer Graphics Forum*, **11:3**, pp. 9-21, The Eurographics Association, 1992.
- [Microsoft 94] Microsoft Corporation, Digital Equipment Corporation, *Common Object Model Specification*, draft version, 1994.
- [Mili 92] Mili H., SoftClass: An Object-Oriented Tool for Software-Reuse, proceedings TOOLS USA'92, in *TOOLS 5*, ISBN 0-13-923178-1, pp. 303-317, Prentice Hall, 1992.
- [Mitchell et al. 91] Mitchell J.L., Pennebaker W.B., Evolving JPEG color data compression standard, in *Standards for Electronic Imaging Systems*, SPIE CR, **37**, pp. 68-97, SPIE, 1991.
- [Morley et al. 91] Morley D., Chiu S., Robbins J., Madux T., Voelker G., Reusable Objects, proceedings TOOLS USA'91, in *TOOLS 4*, ISBN 0-13-923160-9, pp. 237-248, Prentice Hall, 1991.

- [MPEG 94] A Generic Standard for Coding of High-Quality Digital Audio, in *Journal of the Audio Engineering Society*, **42:10**, pp. 780-792, 1994.
- [Nahrstedt et al. 96] Nahrstedt K., Hossain A., Kang S., *A Probe-based Algorithm for QoS Specification and Adaptation*, in proceedings of the Fourth International Workshop on QoS'96, pp. 89-100, GMD, 1996.
- [Naiditch 95] Naiditch D.J., *Rendezvous with Ada 95*, ISBN 0-471-01276-9, Wiley, 1995.
- [Nelson 85] Nelson G., Juno, a Constraint-Based Graphics System, proceedings SIGGRAPH'85, in *Computer Graphics*, **19:3**, pp. 235-243, ACM Press, 1985.
- [Newcomb et al. 91] Newcomb S., Kipp N., Newcomb V., The HyTime - Hypermedia / Time-based document structuring language, *Communications of the ACM*, **34**, pp. 67-83, ACM Press, 1991.
- [Nijenhuis et al. 93] Nijenhuis M.R.M., Blommaert F.J.J., Perceptual Error Measure for Sampled and Interpolated Imagery, proceedings EuroDisplays, in ..., pp. 135-138, 1993.
- [OMG 95] OMG Group, *CORBA 2.0/Interoperability, Universal Networked Objects*, tech.rep. OMG TC 95.3.xx, March OMG, 1995.
- [Par et al. 94] Par S.L.J.D.E. van de, Kate W.R.Th ten, Kohlraush A., Houtsma A.J.M., Bit-Rate saving in Multichannel Sound: Using a Band-Limited Channel to Transmit the Center Signal, in *Journal of the Audio Engineering Society*, **42:7/8**, pp. 555-564, 1994.
- [Pree 94] Pree W., Meta Patterns - A Means for Capturing the Essentials of Reusable Object-Oriented Design, proceedings ECOOP'94, in *Lecture Notes in Computer Science 821*, ISBN 3-540-58202-9/0-387-58202-9, pp. 150-162, Springer Verlag, 1991.
- [Price 93] Price R., MHEG: An Introduction to the future International Standard for Hypermedia Object Interchange, in *ACM Multimedia 93*, pp. 121-128, ACM Press, 1993.
- [Rajagopal et al. 96] Rajagopal M., Sergeant S., *Internet Stream Protocol Version 2 (ST2), Protocol State Machine - Version ST2+*, internet draft "draft-ietf-st2-state-02.txt", 1996.
- [Rankin 91] Rankin J.R., *A Graphics Object Oriented Constraint Solver*, proceedings of the Second Eurographics Worskhop on Object-Oriented Graphics, pp. 93-114, The Eurographics Association, 1991.
- [Rogowitz et al. 92] Rogowitz B.E., Ling D.T., Kellogg W.A., Task Dependence, Veridicality, and Pre-Attentive Vision: Taking Advantage of Perceptually-Rich Computer Environments, in *Human Vision, Visual Processing and Digital Display III*, **1666**, pp. 504-513, 1992.

- 
- [Rolia et al. 96] Rolia J., Richard F., *Quality of Service Management for Federated Applications*, in proceedings of the Fourth International Workshop on QoS'96, pp. 259-270, GMD, 1996.
- [Roufs 92] Roufs J.A.J., Perceptual Image Quality: Concept and Measurement, in *Philips Journal of Research*, **47**, pp. 35-62, 1992.
- [Roufs et al. 94] Roufs J.A.J., Koselka V.J.F., Tongeren A.A.A.M. van, *Global Brightness and the Effect on Perceptual Image Quality*, tech.rep. 983/III, IPO, 1992.
- [Saraswat 93] Saraswat V.A., *Concurrent Constraint Programming*, ISBN 0-262-19297-7, M.I.T. Press, 1993.
- [Shaw et al. 96] Shaw R., Laplante P.A., Salinas J., Riccone R., A Multimedia Speech Learning System for the Hearing Impaired, in *Multimedia Tools and Applications*, **3:1**, pp. 55-70, Kluwer Academic Publishers, 1996.
- [Shenker et al. 95a] Shenker S., Wroclawski J., *Network Element Service Specification Template*, internet draft "draft-ietf-interserv-svc-template-02.txt", 1995.
- [Shenker et al. 95b] Shenker S., Partridge C., Davi B., Breslau L., *Specification of Predictive Quality of Service*, internet draft "draft-ietf-interserv-predictive-svc-01.txt", 1995.
- [Shenker et al. 95c] Shenker S., Partridge C., Wroclawski J., *Specification of Controlled Delay Quality of Service*, internet draft "draft-ietf-interserv-control-del-svc-02.txt", 1995.
- [Shenker et al. 95d] Shenker S., Partridge C., *Specification of Guaranteed Quality of Service*, internet draft "draft-ietf-interserv-guaranteed-svc-03.txt", 1995.
- [Shenker 95] Shenker S., *Specification of General Characterization Parameters*, internet draft "draft-ietf-interserv-charac-00.txt", 1995.
- [Staehli et al. 95] Staehli R., Walpole J., Maier D., A Quality-of-Service Specification for Multimedia Presentations, in *Multimedia Systems*, **3:5/6**, pp. 251-263, Springer Verlag, 1995.
- [Steinmetz 94a] Steinmetz R., Data Compression in Multimedia Computing - Principles and Techniques, in *Multimedia Systems*, **1:4**, pp. 166-172, Springer Verlag, 1994.
- [Steinmetz 94b] Steinmetz R., Data Compression in Multimedia Computing - Standards and Systems, in *Multimedia Systems*, **1:5**, pp. 187-204, Springer Verlag, 1994.
- [Sussman et al. 80] Sussman G.J., Steele Jr. G.L., CONSTRAINTS - A Language for Expressing Almost-Hierarchical Descriptions, in *Artificial Intelligence*, **14**, pp. 1-39, North-Holland Publishing Company, 1980.

- [Teunissen et al. 96] Teunissen K., Westerink J.H.D.M., A Multidimensional Evaluation of the Perceptual Quality of Television Sets, in *SMPTE Journal*, pp. 31-38, 1996.
- [Tsang 93] Tsang E., *Foundations of Constraint Satisfaction*, ISBN 0-12-701610-4, Academic Press, 1993.
- [Verall 91] Verall M.S., Unity Doesn't Imply Unification or Overcoming Heterogeneity Problems in Distributed Software Engineering Environments, in *The Computer Journal*, **34:6**, pp. 522-533, Cambridge University Press, 1991.
- [Veltkamp et al. 92] Veltkamp R.C., Arbab F., Geometric Constraint Satisfaction with Quantum Labels, in *Computer Graphics and Mathematics*, pp. 211-228, Springer Verlag, 1992.
- [Vogt 95] Vogt C., Quality-of-Service management for Multimedia Streams with Fixed Arrival Periods and Variable Frame Sizes, in *Multimedia Systems*, **3:2**, pp. 66-75, Springer Verlag, 1995.
- [Wallace 91] Wallace G.K., The JPEG still picture compression standard, in *Communications of the ACM*, **34:4**, pp. 30-44, ACM Press, 1991.
- [Wilk 91] Wilk M.R., Equate: An Object-Oriented Constraint Solver, proceedings OOPSLA'91, in *SIGPLAN notes*, **26:11**, pp. 286-298, ACM, 1991.
- [Yankelovich 94] Yankelovich N., *Talking vs. Talking: Speech Access to Remote Computers*, in proceedings companion CHI'94, pp. 275-277, ACM Press, 1994.
- [Yankelovich et al. 94] Yankelovich N., Baatz E., *SpeechActs: A Framework for Building Speech Applications*, proceedings AVIOS'94, 1994.
- [Yankelovich et al. 95] Yankelovich N., Levow G.A., Marx M., *Designing SpeechActs: Issues in Speech User Interfaces*, in proceedings CHI'95, pp. 369-376, ACM Press, 1995.
- [Zhang et al. 91] Zhang Y., Mackworth A.K., *Parallel and Distributed Algorithms for Constraint Networks*, tech.rep. TR 91-6, University of British Columbia, 1991.

# About the Author



Hans van Hintum was born on March 7, 1969 in Grave. He attended the Kruisheren Kollege in Uden from 1981 to 1987 and started to study Computer Science at the University of Nijmegen. He received his Master's Degree with credit in August 1992. Afterwards, he joined the department of Interactive Systems at the CWI (Centrum voor Wiskunde en Informatica) in Amsterdam. During this time, he was a member of the Esprit III project 6307: MADE (Multimedia Application Development Environment) and as such he was responsible for most of the design and the complete implementation of the MADE constraint system. Furthermore, he has participated in several international workshops regarding the PREMO standard (PRogramming Environment for Multimedia Objects, ISO/IEC Joint Technical Committee 1, Sub Committee 24). Since February 1997, he is attached to the Dutch National Aerospace Laboratory, the NLR (Nationaal Lucht- en Ruimtevaartlaboratorium).

